TEXAS INSTRUMENTS

MSP430 Family
Mixed-Signal Microcontrollers

Application
Reports

2000

2000   **Analog and Mixed-Signal**

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2000, Texas Instruments Incorporated

MSP430 Family
Mixed-Signal Microcontroller
Application Reports

Author: Lutz Bierl

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.
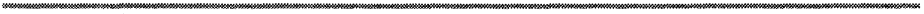
TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Read This First

How to Use This Manual

This document contains the following chapters:

❏ Chapter 1 – MSP430 Microcontroller Family: Introduction to the MSP430 family, advantages of the MSP430 concept, and operating modes

❏ Chapter 2 – MSP430 14-Bit Analog-To-Digital Converter:

❏ Chapter 3 – MSP430 Hardware Applications:

❏ Chapter 4 – MSP430 Application Examples:

❏ Chapter 5 – Software Applications:

❏ Chapter 6 – On-Chip Peripherals:

❏ Chapter 7 – Hints and Recommendations:

❏ Chapter 8 – Architecture and Instruction Set:

❏ Chapter 9 – CPU Registers:

Contents

Figures

Tables

Examples

MSP430 Microcontroller Family

1.1 Introduction

The MSP430 is a 16-bit microcontroller that has a number of special features not commonly available with other microcontrollers:

❑ Complete system on-a-chip — includes LCD control, ADC, I/O ports, ROM, RAM, basic timer, watchdog timer, UART, etc.

❑ Extremely low power consumption — only 4.2 nW per instruction, typical

❑ High speed — 300 ns per instruction @ 3.3 MHz clock, in register and register addressing mode

❑ RISC structure — 27 core instructions

❑ Orthogonal architecture (any instruction with any addressing mode)

❑ Seven addressing modes for the source operand

❑ Four addressing modes for the destination operand

❑ Constant generator for the most often used constants (–1, 0, 1, 2, 4, 8)

❑ Only one external crystal required — a frequency locked loop (FLL) oscillator derives all internal clocks

❑ Full real-time capability — stable, nominal system clock frequency is available after only six clocks when the MSP430 is restored from low-power mode (LPM) 3; — no waiting for the main crystal to begin oscillation and stabilize

The 27 core instructions combined with these special features make it easy to program the MSP430 in assembler or in C, and provide exceptional flexibility and functionality. For example, even with a relatively low instruction count of 27, the MSP430 is capable of emulating almost the complete instruction set of the legendary DEC PDP-11.

Note:

The software examples provided in this document have been tested for functionality and may be used freely for system development.

1.2 Related Documents

The following documents are recommended for MSP430 reference:

❑ The *MSP430 Architecture User's Guide and Module Library* (TI literature number SLAUE10B) contains a detailed hardware description.

❑ The *MSP430 Software User's Guide* (TI literature number SLAUE11) contains further information regarding the instruction set, plus other more common software information.

1.3 Notation

The following abbreviations and special notations are used:

.and.	Logical AND function
.not.	Logical Inversion
.or.	Logical OR function
.xor.	Logical Exclusive-OR function
[ns]	Square brackets contain the unit for a value (here nanoseconds)
ACLK	Auxiliary clock (output of the 32-kHz oscillator)
ACTL.1	Bit 1 (value 2^1) of the register ACTL
ADC	Analog-to-digital converter
AGND	Ground connection for the ADC; Vss (MSP430x31x) or AVss (MSP430x32x)
Background	Normal program
BCD	Binary coded decimal (numbers 0 to 9 coded binary with 4 bits)
CPU	Central processing unit
DCO	Digitally controlled oscillator
(dst)	Destination (location receiving write data)
Foreground	Interrupt driven software parts (interrupt handlers)
I/O	Input and output Port
LCD	Liquid crystal display
LSB	Least significant bit (or byte)
MCLK	Master clock (output of the FLL oscillator) for the CPU
MSB	Most significant bit (or byte)
PC	Program counter (R0 of register set)
R1‖R2	Resistor R1 is connected in parallel with resistor R2
R4‖R3	32-bit number. MSBs in CPU register R4, LSBs in R3
RAM	Random access memory (data memory)

ROM	Read only memory (program memory)
SP	Stack pointer (R1 of register set)
(src)	Source (location supplying read data)
TOS	Top of stack (data word the Stack Pointer SP points to)

NOTES:If no units are defined for equations, the following standard units are used: Volt, Ampere, Farad, seconds and Ohm.

1.4 MSP430 Family

The MSP430 family currently consists of three subfamilies:

❑ MSP430C31x

❑ MSP430C32x

❑ MSP430C33x

All three are described in detail in the *MSP430 Family Architecture User's Guide and Module Library.* The hardware features of the different devices are shown in Table 1, Figure 1, Figure 2, and Figure 3.

Table 1–1. MSP430 Sub-Families Hardware Features

Hardware Item	MSP430C31x	MSP430C32x	MSP430C33x
14-bit ADC	No	Yes	No
16-bit timer_A	No	No	Yes
Basic timer	Yes	Yes	Yes
FLL oscillator	Yes	Yes	Yes
HW/SW UART	Yes	Yes	Yes
HW-multiplier	No	No	Yes
I/O ports with interrupt	8	8	24
I/O ports without interrupt	0	0	16
LCD segment lines	23	21	30
Package	56 SSOP	64 QFP	100 QFP
Universal timer/port module	Yes	Yes	Yes
USART (SCI or SPI)	No	No	Yes
Watchdog timer	Yes	Yes	Yes

NOTE: Examples and explanations in this document are applicable for all MSP430 devices, unless otherwise noted.

1.4.1 MSP430C31x
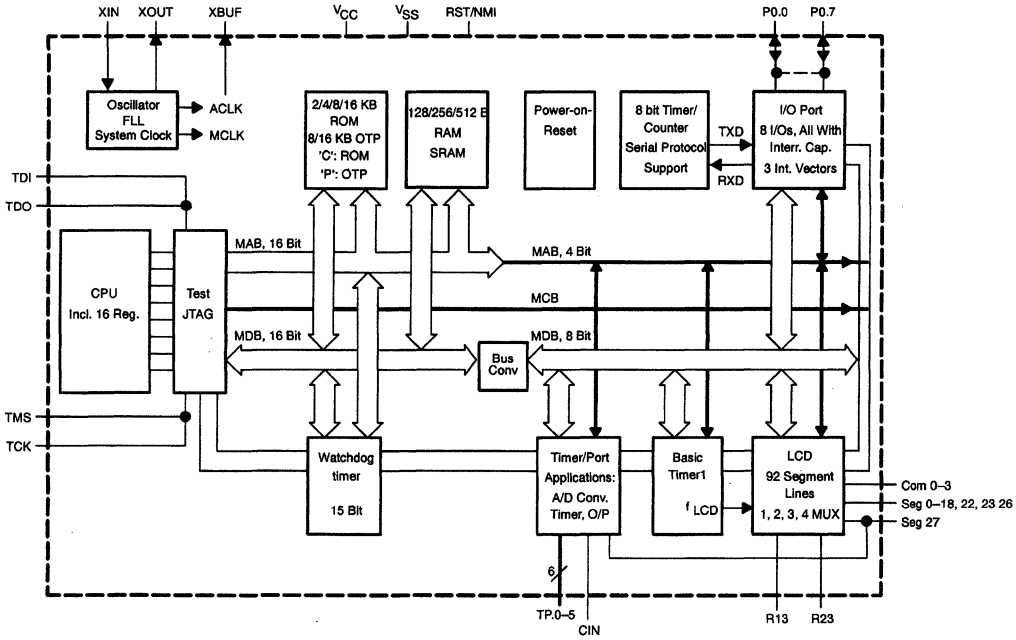
Figure 1–1. MSP430C31x Block Diagram

1.4.2 MSP430C32x
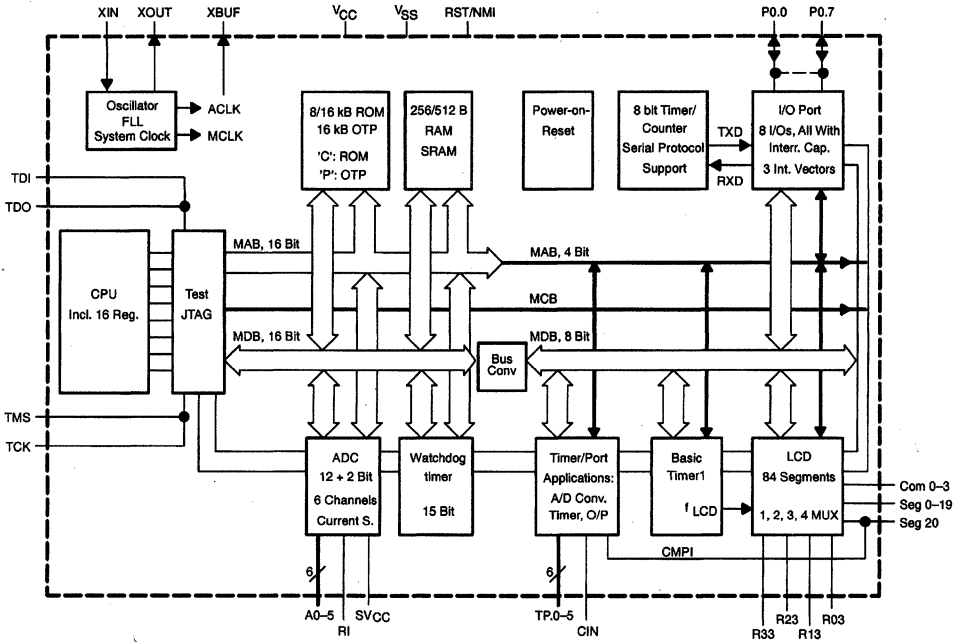
Figure 1–2. MSP430C32x Block Diagram

1.4.3 MSP430C33x

Figure 1–3. MSP430C33x Block Diagram
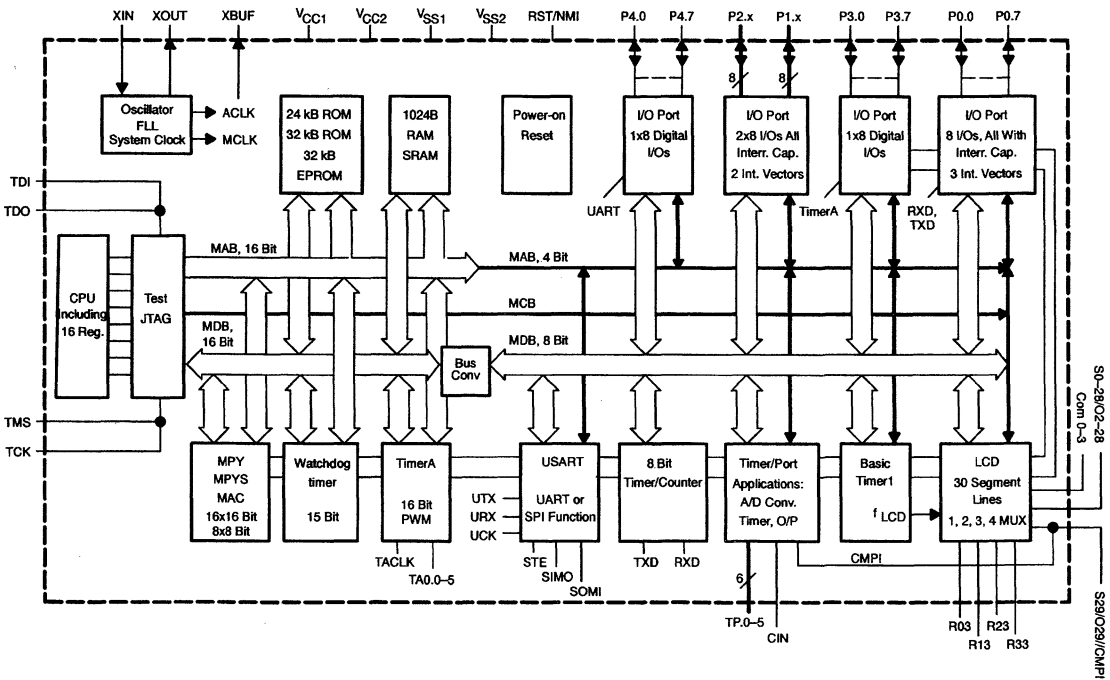
1.5   Advantages of the MSP430 Concept

The MSP430 concept differs considerably from other microcontrollers and offers some significant advantages over more traditional designs.

1.5.1   RISC Architecture Without RISC Disadvantages

Typical RISC architectures show their highest performance in calculation- intensive applications in which several registers are loaded with input data, all calculations are made within the registers, and the results are stored back into RAM. Memory accesses (using addressing modes) are necessary only for the LOAD instructions at the beginning and the STORE instructions at the end of the calculations. The MSP430 can be programmed for such operation, for example, performing a pure calculation task in the floating point without any I/O accesses.

Pure RISC architectures have some disadvantages when running real-time applications that require frequent I/O accesses, however. Time is lost whenever an operand is fetched and loaded from RAM, modified, and then stored back into RAM.

The MSP430 architecture was designed to include the best of both worlds, taking advantage of RISC features for fast and efficient calculations, and addressing modes for real-time requirements:

❑   The RISC architecture provides a limited number of powerful instructions, numerous registers, and single-cycle execution times.

❑   The more traditional microcomputer features provide addressing modes for *all* instructions. This functionality is further enhanced with 100% orthogonality, allowing any instruction to be used with any addressing mode.

1.5.2   Real-Time Capability With Ultra-Low Power Consumption

The design of the MSP430 was driven by the need to provide full real-time capability while still exhibiting extremely low power consumption. Average power consumption is reduced to the minimum by running the CPU and certain other functions of the MSP430 only when it is necessary. The rest of the time (the majority of the time), power is conserved by keeping only selected low-power peripheral functions active.

But to have a true real-time capability, the device must be able to shift from a low-power mode with the CPU off to a fully active mode with the CPU and all other device functions operating nominally in a very short time. This was accomplished primarily with the design of the system clock:

❏ No second high frequency crystal is used — inherent delays can range from 20 ms to 200 ms until oscillator stability is reached

❏ Instead, a sophisticated FLL system clock generator is used — generator output frequency (MCLK) reaches the nominal frequency within 8 cycles after activation from low power mode 3 (LPM3) or sleep mode

This design provides real-time capability almost immediately after the device comes out of a LPM — as if the CPU is always active. Only two additional MCLK cycles (2 μs @ f_C = 1 MHz) are necessary to get the device from LPM3 to the first instruction of the interrupt handler.

1.5.3 Digitally Controlled Oscillator Stability

The digitally controlled oscillator (DCO) is voltage and temperature dependent, which does not mean that its frequency is not stable. During the active mode, the integral error is corrected to approximately zero every 30.5 μs . This is accomplished by switching between two different DCO frequencies. One frequency is higher than the programmed MCLK frequency and the other is lower, causing the errors to essentially cancel-out. The two DCO frequencies are interlaced as much as possible to provide the smallest possible error at any given time. See *System Clock Generator* for more information.

1.5.4 Stack Processing Capability

The MSP430 is a true stack processor, with most of the seven addressing modes implemented for the stack pointer (SP) as well as the other CPU registers (PC and R4 through R15). The capabilities of the stack include:

❏ Free access to all items on the stack — not only to the top of the stack (TOS)

❏ Ability to modify subroutine and interrupt return addresses located on the stack

❏ Ability to modify the stored status register of interrupt returns located on the stack

❏ No special stack instructions — all of the implemented instructions may be used for the stack and the stack pointer

❏ Byte and word capability for the stack

❏ Free mix of subroutine and interrupt handling — as long as no stack modification (PUSH, POP, etc.) is made, no errors can occur

For more information concerning the stack, see *Appendix A.*

1.6 MSP430 Application Operating Modes

MSP430 applications fall into two main classes, depending on the power supply:

❑ AC power-driven applications such as electricity meters and AC-powered controllers. In these applications, the microcontroller needs to be active at all times. The low current consumption of the MSP430 when active (900 μA @ 5 V & f_C = 1 MHz) puts it well within the typical low-power category now (currently < 40 mA) and in the future as tolerable current consumption diminishes.

❑ Battery-powered applications such as gas meters, water flow meters, heat volume counters, data loggers, and other controller and remote metering tasks. For these applications, power consumption is the key issue since operation from a single battery for 10 years or longer is often required. The average current drawn by the MSP430 needs to be in the range of the self discharge current of the battery, approximately 1 μA to 3 μA.

MSP430 has six operating modes, each with different power requirements. Three of these modes are important for battery-powered applications:

❑ Active mode — CPU and other device functions run all the time

❑ Low power mode 3 (LPM3) — the normal mode for most applications during 99% to 99.9% of the time. This mode is also called done mode or sleep mode

❑ Low power mode 4 (LPM4) — the mode typically used during storage. This mode is also called off mode

1.6.1 Active Mode

Active mode is used for calculations, decision-making, I/O functions, and other activities that require the capabilities of an operating CPU. All of the peripheral functions may be used, provided that they are enabled. The examples shown in this document use the active mode.

1.6.2 Low Power Mode 3 (LPM3)

LPM3 is the most important mode for battery-powered applications. The CPU is disabled, but enabled peripherals stay active. The basic timer provides a precise time base. When enabled, interrupts restore the CPU, switch on MCLK, and start normal operation. Table 1–2 lists the status of the MSP430 system when in LPM3.

Table 1–2. System Status During LPM3

Active	Not Active
RAM	CPU
ACLK	MCLK
32768 Hz oscillator	Disabled peripherals
LCD driver (if enabled)	Disabled interrupts
Basic timer (if enabled)	FLL
I/O ports	
8-bit timer	
Enabled peripherals	
Universal timer/port	
RESET logic	

LPM3 is activated by the following code:

```
;
; Definitions for the Operating Modes
;
GIE      .EQU   008h   ; General Interrupt enable in SR
CPUOFF   .EQU   010h   ; CPU off bit in SR
OSCOFF   .EQU   020h   ; Oscillator off bit in SR
SCG0     .EQU   040h   ; System Clock Generator Bit 0
SCG1     .EQU   080h   ; System Clock Generator Bit 1
HOLD     .EQU   080h   ; 1: Hold Watchdog
CNTCL    .EQU   008h   ; Watchdog Reset Bit
;
; Enter LPM3, enable interrupts. The Watchdog
; must be held if the ACLK is used for timing
;
        MOV    #05A00h+HOLD+CNTCL,&WDTCTL   ; Define WD
        BIS    #CPUOFF+GIE+SCG1+SCG0,SR     ; Enter LPM3
;
```

After the completion of the interrupt routine the software returns to the instruction that set the CPUoff bit. The normal wake-up from LPM3 comes from the basic timer, programmed to wake the CPU at regular intervals (ranging from 0.5 Hz to 64 Hz, or more often) to maintain a software timer. This software timer controls all necessary system activities.

Example 1–1. Interrupt Handling I

The MSP430 system runs normally in LPM3. The enabled interrupt of the basic timer wakes the system once every second. After one minute, measurements are made and then the system returns to LPM3.

```
;
```

```
; Interrupt handler for Basic Timer: Wake-up with 1Hz
;
BT_HAN  MOV   #05A00h+CNTCL,&WDTCTL  ; Reset watchdog
        INC.B SECCNT                 ; Counter for seconds +1
        CMP.B #60,SECCNT             ; 1 minute elapsed?
        JHS   MIN1                   ; Yes, do necessary tasks
        RETI                         ; No return to LPM3
;
; One minute elapsed: Return is removed from stack, a branch to
; the necessary tasks is made. There it is decided how to proceed
;
MIN1    INC   MINCNT                 ; Minute counter +1
        CLR   SECCNT                 ; 0 -> SECCNT
        ADD   #4,SP                  ; House keeping: SR, PC off Stack
        BR    #TASK                    ; Do tasks
        ...
;
TASK ...                             ; Start of necessary tasks
;
; All measurements and calculations are made: Return to LPM3
;
        MOV   #05A00h+HOLD+CNTCL,&WDTCTL ; Hold WD
        BIS   #CPUOFF+GIE+SCG0+SCG1,SR  ; Enter LPM3
```

LPM3 is the lowest current consumption mode that still allows the use of a real-time clock. The basic timer can interrupt the LPM3 at relatively long time intervals (up to 2 seconds) and update the real-time clock. If the status register is not changed during the interrupt routines, the RETI instruction returns to the instruction that set the CPUoff bit (and placed the CPU in LPM3). The program counter points to the next instruction, which is not executed unless the interrupt routine resets the CPUoff bit during its run.

If the MSP430 is awakened from LPM3, two additional clock cycles are needed to load the PC with the interrupt vector address and start the interrupt handler (8 clocks compared to 6 when in the active mode).

Example 1–2. Interrupt Handling II

The MSP430 system runs normally in LPM3. The enabled interrupt of the basic timer wakes the system once every second. After one minute, measurements are made and then the system returns to LPM3. The branch to the task is made by resetting the CPUoff bit inside the interrupt routine.

```
; Interrupt handler for Basic Timer: Wake-up with 1 Hz
```

```
;
BT_HAN MOV    #05A00h+CNTCL,&WDTCTL      ; Reset watchdog
       INC.B  SECCNT                     ; Counter for seconds +1
       CMP.B  #60,SECCNT                 ; 1 minute over?
       JHS    MIN1                       ; Yes, do necessary tasks
       RETI                              ; No return to LPM3
;
; One minute elapsed: CPUoff is reset, the program continues
; after the instruction that set the CPUoff bit (label TASK)
;
MIN1   CLR    SECCNT                     ; 0 -> SECCNT
       INC    MINCNT                     ; Minute counter + 1
       BIC    #CPUOFF+SCG1+SCG0,0(SP)          ; Reset CPUoff-bit to
continue
       RETI                             ; at label TASK
;
; Background part: Return to LPM3
;
DONE   MOV    #05A00h+HOLD+CNTCL,&WDTCTL ; Hold WD
       BIS    #CPUOFF+GIE+SCG0+SCG1,SR  ; Enter LPM3
;
; Program continues here if CPUoff bit was reset inside of the
; Basic Timer Handler.
;
TASK   ...                              ; Tasks made every minute
       JMP    DONE                       ; Back to LPM3
```

Note:

The two 8-bit counters of the universal timer/port may also be used during LPM3. If a counter is incremented by an external signal (inputs CIN, CMPI, or TPIN.5) from 0FFh to 0h, then the appropriate RCxFG-flag is set. If interrupt is enabled, the CPU wakes up.

1.6.3 Low Power Mode 4 (LPM4)

Low power mode 4 (LPM4) is used if the absolute lowest supply current is necessary or if no timing is needed or desired (no change of the RAM content is allowed). This is normally the case for storage preceding or following the calibration process. Table 3 lists the status of the MSP430 system when in LPM4.

Table 1–3. System During LPM4

Active	Not Active
RAM	CPU
I/O ports	MCLK
Enabled interrupts	ACLK
Universal timer/port (external clock)	FLL
RESET logic	Disabled peripherals
	Disabled interrupts
	Watchdog
	Timers

Once the MSP430 is waked from LPM4, the software has to decide if it is necessary to either enter LPM4 again (if the wake-up was caused by EMI, for example), or to enter one of the other operating modes. To ensure the correct decision is made, a code can be placed on a port that can be checked by the MSP430 software. Then, the active mode is entered only if this code is present.

The start-up frequency of the DCO is approximately 500 kHz and may last up to 4 seconds until a stable MCLK frequency is reached. To enter the LPM4 the following code is necessary:

```
; Enter LPM4, enable GIE
;
        BIS    #CPUOFF+OSCOFF+GIE+SCG1+SCG0,SR
```

The exit from LPM4 is principally the same as described for LPM3. Interrupt handler software has to determine if the CPU stays active or if a return to a low-power mode is necessary.

When entering the LPM4 the information in control registers SCFI0 and SCFI1 of the system clock frequency integrator (SCFI) remains stored. If at this time the ambient temperature is high, SCFI1 contains a relatively high value to compensate the negative temperature coefficient of the DCO. If the LPM4 is later exited and the ambient temperature is very low, it is possible that the resulting DCO frequency, based on the value in SCFI1, will be outside of the oscillator range. It is therefore a good programming practice to set the SCFI control register to a low value before entering LPM4.

```
Enter LPM4, enable GIE

    CLRC                                ; Ensure that new MSB is 0
    RRC    &SCFI1                       ; Use halved tap number
    BIS    #CPUOFF+OSCOFF+GIE+SCG1+SCG0,SR  ; Enter LPM4
```

Note:

The two 8-bit counters of the universal timer/port may also be used during LPM4. If a counter is incremented by an external signal (inputs CIN, CMP, or TPIN.5) from 0FFh to 0h, then the appropriate RCxFG-flag is set. If interrupt is enabled, the CPU wakes up.

Analog-to-Digital Converters

Architecture and Function of the MSP430 14-Bit ADC

Lutz Bierl

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

List of Tables

Architecture and Function of the MSP430 14-Bit ADC

Lutz Bierl

ABSTRACT

This application report describes the architecture and function of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. The principles of the ADC are explained and software examples are given. The report also explains the function of all hardware registers in the ADC. The *References* section at the end of the report lists related application reports in the MSP430 14-bit ADC series.

1 Introduction

The analog-to-digital converter (ADC) of the MSP430 family can work in two modes: the 12-bit mode or the 14-bit mode. Hardware registers allow easy adaptation to different ADC tasks. The following paragraphs describe the modes and hardware registers.

> **NOTE:** The *MSP430 Family Architecture Guide and Module Library* data book[1] is recommended. The hardware-related information given there is very valuable and complements the information given in this application report.

> **NOTE:** For related application reports in the MSP430 14-bit ADC series, see the *References* section.

Figure 1 shows the block diagram of the MSP430 14-bit ADC.
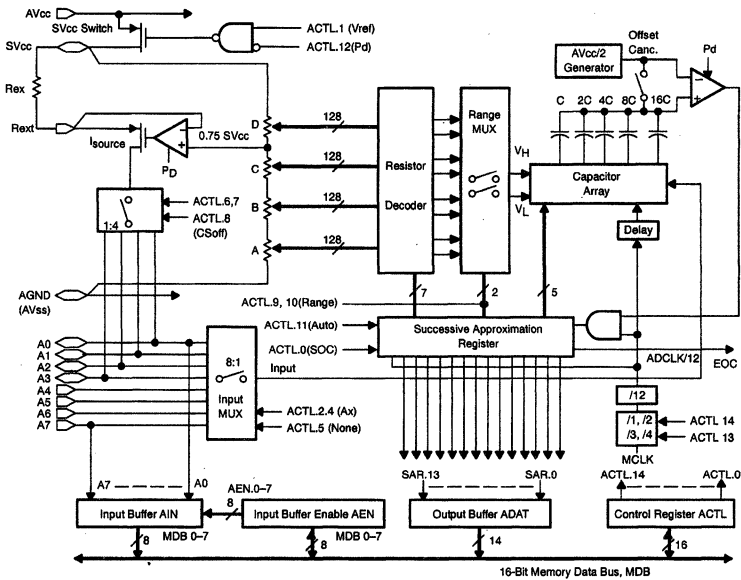
Figure 1. Hardware of the 14-Bit ADC

1.1 Characteristics of the 14-Bit ADC

- Monotonic over the complete ADC range
- Eight analog inputs; may be switched individually to digital input mode
- Programmable current source on four analog inputs. Independent of the selected conversion input: current source output and ADC input pins may be different
- Relative (ratiometric) or absolute measurement possible
- Sample and hold function with defined sampling time
- End-of-conversion flag usable with interrupt or polling
- Last conversion result is stored until start of next conversion
- Low power consumption and possibility to power down the peripheral
- Interrupt mode without CPU processing possible
- Programmable 12-bit or 14-bit resolution
- Four programmable ranges (one quarter of SVcc each)
- Fast conversion time
- Four clock adaptations possible (MCLK, MCLK/2, MCLK/3, MCLK/4)
- Internal and external reference supply possible
- Large supply voltage range

2   ADC Function and Modes

The MSP430 14-bit ADC has two range modes and two measurement modes.

The two range modes are:

*   *14-bit mode:* The ADC converts the input range from AVss to SVcc. The ADC automatically searches for one of the four ADC ranges (A, B, C, or D) that is appropriate for the input voltage to be measured.
*   *12-bit mode:* The ADC uses only one of the four ranges (A, B, C, or D). The range is fixed by software. Each range covers a quarter of the voltage at the SVcc terminal. This conversion mode is used if the voltage range of the input signal is known.

The two measurement modes are:

*   *Ratiometric mode:* A value is measured as a ratio to other values, independent of the actual SVcc voltage.
*   *Absolute mode:* A value is measured as an absolute value.

Figure 2 shows different methods to connect analog signals to the MSP430 ADC. The methods shown are valid for the 12-bit and 14-bit conversion modes:

1. Current supply for resistive sensors   Rsens1 at analog input A0
2. Voltage supply for resistive sensors   Rsens2 at analog input A1
3. Direct connection of input signals   Vin at analog input A2
4. Four-wire circuitry with current supply   Rsens3 at output A3 and inputs A4 and A5
5. Reference diode with voltage supply   Dr1 at analog input A6
6. Reference diode with current supply   Dr2 at analog input A7
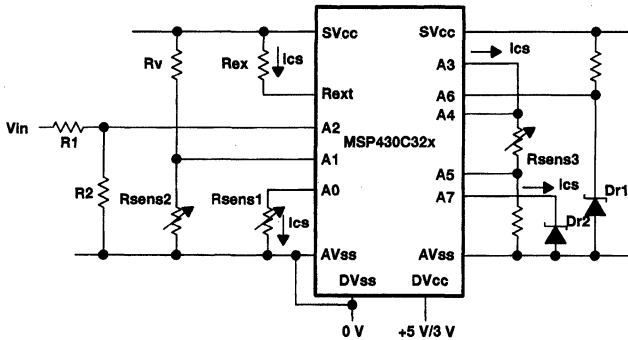
Figure 2. Possible Connections to the Analog-to-Digital Converter

The calculation formulas for all connection methods shown in Figure 2 are explained in the application report, *Application Basics for the MSP430 14-Bit ADC* (SLAA046). [3]

2.1 Function of the ADC

See Figures 1, 9, and 12 for this explanation. The full range of the ADC is made by 4×128 equal resistors connected between the SVcc pin and the AVss (AGND) pin. Setting the conversion-start (SOC) bit in the ACTL control register activates the ADC clock for a new conversion to begin.

The normal ADC sequence starts with the definition of the next conversion; this is done by setting the bits in the ACTL control register with a single instruction. The power-down (PD) bit is set to zero; the SOC bit is not changed by this instruction. After a minimum 6-μs delay to allow the ADC hardware to settle, the SOC bit may be set. The ADC clock starts after the SOC bit is set, and a new conversion starts.

- If the 12-bit mode is selected (RNGAUTO = 0) then a 12-bit conversion starts in a fixed range (A, B, C or D) selected by the bits ACTL.9 to ACTL.10.
- If the 14-bit mode is selected (RNGAUTO = 1), a sample is taken from the selected input Ax that is used only for the range decision. The found range is fixed afterwards – it delivers the two MSBs of the result – and the conversion continues like the 12-bit conversion. This first decision is made by the block range MUX.

This first step fixes the range and therefore the 2 MSBs. Each range contains a block of 128 resistors.

To obtain the 12 LSBs, a sample is taken from the selected input Ax and is used for the conversion. The 12-bit conversion consists of two steps:

- The seven MSBs are found by a successive approximation using the block resistor decode. The sampled input voltage is compared to the voltages generated by the fixed 2^7 (128) equally weighted resistors connected in series. The resistor whose leg voltages are closest to the sampled input voltage-which means between the two leg voltages—is connected to the capacitor array (see Figure 1).
- The five LSBs are found by a successive approximation process using the block capacitor array. The voltage across the selected resistor (the sampled voltage lies between the voltages at the two legs of the resistor) is divided into 2^5 (32) steps and compared to the sampled voltage.

After these three sequences, a 14-bit respective 12-bit result is available in the register ADAT.

Figure 3 shows where the result bits of an analog-to-digital conversion come from:
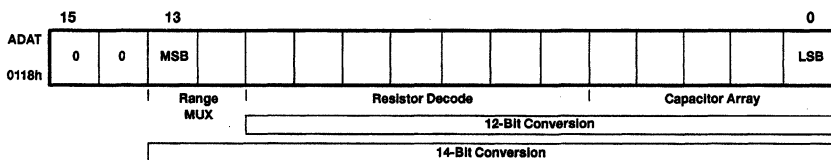
Figure 3. Sources of the Conversion Result

NOTE: The result of the 12-bit conversion does not contain range information: the result bits 12 and 13 are both zero. If these two bits are necessary for the calculation, they need to be inserted by software e.g. 2000h for range C.

2.1.1 ADC Timing Restrictions

To get the full accuracy for the ADC measurements, some timing restrictions need to be considered:

- If the ADCLK frequency is chosen too high, an accurate 14- or 12-bit conversion cannot be assured. This is due to the internal time constants of the sampling analog input and conversion network. The ADC is still functional, but the conversion results show a higher noise level (larger bandwidth of results for the same input signal) with higher conversion frequencies.

- If the ADCLK frequency is chosen too low, then an accurate 14- or 12-bit conversion cannot be assured due to charge losses within the capacitor array of the ADC. This remains true even if the input signal is constant during the sampling time.

- After the ADC module has been activated by resetting the power-down bit, at least 6 μs (power-up time in Figure 9) must elapse before a conversion is started. This is necessary to allow the internal biases to settle. This power-up time is automatically ensured for MCLK frequencies up to 2.5 MHz if the measurement is started the usual way: by separation of the definition and the start of the measurement inside of the subroutine:

```
MOV     #xxx,&ACTL   ; Define ADC measurement
CALL    #MEASR       ; Start measurement with SOC=1
...                  ; ADC result in ADAT
```

If higher MCLK frequencies are used, then a delay needs to be inserted between the definition and the start of the measurement. See the source of the MEASR subroutine in section 2.2.2. The number n of additional delay cycles (MCLK cycles) needed is:

$$n \geq (6 \, \mu s \times MCLK) - 15$$

- If the input voltage changes very fast, then the range sample and the conversion sample may be captured in different ranges. See section 2.2.1 if this cannot be tolerated. For applications like an electricity meter, this doesn't matter: the error occurs as often for the increasing voltage as for the decreasing voltage so the resulting error is zero.

- After the start of a conversion, no modification of the ACTL register is allowed until the conversion is complete. Otherwise the ADC result will be invalid.

The previously described timing errors lead to spikes in the ADC characteristic: the ADC seems to get caught at certain steps of the ADC. This is not an ADC error; the reasons are violations of the ADC timing restrictions. See Figure 4. The x-axis shows the range A from step 0 to step 4096, the y-axis shows the ADC error (steps).

Range A
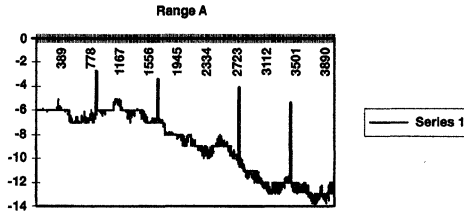
Figure 4. ADC Spikes Due to Violated Timing Restrictions

The ADC always runs at a clock rate set to one twelfth of the selected ADCLK. The frequency of the ADCLK should be chosen to meet the conversion time defined in the electrical characteristics (see data sheet). The correct frequency for the ADCLK can be selected by two bits (ADCLK) in the control register ACTL. The MCLK clock signal is then divided by a factor of 1, 2, 3, or 4. See Section 3.5.

2.1.2 Sample and Hold

The sampling of the ADC input takes 12 ADCLK cycles; this means the sampling gate is open during this time (12 μs at1 MHz). The sampling time is identical for the range decision sample and the data conversion sample.

The input circuitry of an ADC input pin, Ax, can be seen simplified as an RC low pass filter during the sampling period (12/ADCLK): 2 kΩ in series with 42 pF. The 42-pF capacitor (the sample-and-hold capacitor) must be charged during the 12 ADCLK cycles to (nearly) the final voltage value to be measured, or to within 2^{-14} of this value.
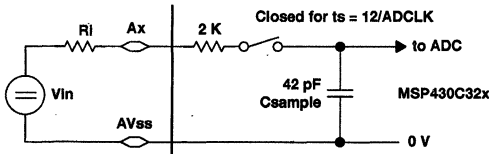
Figure 5. Simplified Input Circuitry for Signal Sampling

The sample time limits the internal resistance, Ri, of the source to be measured:

$$(Ri + 2\ k\Omega) \times 42\ pF < \frac{12}{\ln\left(2^{14}\right) \times ADCLK}$$

Solved for Ri with ADCLK = 1 MHz this results in:

$$Ri < 27.4\ k\Omega$$

This means, for the full resolution of the ADC, the internal resistance of the input signal must be lower than 27.4 kΩ.

If a resolution of n bits is sufficient, then the internal resistance of the ADC input source can be higher:

$$Ri < \frac{12}{ln(2^n) \times 42 \ pF \times ADCLK} - 2 \ k\Omega$$

For example, to get a resolution of 13 bits with ADCLK = 1 MHz, the maximum Ri of the input signal is:

$$Ri < \frac{12}{ln(2^{13}) \times 42 \ pF \times 10^6} - 2 \ k\Omega = 31.7 \ k\Omega - 2 \ k\Omega = 29.7 \ k\Omega$$

To achieve a result with 13 bit-resolution, Ri must be lower than 29.7 kΩ.

2.1.3 Absolute and Relative Measurements

The 14-bit ADC hardware allows absolute and relative modes of measurement.

2.1.3.1 Relative Measurements

As Figure 6 shows, relative measurements use resistances (sensors) that are independent of the supply voltage. This is the typical way to use the ADC. The advantage is independence from the supply voltage; it does not matter if the battery is new (Vcc = 3.6 V) or if it has reached the end of life (Vcc = 2.5 V).
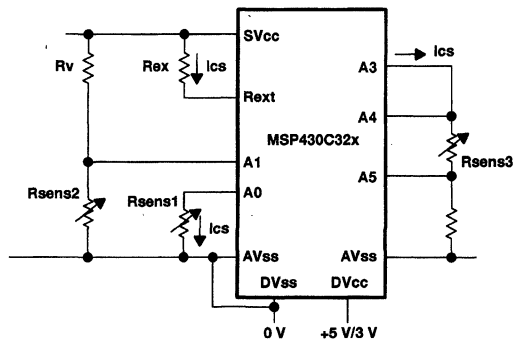
Figure 6. Relative Measurements With the MSP430C32x

2.1.3.2 Absolute Measurements

As Figure 7 shows, absolute measurements measure voltages and currents. The reference used for the conversion is the voltage applied to the SVcc terminal, regardless of whether an external reference is used or if SVcc is connected to AVcc internally. An external reference is necessary if the supply voltage AVcc (the normal reference) cannot be used for reference purposes, for example a battery supply.
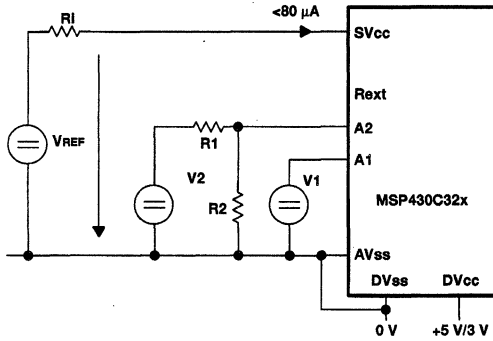
Figure 7. Absolute Measurements Using External Reference Voltage

2.2 Using the ADC in 14-Bit Mode

The 14-bit mode is used if the range of the input voltage exceeds one ADC range. The total input signal range is from analog ground (AVss) to the voltage at SVcc (external reference voltage or AVcc).
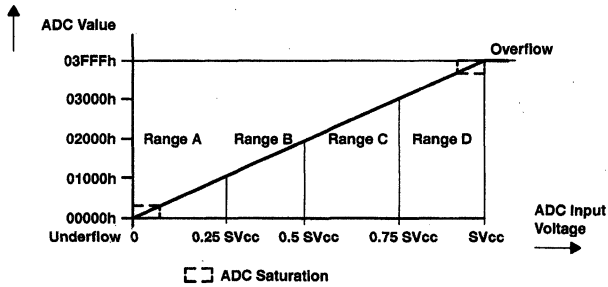
Figure 8. Complete 14-Bit ADC Range

The dashed boxes at the AVss and SVcc voltage levels indicate the saturation areas of the ADC; the measured results are 0h at AVss and 3FFFh at SVcc. The saturation areas are smaller than 10 ADC steps.

The nominal ADC formula for the 14-bit conversion is:

$$N = \frac{VAx}{VREF} \times 2^{14} \;\rightarrow\; VAx = \frac{N \times VREF}{2^{14}}$$

Where:

N = 14-bit result of the ADC conversion
VAx = Input voltage at the selected analog input Ax   [V]
$VREF$ = Voltage at pin SVcc (external reference or internal AVcc)   [V]

2.2.1 Timing

The two ADCLK bits (ACTL.13 and ACTL.14) in the ACTL control register are used to select the ADCLK frequency best suited for the ADC. The MCLK clock signal can be divided by a factor 1, 2, 3, or 4 to get the best suited ADCLK.

Using the autorange mode (RNGAUTO/ACTL.11 = 1) executes a 14-bit conversion. The selected analog input signal at input Ax is sampled twice. The range decision is made after the first sampling of the input signal; the 12-bit conversion is made after the second sampling. Both samplings are 12 ADCLK cycles in length. Altogether the 14-bit conversion takes 132 ADCLK cycles. See Figure 9 for timing details.
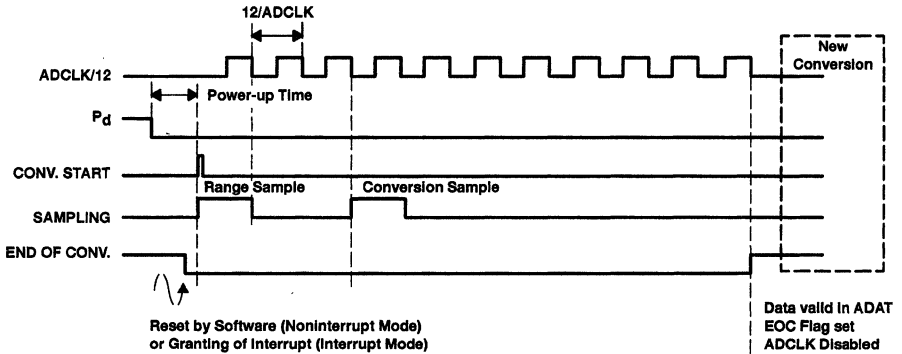
Figure 9. Timing for the 14-bit Analog-to-Digital Conversion

The input signal must be valid and steady during this sampling period to obtain an accurate conversion. It is also recommended that no activity occur during the conversion at analog inputs that are switched to the digital mode.

If the input voltage to the ADC changes during the measurement, it is possible for the range decision sample to be taken in a different ADC range than the conversion sample. The result of these conditions is saturated values:

- Increasing input voltage: nFFFh with range n = 0...2
- Decreasing input voltage: n000h with range n = 1...3

The saturated result is the best possible result under this circumstance: an analog input that changes from 2FF0h to 3020h during the sampling period delivers the saturated result 2FFFh and not 2000h.

The following software sequence can be used to check the result of an A/D conversion if the two samples (range and conversion) were taken in different ranges. If this is the case, the measurement is repeated.

```
LM  MOV    #xxx,&ACTL      ; Define measurement
    CALL   #MEASR          ; Measure ADC input
    MOV    &ADAT,R5        ; Copy ADC result
    AND    #0FFFh,R5       ; 12 LSBs stay
```

```
        JZ    LM              ; Yes, ADC value too high (n000h)
        CMP   #0FFFh,R5       ; Bits 11 to 0 all 1s?
        JEQ   LM              ; Yes, ADC value too low  (nFFFh)
        ...                   ; Both samples taken in same range
```

2.2.2 Software Example

The often-used measurement subroutine MEASR is shown below. It contains all necessary instructions for a measurement that uses polling for the completion check. The subroutine assumes a preset ACTL register; all bits except the SOC bit must be defined before the setting of the SOC bit. The subroutine may be used for 12-bit and 14-bit conversions. Up to an MCLK frequency of 2.5 MHz no additional delays are necessary to ensure the power-up time.

```
; ADC measurement subroutine.
; Call:  MOV   #xxx,&ACTL   ; Define ADC measurement. Pd=0
;        CALL  #MEASR       ; Measure with ADC
;        BIS   #PD,&ACTL    ; Power down the ADC
;  ...                      ; ADC result in ADAT
;
MEASR    BIC.B #ADIFG,&IFG2 ; Clear EOC flag
         ...                ; Insert delays here (NOPs)
         BIS   #SOC,&ACTL   ; Start measurement
M0       BIT.B #ADIFG,&IFG2 ; Conversion completed?
         JZ    M0           ; No
         RET                ; Result in ADAT
```

2.3 Using the ADC in 12-Bit Mode

The following mode is used if the range of the input voltage is known. If, for example, a temperature sensor is used whose signal range always fits into one range (for example range B), then the 12-bit mode is the right selection. The measurement time with MCLK = 1 MHz is only 96 μs compared with 132 μs if the autorange mode is used. Figure 10 shows the four ranges compared to the voltage at SVcc. The possible ways to connect sensors to the MSP430 are the same as shown for the 14-bit ADC in Figure 2.

This mode should be used only if the signal range is known and the saved 36 ADCLK cycles are a real advantage.
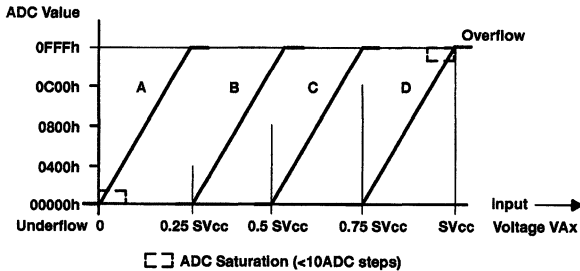
Figure 10. The Four 12-Bit ADC Ranges A to D

NOTE: The ADC results 0000h and 0FFFh mean underflow and overflow: the voltage at the measured analog input is below or above the limits of the programmed range.

All of the formulas given for the 12-bit mode assume a faultless conversion result N:

$$0 < N < 0FFFh$$

If underflow or overflow are not checked, erroneous calculation results occur.

Figure 11 shows how any of the four ADC ranges appears to the software:
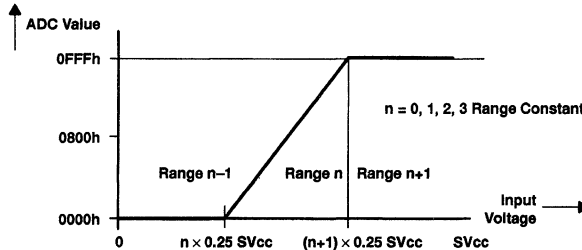
Figure 11. Single 12-Bit ADC Range

The nominal ADC formula for the 12-bit conversion is:

$$N = \frac{VAx - (n \times 0.25 \times VREF)}{VREF} \times 2^{14} \;\rightarrow\; VAx = VREF \times \left(\frac{N}{2^{14}} + n \times 0.25\right)$$

Where:
    N = 12-bit result of the ADC conversion
    VAx = Input voltage at the selected analog input Ax   [V]
    $VREF$ = Voltage at pin SVcc (external reference or internal AVcc)   [V]
    n = Range constant (n = 0, 1,2, 3 for ranges A, B, C, D)

To get the 14-bit equivalent N_{14} of a 12-bit ADC result N_{12}, the following formula may be used:

$$N_{14} = N_{12} + n \times 1000h$$

To check if the result of a 12-bit A/D conversion is correct, the following software sequence can be used:

```
MOV    #xxx,&ACTL        ; Define measurement
CALL   #MEASR            ; Measure ADC input
TST    &ADAT             ; Check if underflow (000h)
JZ     UFL               ; Underflow: go to error handling
CMP    #0FFFh,&ADAT      ; Check if overflow (0FFFh)
JEQ    OFL               ; Overflow: go to error handling
...                      ; Result is correct: use ADAT
```

2.3.1 Timing

The two ADCLK bits (ACTL.13 and ACTL.14) in the ACTL control register are used to select the ADCLK frequency best suited for the ADC. The MCLK clock signal can be divided by a factor 1, 2, 3, or 4 to get the best suited ADCLK.

Disabling the autorange mode (RNGAUTO/ACTL.11 = 0) executes a 12-bit conversion; the range defined by the ACTL.10 and ACTL.9 bits is used. The selected analog input signal at input Ax is sampled once; after the sampling, the 12-bit conversion is executed. The 12-bit conversion takes 96 ADCLK cycles. See Figure 12 for timing details.
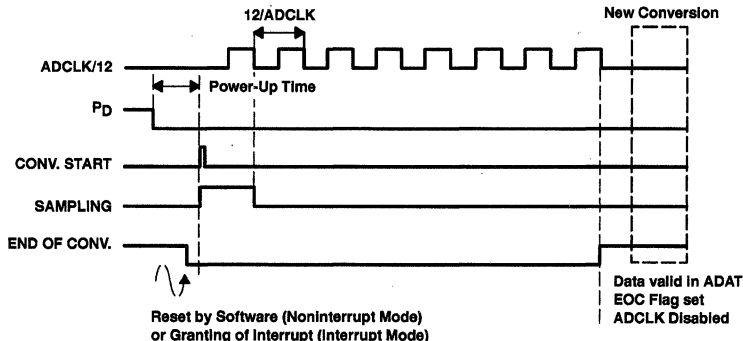
Figure 12. Timing for the 12-Bit A/D Conversion

2.3.2 Software Example

The measurement of Rsens1 is shown in Figure 2. With MCLK = 2.2 MHz, the result is always located in range B. The result must be converted to a 14-bit value to be used by software routines written for 14-bit results.

```
MOV     #ADCLK2+RNGB+CSA0+A0+VREF,&ACTL   ; Define ADC
CALL    #MEASR                           ; Measure with ADC
MOV     &ADAT,R5                         ; 12-bit ADC result to R5
ADD     #1000h,R5                        ; Add start address of range B
...                                      ; 14-bit value in R5
```

3　The A/D Controller Hardware

Paragraph 2 describes the analog-to-digital conversion. The mnemonics used are defined in the appendix.

3.1　ADC Control Registers

The ADC control registers are in the MSP430 memory area where only word addresses are possible. This means that all registers are word-structured and should be accessed by word instructions only. Byte addressing results in a nonpredictable operation.

The access description below the register bits has the following meaning:

- rw-0　read/write bit, reset after power-up clear (PUC)
- rw-1　read/write bit, set after power-up clear
- r0　read as zero
- r　read only
- (w)r0　write only. Writing generates a pulse, no reset necessary. Read as zero

3.1.1　ACTL Control Register

The ACTL control register is the main register for programming the ADC. Its content (shown in Figure 13) defines the current operation. All of the bits should be changed only after a completed conversion. Otherwise a faulty result will occur.
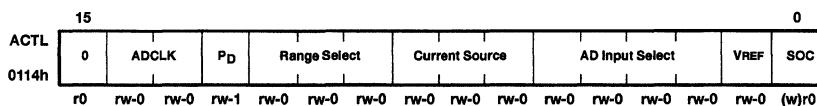
Figure 13.　ACTL Control Register

3.1.1.1　Conversion Start (SOC)

The SOC write-only bit (see Figure 14) starts an analog-to-digital conversion. The conditions of the measurement are defined with the other bits of the ACTL register. It is not necessary to reset the bit. The SOC bit is always read as a zero.
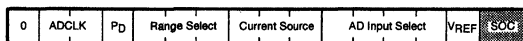
Figure 14.　Conversion Start (SOC)

EXAMPLE: start the ADC as defined by the content of the ACTL register.

```
MOV    #ADCLK2+RNGA+CSOFF+A0+VREF,&ACTL   ; Define ADC
...                                       ; Delay 6us to allow settling
BIS    #SOC,&ACTL                         ; Start ADC conversion
```

3.1.1.2　Voltage Reference Bit (VREF)

The VREF bit (see Figure 15) defines whether an internal or an external reference voltage is used for the A/D conversion.

| 0 | ADCLK | P_D | Range Select | Current Source | AD Input Select | V_REF | SOC |
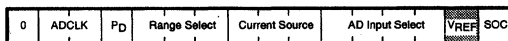
Figure 15. Voltage Reference Bit (VREF)

VREF = 0: External reference. The transistor between AVcc and SVcc is switched off. The SVcc terminal is an input pin for an external reference voltage. The external reference source must be able to supply a current up to 80 µA. The voltage range for the external reference is $AVcc/2 \leq VREF \leq AVcc$.

VREF = 1: Internal reference. The transistor between AVcc and SVcc is switched on: the SVcc output terminal is connected to the analog supply voltage AVcc. No external voltage should be supplied to the SVcc terminal.

EXAMPLE: define an A/D conversion with the internal reference voltage AVcc.

```
MOV    #ADCLK2+RNGAUTO+CSOFF+A0+VREF,&ACTL
```

Start an A/D conversion with an external reference connected to the SVcc terminal (VREF = 0).

```
MOV    #ADCLK2+RNGAUTO+CSOFF+A0,&ACTL    ; Vref = 0
CALL   #MEASR                            ; Start the measurement
```

3.1.1.3 ADC Input Select Bits

The four ADC input select bits (see Figure 16) define which of the possible eight analog inputs is selected for the A/D conversion.

| 0 | ADCLK | P_D | Range Select | Current Source | AD Input Select | V_REF | SOC |
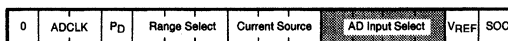
Figure 16. ADC Input Selection Bits

Table 2 lists the possible ADC input selections.

Table 1. ADC Input Selection Bits

INPUT SELECTION CODE	MNEMONIC	SELECTED ANALOG INPUT	COMMENT
0	A0	A0	Signal at the pin A0 is selected
1	A1	A1	Signal at the pin A1 is selected
2	A2	A2	Signal at the pin A2 is selected
3	A3	A3	Signal at the pin A3 is selected
4	A4	A4	Signal at the pin A4 is selected
5	A5	A5	Signal at the pin A5 is selected
6	–	A6	Not implemented with the MSP430C32x
7	–	A7	Not implemented with the MSP430C32x
8–15	–	None	No analog input is selected

EXAMPLE: to start an ADC conversion for analog input A3 with unchanged conditions:

```
BIC    #03Ch+PD,&ACTL        ; Reset all input select bits
...                          ; 6 µs delay
BIS    #A3+SOC,&ACTL         ; Start conversion for A3
```

3.1.1.4 Current Source Output Select Bits

The three current source output select bits (see Figure 17) define the analog input Ax where the output current of the current source is switched. To switch the current source off, ACTL.8 (CSOFF) is set to one.

| 0 | ADCLK | P_D | Range Select | Current Source | AD Input Select | V_REF | SOC |

Figure 17. Current Source Output Select Bits

Table 2. Current Source Output Select Bits

OUTPUT SELECTION CODE	MNEMONIC	SELECTED CURRENT OUTPUT	COMMENT
0	CSA0	A0	Current source connected to pin A0
1	CSA1	A1	Current source connected to pin A1
2	CSA2	A2	Current source connected to pin A2
3	CSA3	A3	Current source connected to pin A3
4–7	CSOFF	Off	Current source switched off

EXAMPLE: connect the current source to pin A3, and start measurement at pin A4. All other ADC conditions stay unchanged. This example refers to the hardware configuration for Rsens3 shown in Figure 2.

```
BIC   #01FCh+PD,&ACTL       ; Reset SOC and input sel. Bits
      ...                    ;6 µs delay
BIS   #CSA3+A4+SOC,&ACTL     ; Start conversion for A4
```

3.1.1.5 Range Selection Bits

The three range select bits (see Figure 18) define the ADC range that is used for the conversion.

| 0 | ADCLK | P_D | Range Select | Current Source | AD Input Select | V_REF | SOC |
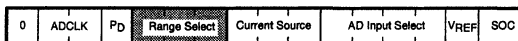
Figure 18. Range Select Bits

Table 3. Range Select Bits

RANGE SELECTION CODE	MNEMONIC	SELECTED RANGE	COMMENT
0	RNGA	A	0 to 0.25 × SVcc
1	RNGB	B	0.25 to 0.5 × SVcc
2	RNGC	C	0.5 to 0.75 × SVcc
3	RNGD	D	0.75 × SVcc to SVcc
4–7	RNGAUTO	A, B, C, D	Automatic range select

EXAMPLE: prepare the ACTL register for measurement of analog input A3 using the internal reference, and with the current source connected to A3 and fixed to range B.

```
MOV   #RNGB+CSA3+A3+VREF,&ACTL
```

3.1.1.6 Power Down Bit (Pd)

The power-down bit (see Figure 19) reduces the power consumption of the ADC to the lowest possible value. It switches off the comparator, the SVcc switch, and the current source.

| 0 | ADCLK | Pᴅ | Range Select | Current Source | AD Input Select | Vʀᴇꜰ | SOC |
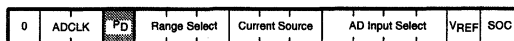
Figure 19. Power Down Bit (Pd)

Pd = 0:   The ADC is switched on.

Pd = 1:   The SVcc switch is off, the comparator is powered down and the current source is off. This ensures the minimum current consumption for the ADC.

EXAMPLE: power down the ADC for minimum current consumption.

```
BIS #PD,&ACTL ; Power down the ADC
```

3.1.1.7 Clock Frequency Selection Bits

The two clock frequency selection bits (see Figure 20) select the optimum clock frequency for the ADC. This is necessary due to the relatively low maximum ADCLK frequency (1.5 MHz) compared to the maximum MCLK frequency (3.3 MHz).

| 0 | ADCLK | Pᴅ | Range Select | Current Source | AD Input Select | Vʀᴇꜰ | SOC |
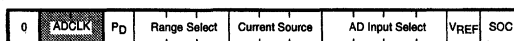
Figure 20. Clock Frequency Selection Bits

Table 4. Clock Frequency Selection Bits

SELECTION CODE	MNEMONIC	DIVISION FACTOR	ADCLK FREQUENCY	COMMENT
0	ADCLK1	1	MCLK	MCLK ≤ 1.5 MHz
1	ADCLK2	2	MCLK/2	MCLK > 1.5 MHz
2	ADCLK3	3	MCLK/3	MCLK > 3.0 MHz
3	ADCLK4	4	MCLK/4	(MCLK > 4.5 MHz)

EXAMPLE: For MCLK = 2.5 MHz, the highest possible ADCLK frequency (1.25 MHz) is set.

```
MOV #ADCLK2+RNGAUTO+A3+VREF,&ACTL
```

3.1.1.8 Bit 15

Bit 15 (see Figure 21) should always be set to zero to maintain software compatibility with future versions of the ADC.

| 0 | ADCLK | Pᴅ | Range Select | Current Source | AD Input Select | Vʀᴇꜰ | SOC |
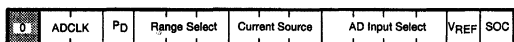
Figure 21. Bit 15

3.1.2 A/D Data Register ADAT

The ADC data register ADAT contains the result of the last A/D conversion. The conversion data is valid in the ADAT register at the end of a conversion and stays valid until another A/D conversion is started with the setting of the SOC bit (ACTL.0). The read-only structure of the ADAT register does not allow read/modify/write instructions like ADD or BIC with the ADAT register used as the destination: only the instructions BIT, TST and CMP may be used this way. With the ADAT register as a source, all instructions may be used.

Figure 22 shows the result of a 12-bit conversion: the value is always between 000h (underflow) and FFFh (overflow), independent of the ADC range used. The missing range information (bits 12 and 13) must be added by the software.

15				11											0	
ADAT	0	0	0	0	MSB										LSB	ACTL.11=0
0118h																
	r0	r0	r0	r0	r	r	r	r	r	r	r	r	r	r	r	r
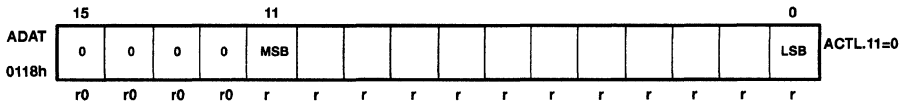
Figure 22. The Data Register ADAT, 12-Bit A/D Conversion

Figure 23 shows the result of a 14-bit conversion: the value is between 0000h (underflow) and 3FFFh (overflow). Result bits 13 and 12 indicate the range of the result:

- 00   Range A   0 to $0.25 \times SVcc$
- 01   Range B $0.25 \times SVcc$ to $0.50 \times SVcc$
- 10   Range C $0.50 \times SVcc$ to $0.75 \times SVcc$
- 11   Range D $0.75 \times SVcc$ to $SVcc$

15		13													0	
ADAT	0	0	MSB												LSB	ACTL.13=1
0118h																
	r0	r0	r	r	r	r	r	r	r	r	r	r	r	r	r	r
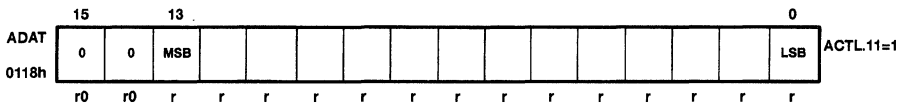
Figure 23. Data Register ADAT, 14-Bit A/D Conversion

To read the result of the last conversion, use a simple MOV instruction:

```
MOV &ADAT,R5      ; Copy the ADC result to R5
...               ; A new conversion may begin
```

3.1.3 Input Register AIN

Input register AIN (see Figure 24) is a read-only word register; however, only the low byte of the register is implemented. The same access restrictions are valid as described for the ADAT register. AIN.0 to AIN.7 correspond to the input terminals A0 to A7. The high byte of the register is read as 00h. Input register AIN shows the digital input information at the input terminals that are switched to the digital mode (AEN.x = 1). The formula for the bit AIN.x is:

$$AIN.x = Ax \text{ .and. } AEN.x$$

Where:
    $AIN.x$ = Bit x of the input register AIN
    Ax = Logic level at the analog input Ax
    $AEN.x$ = Bit x of the input enable register AEN

This means, that analog inputs (AEN.x = 0) are read as zero.

15							7								0
AIN	0	0	0	0	0	0	0	0	AIN.7	AIN.6	AIN.5	AIN.4	AIN.3	AIN.2	AIN.1	AIN.0
0110h																
	r0	r0	r0	r0	r0	r0	r0	r0	r	r	r	r	r	r	r	r
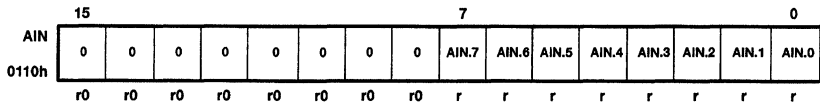
Figure 24. Input Register AIN

EXAMPLE: The A5 input terminal is used as a digital input. Test if this input is high; if yes, jump to label A5HI:

```
INITA5  BIS   #20h,&AEN     ; Use pin A5 as digital input
        . . .
        BIT   #020h,&AIN    ; Pin A5 high?
        JNZ   A5HI          ; Yes, goto A5HI
        . . .               ; No, A5 is low
```

NOTE: Only digital inputs with very low activity or controlled access (e.g. keyboard scan) should be connected to inputs A0 to A7. Otherwise, this activity influences the measurement results of the analog inputs.

3.1.4 Input Enable Register AEN

Input enable register AEN (see Figure 25) is a read/write word. However, only the low byte of the register is implemented. AEN.0 to AEN.7 correspond to input terminals A0 to A7. The high byte of the register is read as 00h. The initial state of all bits is reset.

	15							7								0
AEN 0112h	0	0	0	0	0	0	0	0	AEN.7	AEN.6	AEN.5	AEN.4	AEN.3	AEN.2	AEN.1	AEN.0
	r0	r0	r0	r0	r0	r0	r0	r0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0
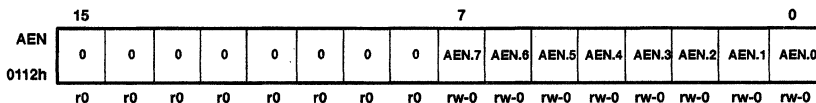
Figure 25. Input Enable Register AEN

Input enable register bits AEN.x control the function of input pins A0 to A7:

AEN.x = 0: Input terminal Ax is used as an analog input. Bit AIN.x is read as zero.

AEN.x = 1: Digital input. The bit read in the AIN register represents the logical level at the appropriate Ax terminal.

EXAMPLE: The A5 and A4 input terminals are used as digital inputs. An application is given with the AIN terminal example.

```
        BIS   #030h,&AEN         ; Pin A5 and A4 digital inputs
```

3.2 Current Source

A stable, programmable current source is available at the four analog inputs A0 to A3. With programming resistor Rex between terminals SVcc and Rext, it is possible to get a defined current, Ics, out of the programmed analog input Ax. Ics is directly related to the voltage at SVcc. This allows relative measurements to be made using the current source that are independent from the ADC supply voltage SVcc. The analog input to be measured and the analog input used for the current source are independent of each other; this means that the current source may be programmed to input A3 and the measurement taken from inputs A4 and A5, as shown in Figure 6 for Rsens3.

3.2.1 Normal Use of the Current Source

Figure 26 shows the normal use of the current source: the generated current Ics flows through the addressed analog input A0 and generates a voltage drop Vin at the connected sensor Rsens. This voltage drop Vin is multiplexed to the ADC and measured.

The current Ics defined by the external resistor Rex is:

$$Ics = \frac{0.25 \times Vref}{Rex}$$

The input voltage Vin at the selected analog input with the current Ics and a connected sensor Rsens is:

$$Vin = Rsens \times Ics = Rsens \times \frac{0.25 \times V_{REF}}{Rex} \;\rightarrow\; Rsens = \frac{Vin}{Ics}$$

The ADC result N for an input voltage Vin is:

$$N = 2^{14} \times \frac{Vin}{V_{REF}} \rightarrow Vin = \frac{V_{REF} \times N}{2^{14}}$$

The above equations lead to the measured sensor resistance Rsens:

$$Rsens = \frac{Rex}{0.25 \times V_{REF}} \times Vin = \frac{Rex}{0.25 \times V_{REF}} \times \frac{V_{REF}}{2^{14}} \times N = Rex \times N \times 2^{-12}$$

The result N of the A/D conversion is:

$$N = \frac{0.25 \times V_{REF}}{Rex} \times \frac{2^{14}}{V_{REF}} \times Rsens = \frac{Rsens}{Rex} \times 2^{12}$$

Where:

Rex = Resistor between pins SVcc and Rex (defines current Ics)   [Ω]
Rsens= Resistor to be measured (connected between Ax and AGND)   [Ω]
VREF = Voltage at SVcc. External (VREF = 0) or internal (VREF = 1)   [V]
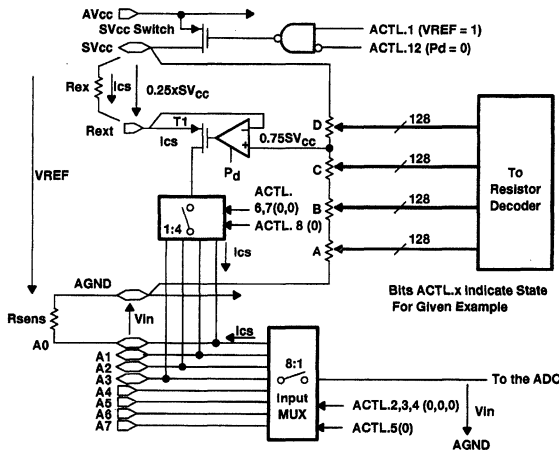
Figure 26. The Current Source

If the 12-bit conversion is used, the above equations change to:

$$N = \frac{\dfrac{0.25 \times V_{REF}}{Rex} \times Rsens - n \times 0.25 \times V_{REF}}{V_{REF}} \times 2^{14} = \left(\frac{Rsens}{Rex} - n\right) \times 2^{12}$$

This gives, for the unknown resistor Rsens:

$$Rsens = Rex \times \left(\frac{N}{2^{12}} + n\right)$$

The code sequence for the measurement shown in Figure 26 is:

```
MOV     #ADCLK1+RNGA+CSA0+A0+VREF,&ACTL   ; Define ADC

CALL    #MEASR                           ; Measure Rsens at A0

...                                      ; Result in ADAT
```

When using the current source, it is not possible to use the full range of the ADC: only the range defined with *Load Compliance* in the *Electrical Description* is valid ($0.5 \times SVcc$, which means only the ranges A and B). Figures 28 and 29 show the typical error characteristics of the current source at its limit. Figure 28 shows the error characteristic for Vcc = 4.5 V and a relatively high Rex (1 kΩ). It shows that up to a ratio of 0.745 for VA0/SVcc (which means range A, B, and nearly all of range C) the current source works correctly. Then ΔIcs (the difference between the programmed Ics and the real Ics) increases linearly with

$$\frac{\Delta V}{\Delta I} = Rex$$

The reason is saturated transistor T1 of the current source. When T1 is saturated, only the external resistor Rex determines the current Ics. Figure 27 shows the measurement circuitry and an explanation of the error curves. The small dashed box indicates the area that is magnified in Figures 28 and 29.
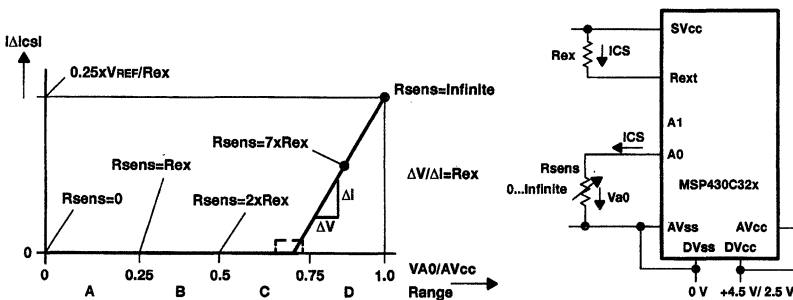
Figure 27. Measurement Circuitry for the Error of the Current Source
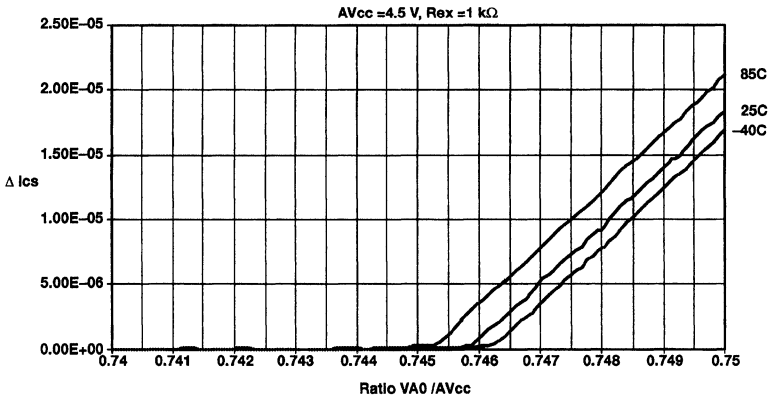
Figure 28. Error of the Current Source at the Limit

Figure 29 gives the characteristic at the other extreme: Vcc = 2.5 V and Rex = 150 Ω. The slope beyond the operation limit of the current source (here at VA0/AVcc = 0.7125) is also:

$$\frac{\Delta V}{\Delta I} = Rex$$
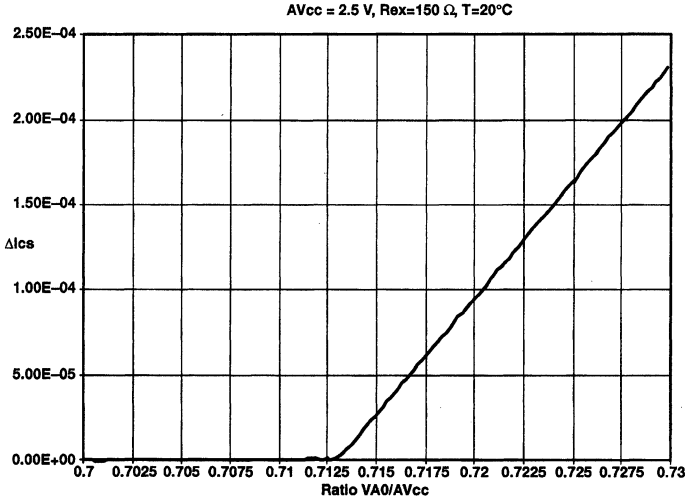
Figure 29. Error of the Current Source at the Limit

The characteristic shown in Figure 29 indicates that the current source works up to 71% of the applied AVcc under worst case conditions; this includes ADC ranges A, B and 84% of range C. If Rex is chosen as 1 kΩ and SVcc is 4.5 V, then the current source works up to a ratio of 0.745, which means it covers nearly 98% of range C.

If the current source is used with an external amplifier (operational amplifier) that amplifies the output signal coming from the current source, then the full range of the ADC can be used with a different ADC input. Figure 30 shows such a circuit. The signal at analog input A0 can use the full range of the A/D converter; the signal at A1 is restricted to the working area of Ics that is shown in Figures 28 and 29.

The equations for the circuitry are explained in *Application Basics for the MSP430 14-Bit ADC Application Report* (SLAA046).[3]

Figure 30. Application of the Current Source With the Full ADC Range at Input A0

3.2.2 Current Source Used for Level Shifting

If analog signals that lie partially or totally outside of the ADC range of the MSP430 (AVss to SVcc), need to be measured then the current source can be used to shift the signal level into the measurable range.

The current transformer, shown on the left in Figure 31, outputs a secondary voltage that is proportional to the primary current, Iac. The signed output voltage (symmetrical to the AVss voltage) is shifted into the middle of the ADC range by a current Ics through the resistor Rsh. This current Ics must be small, due to the sensitivity of current transformers to dc biasing.

Figure 31. Current Measurement With Level Shifting

The right side of Figure 31 shows the measurement of a signed dc current. Due to the two directions of the accumulator current (charge and discharge current) level shifting is necessary: the charge current generates a positive voltage, Vsh; the discharge current generates a negative voltage, at the shunt. The current, Ics, together with resistor Rc, also shifts the voltage drop of the discharge current into the ADC range.

The advantages of level shifting by the current source are:

• Possibility to measure signals that are outside of the ADC range

• Omission of the saturation area near the AVss voltage

• Possible readjustment of the zero current ADC value during periods with no current flow

3.3 SVcc Terminal

The SVcc terminal is the reference for all ADC measurements. The voltage applied to this terminal refers to the result value 2^{14} (16,384), regardless of whether the reference voltage is applied internally or externally (external VREF). The VREF bit located in the ACTL registers defines whether the internal reference AVcc is used (VREF = 1) or an external voltage is used (VREF = 0).

3.3.1 SVcc Terminal Used as an Output for the ADC Reference Voltage

Typically, the SVcc terminal is used to supply the reference and voltage to the ADC circuitry. It can be activated while measurements are being taken and deactivated for low power periods. Figure 32 shows an example of this. All of the sensors connected to the MSP430 are powered by the SVcc terminal.

The SVcc terminal outputs the AVcc voltage if the following conditions are true:

$VSVcc = VREF$ .and. @ Pd .and. VAVcc
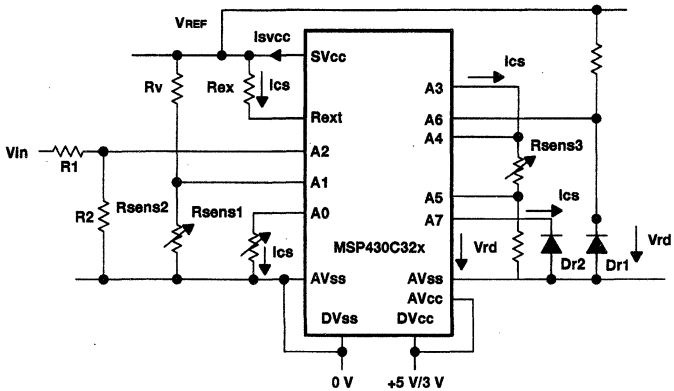
Figure 32. SVcc Terminal Used as an Output

The voltage, Vin, at analog input A2 is measured in comparison to the voltage at SVcc. If the voltage, VREF, at SVcc is known (AVcc is stable and known, ISVcc is small), then Vin can be measured exactly. Otherwise an external reference diode (or equivalent) may be connected to a free analog input, and its voltage, Vrd, is measured. See Figure 32. The formula for Vin is then:

$$Vin = Vrd \times \frac{Nin}{Nrd} \times \frac{R1 + R2}{R2}$$

Where:

    Vrd = Voltage of the reference diode   [V]
    Nin = 14-bit result for Vin
    Nrd = 14-bit result for the voltage Vrd of the reference diode

3.3.2 SVcc Terminal Used as an Input for the ADC Reference Voltage

For absolute voltage measurements an external reference voltage, VREF, is necessary (see Figure 33). The sensor measurements for Rsens1 to Rsens3 are made the same way as with the internal reference voltage. The only difference is the VREF bit of the ACTL register: it is set to zero to allow an external reference voltage to be used. The formula for Vin is:

$$Vin = VREF \times \frac{N}{2^{14}} \times \frac{R1 + R2}{R2}$$
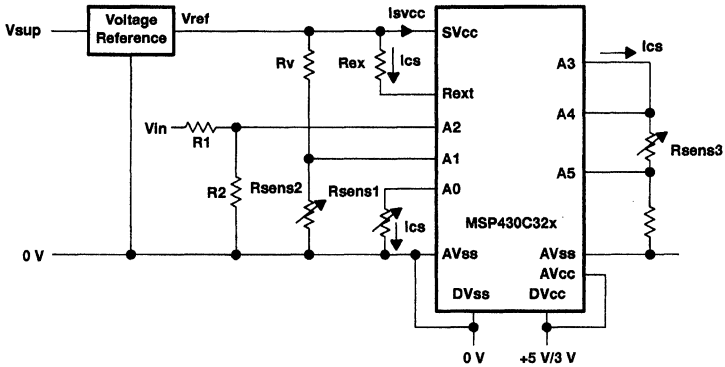
Figure 33. SVcc Terminal Used as an Input for a Reference Voltage

NOTE: If an external voltage reference is used, then it must be able to deliver not only the current for the external circuitry but also a maximum current of 80 µA at 5 V to supply the parts of the ADC that are connected to SVcc.

The maximum voltage at SVcc when used as an input is the voltage applied to AVcc.

Measurements with the ADC using external reference voltages down to 1.2 V at SVcc showed that the ADC does not change its characteristic. However, the noise of the result doubles when compared to a 5-V supply. This is due to the voltage-independent noise generated by the ADC.

3.3.3 Connection of Current Consuming Loads to SVcc

If the current drawn by the external ADC circuitry exceeds 8 mA, then an external switch for the external analog voltage should be considered. A simple PNP transistor can be used for this purpose as shown in Figure 34. The SVcc terminal is used as an input pin for the external reference voltage (ADC control bit VREF = 0). This method allows the full accuracy of the ADC also with current consuming loads. Output TP.0 switches the power to the current consuming loads off and on.

The schematic in Figure 34 is simplified for clarity. The connection principle shown in *Application Basics for the MSP430 14-Bit ADC Application Report* (SLAA046)[3] needs to be applied, especially with the larger currents flowing here.
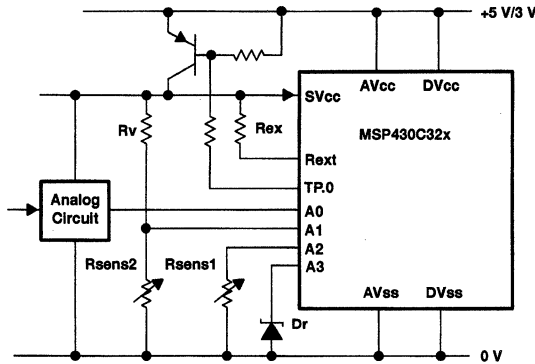
Current Consuming Analog Parts (I>> 8 mA)

Figure 34. Connection of Current Consuming Loads to SVcc

The software for switching the PNP transistor follows. The TP-port handling may be included in the MEASR subroutine if this is an advantage. The example refers to the hardware shown in Figure 34. Rsens1 is measured.

```
BIC.B    #TP0,&TPD              ; TP.0 pin is low if enabled
BIS.B    #TP0,&TPE              ; Enable TP0: switch PNP on
MOV      #ADCLK+RNGA+CSA2+A2,&ACTL ; ADC: ext. reference
CALL     #MEASR                 ; Measure Rsens1 at A2
BIC.B    #TP0,&TPE              ; Switch PNP off: TP0 Hi-Z
...                             ; Result in ADAT
```

3.4 Interrupt Handling

All of the ADC software examples shown previously use polling techniques to check for conversion completion. This takes up computing power that can be used more effectively if interrupt techniques are used.

3.4.1 Interrupt Flags

ADC interrupt flags are not located in the ACTL control register. This allows advanced interrupt handling. Several interrupt enable flags in a common byte can be disabled and enabled together with minimal effort, something that is impossible with flags located in the individual control words. The two flags controlling the interrupt of the ADC are:

```
IE2     .EQU   01h   ; Interrupt Enable Register 2
ADIE    .EQU   04h   ; ADC interrupt enable bit (IE2.2)
;
IFG2    .EQU   03h   ; INTERRUPT FLAG REGISTER 2
ADIFG   .EQU   04h   ; ADC "EOC" Bit (IFG2.2)
```

3.4.2 Interrupt Handlers

The interrupt structure of the ADC allows the conversion time to be used for other calculations or procesor tasks. Two ADC interrupt handler examples follow:

EXAMPLE: analog input A0 (without current source) and A1 (with the current source enabled) are measured alternately. The measured 14-bit results are stored in address MEAS0 for input A0 and MEAS1 for input A1. The time interval between the two measurements is defined by the 8-bit timer: each timer interrupt starts a new conversion for the previously prepared analog input. Other timers may also be used for the generation of the time interval.

```
; Analog input      A0         A1
; Current Source    OFF        ON
; Result to         MEAS0      MEAS1
; Range selection   AUTO       AUTO
; Reference         SV_cc      SV_cc
;
; Initialization part for the ADC:
;
       MOV    #RNGAUTO+CSOFF+A0+VREF,&ACTL
       BIS.B  #ADIE,&IE2    ; Enable ADC interrupt
       MOV.B  #0FFh-3,&AEN  ; Only A0 and A1 analog inputs
       ...                  ; Initialize other modules
;
; ADC interrupt handler: A0 and A1 are measured alternately.
; The next measurement is prepared but not started.
; The interrupt flag ADIFG is reset automatically
;
ADC_INT BIT   #A1,&ACTL     ; A1 result in ADAT?
        JNZ   ADI           ; Yes
        MOV   &ADAT,MEAS0    ; A0 value is actual
        MOV   #RNGAUTO+CSON+A1+VREF,&ACTL    ; A1 next meas.
        RETI
ADI     MOV   &ADAT,MEAS1                     ; A1 value is actual
        MOV   #RNGAUTO+CSOFF+A0+VREF,&ACTL  ; A0 next meas.
        RETI
;
; 8-bit timer interrupt handler: the ADC conversion is started
; for the previously prepared ADC input
;
T8BINT  BIS   #SOC,&ACTL            ; Start conversion for the ADC
        ...                         ; Execute other timer tasks
        RETI
;
        .SECT  "INT_VEC0",0FFEAh  ; Interrupt vectors
        .WORD  ADC_INT            ; ADC interrupt vector
        .SECT  "INT_VEC1",0FFF8h
        .WORD  T8BINT             ; 8-bit timer interrupt vector
```

The software for the 12-bit conversion is similar to that for the 14-bit conversion, the only difference being the replacement of the RNGAUTO bit during the initialization of the ACTL control register. Instead, the desired range (RNGA, RNGB, RNGC, or RNGD) is included in the initialization part of each measurement.

NOTE: An independent timer—like that used in the example above—is recommended; do not use the ADC interrupt handler to restart the ADC. If the ADC interrupt handler starts the next conversion, then any interrupt failure leads to a flip-flop effect; the missing ADC interrupt does not start a new conversion, and the ADC activity ceases.

EXAMPLE: for best results the CPU is switched off during the ADC measurement. The measurement subroutine starts the conversion and switches off the CPU afterwards. The interrupt routine called by the conversion completion resets the CPUoff bit (SR.4) of the stored status register SR and allows the CPU to continue with the measured ADC result. The 12-bit result is moved to R5.

```
CPUoff    .equ   010h        ; SR: CPU off bit
GIE       .equ   008h        ; SR: General Intrpt enable
RNGB      .equ   0200h       ; ACTL: Select Range B
;
          ...
          BIC.B  #ADIFG,&IFG2 ; Reset ADC flag
          BIS.B  #ADIE,&IE2   ; ADC Intrpt Enable
          EINT                ; Enable GIE interrupt
          MOV    #RNGB+CSOFF+A1+VREF,&ACTL  ; Define ADC
          CALL   #MEASURE     ; Measure with ADC
          MOV    &ADAT,R5     ; Result to R5
          ...                 ; Process result in R5
;
; Subroutine: CPU is switched off to get minimum noise
;
MEASURE   BIS    #SOC,&ACTL      ; Start ADC conversion
          BIS    #CPUoff,SR   ; Switch CPU off, MCLK active
          NOP                 ; Wait for completion of ADC
          RET                 ;
;
; Interrupt Handler for the Analog-to-Digital Converter
; The CPUoff bit of the saved SR is cleared to allow the
; software to continue after the RETI
;
ADC_INT   BIC    #CPUoff,0(SP) ; Allow SW run (CPUoff = 0)
          RETI
;
; Interrupt Vectors
;
          .sect  "INT_VEC1",0FFEAh
          .WORD  ADC_INT      ; ADC Vector
```

3.5 ADC Clock Generation

The frequency of the ADC clock, ADCLK, must be in a certain range as discussed in section 2.1.1 *ADC Timing Restrictions*. To allow the adaptation of the ADCLK to the full range of the MCLK frequency, four possibilities of prescaling are provided:

- MCLK   if MCLK < 1.5 MHz
- MCLK/2   if MCLK > 1.5 MHz
- MCLK/3   if MCLK > 3.0 MHz

This allows an MCLK/ADCLK combination to be selected for nearly all applications that fits the calculation needs, while providing the necessary A/D conversion speed.

4 ADC Characteristics

The next four figures show typical measured ADC characteristics: the absolute error (ADC steps) is dependent on the input value (ADC steps from 5 to 16380). Error characteristics like these are used with *Additive Improvement of the MSP430 14-Bit ADC Characteristic* (SLAA047)[4], *Linear Improvement of the MSP430 14-Bit ADC Characteristic* (SLAA048)[5], and *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic* (SLAA050)[6] to illustrate the improvements possible by methods using different hardware and software.
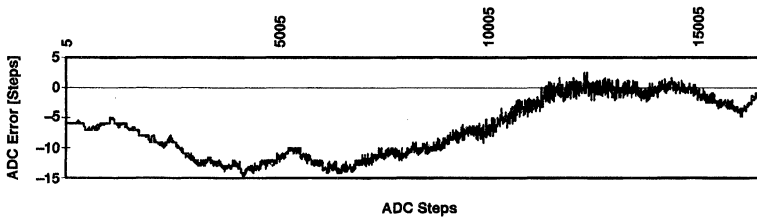
Figure 35. Error Characteristic Device 1
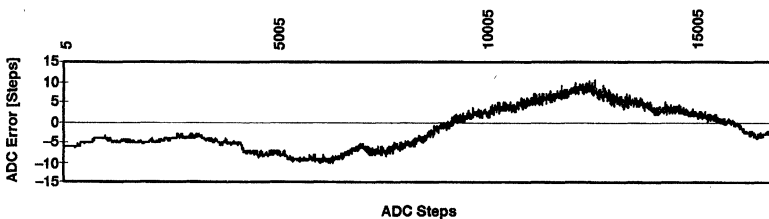
Figure 36. Error Characteristic Device 2
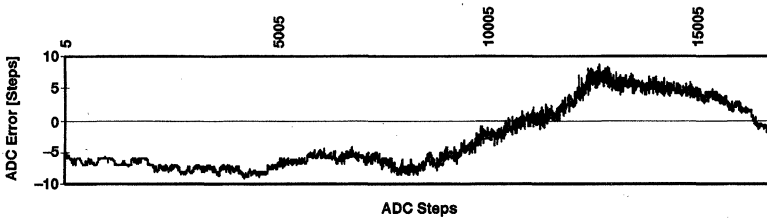
Figure 37. Error Characteristic Device 3
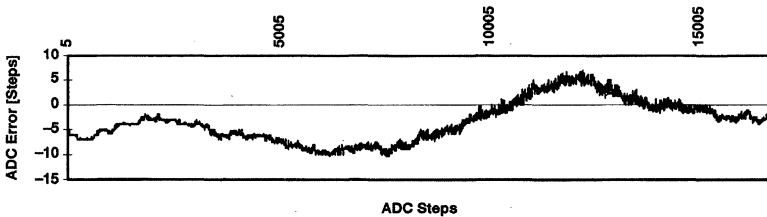
Figure 38. Error Characteristic Device 4

5 Summary

This application report complements *Application Basics for the MSP430 14-Bit ADC* (SLAA046)[3] that contains applications of the 14-bit ADC. *Additive Improvement of the MSP430 14-Bit ADC Characteristic* (SLAA047)[4] explains different methods to minimize the ADC error, and the limitations of the ADC.

All five of the application reports in the MSP430 14-bit ADC series include system applications (hardware and proven software) using all parts and modes of the ADC.

6 References

1. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B
2. Data Sheet, MSP430x32x Mixed Signal Microcontroller, 1998, Literature #SLAS164
3. *Application Basics for the MSP430 14-Bit ADC application report*, 1999, Literature #SLAA046
4. *Additive Improvement of the MSP430 14-Bit ADC Characteristic application report*, 1999, Literature #SLAA047
5. *Linear Improvement of the MSP430 14-Bit ADC Characteristic application report*, 1999, Literature #SLAA048
6. *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA050
7. *MSP430 Metering Application Report*, 1998, Literature #SLAAE10C

Appendix A   Definitions Used With the Application Examples

```
; HARDWARE DEFINITIONS
;
AIN        .equ   0110h ; Input register (for digital inputs)
AEN        .equ   0112h ; 0: analog input   1: digital input
;
ACTL       .equ   0114h ; ADC control register: control bits
SOC        .equ   01h   ; Conversion start
VREF       .equ   02h   ; 0: ext. reference    1: SVcc on
A0         .equ   00h   ; Input A0
A1         .equ   04h   ; Input A1
A2         .equ   08h   ; Input A2
A3         .equ   0Ch   ; Input A3
A4         .equ   10h   ; Input A4
A5         .equ   14h   ; Input A5
CSA0       .equ   00h   ; Current Source to A0
CSA1       .equ   40h   ; Current Source to A1
CSA2       .equ   80h   ; Current Source to A2
CSA3       .equ   0C0h  ; Current Source to A3
CSOFF      .equ   100h  ; Current Source off
CSON       .equ   000h  ; Current Source on
RNGA       .equ   000h  ; Range select A (0 ... 0.25×SVcc)
RNGB       .equ   200h  ; Range select B (0.25..0.50×SVcc)
RNGC       .equ   400h  ; Range select C (0.5...0.75×SVcc)
RNGD       .equ   600h  ; Range select D (0.75..SVcc)
RNGAUTO    .equ   800h  ; 1: range selected automatically
PD         .equ   1000h ; 1: ADC powered down
ADCLK1     .equ   0000h ; ADCLK = MCLK
ADCLK2     .equ   2000h ; ADCLK = MCLK/2
ADCLK3     .equ   4000h ; ADCLK = MCLK/3
ADCLK4     .equ   6000h ; ADCLK = MCLK/4
;
ADAT       .equ   0118h ; ADC data register (12 or 14-bits)
;
IFG2       .equ   03h   ; Interrupt flag register 2
ADIFG      .equ   04h   ; ADC "EOC" bit (IFG2.2)
;
IE2        .equ   01h   ; Interrupt enable register 2
ADIE       .equ   04h   ; ADC interrupt enable bit (IE2.2)
;
TPD        .equ   04Eh  ; TP-port: address data register
TPE        .equ   04Fh  ; TP-port: address of enable register
TP0        .equ   1     ; Bit address of TP.0
TP1        .equ   2     ; Bit address of TP.1
```

Application Basics for the MSP430 14-Bit ADC

Lutz Bierl

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

List of Tables

Application Basics for the MSP430 14-Bit ADC

Lutz Bierl

ABSTRACT

This application report gives a detailed overview of several applications for the 14-bit analog-to-digital converter (ADC) of the MSP430 family. Proven software examples and basic circuitry are shown and explained. The 12-bit mode is also considered, when possible. The *References* section at the end of the report lists related application reports in the MSP430 14-bit ADC series.

1 Introduction

The application report *Architecture and Function of the MSP430 14-Bit ADC*[1] explained the architecture and function of the MSP430 14-bit analog-to-digital converter (ADC). The hardware (registers, current source, used reference, interrupt handling, clock generation) was explained in detail and typical ADC characteristics were shown.

Figure 1 shows the block diagram of the MSP430 14-bit ADC.
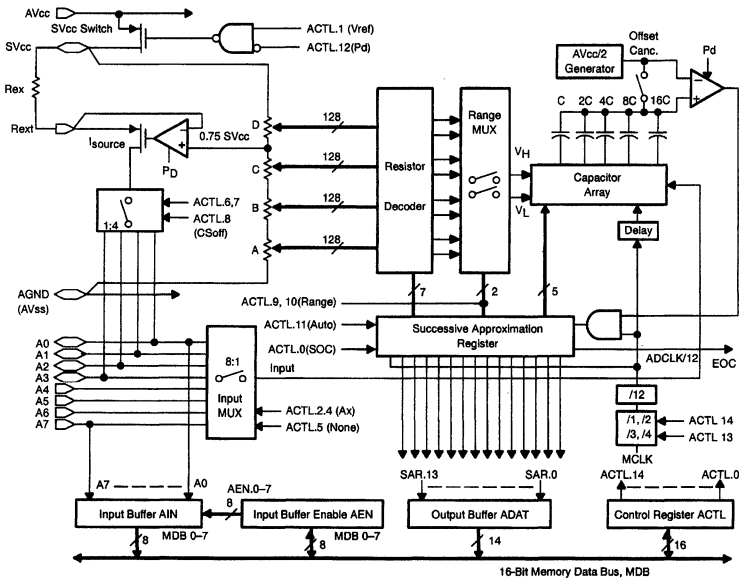
Figure 1. MSP430 14-Bit ADC Hardware

2 Applications

This application report shows several methods for connecting resistive sensors, bridge assemblies, and analog signals to the ADC. Solutions are given for the 12-bit and 14-bit conversions, with and without using the integrated current source. The equations shown result in voltages and resistances. To calculate the sensor values (pressure, current, temperature, light intensity a.s.o) normally with non-linear equations, refer to the following sections of Chapter 5:

- Table Processing (Section 5.2)
- Temperature Calculations for Sensors (Section 5.5.6)
  - Table Processing for Sensor Calculations
  - Algorithms for Sensor Calculations
  - Coefficient Calculations for the Equations
- The Floating Point Package (Section 5.6)

2.1 Connection of Analog Signals and Sensors

Figure 2 shows possible methods for connecting analog signals to the ADC. The methods shown are valid for the 12-bit and 14-bit conversion modes:

1. Current supply for resistive sensors   Rsens1 at analog input A0
2. Voltage supply for resistive sensors   Rsens2 at analog input A1
3. Direct connection of input signals   Vin at analog input A7
4. Four-wire circuitry with current supply   Rsens3 at output A3 and inputs A4 and A5
5. Reference diode with voltage supply   Dr1 at analog input A6
6. Reference diode with current supply   Dr2 at analog input A2

The resistance of the wiring, Rwire, in the following equations may be neglected if it is low compared to the sensor resistance.
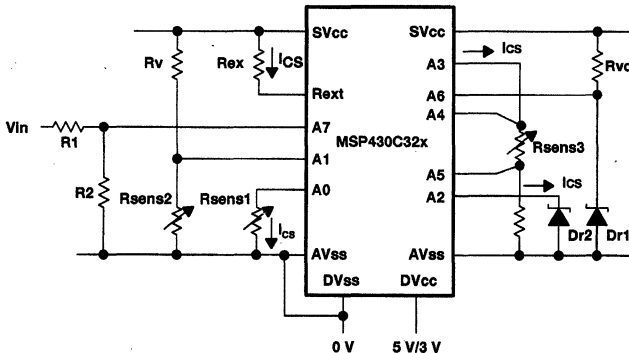
Figure 2. Possible Connections to the ADC

2.1.1 Current Supply for Sensors

The ADC formula for the resistor Rx in figure 3 (Rsens1 in Figure 2) which is fed from the current source is (14-bit conversion):

$$N = \frac{V_{A0}}{V_{REF}} \times 2^{14} = \frac{Ics \times (Rx + 2 \times Rwire)}{V_{REF}} \times 2^{14}$$

$$N = \frac{\frac{0.25 \times V_{REF}}{Rex} \times (Rx + 2 \times Rwire)}{Vref} \times 2^{14} = \frac{Rx + 2 \times Rwire}{Rex} \times 2^{12}$$

This leads to:

$$Rx = Rex \times \frac{N}{2^{12}} - 2 \times Rwire$$

For the 12-bit conversion the formula is:

$$N = \frac{V_{A0} - n \times 0.25 \times V_{REF}}{V_{REF}} \times 2^{14} = \left(\frac{Rx + 2 \times Rwire}{Rex} - n\right) \times 2^{12}$$

This leads to:

$$Rx = Rex \times \left(\frac{N}{2^{12}} + n\right) - 2 \times Rwire$$

Where:
	N	ADC conversion result for resistor Rx	
	Rx	Sensor resistance	[Ω]
	Rex	Current source resistance (defines Ics)	[Ω]
	Rwire	Wiring resistance (one direction only)	[Ω]
	V_{REF}	Voltage at terminal SVcc (internal or external reference)	[V]
	V_{A0}	Voltage at the analog input A0	[V]
	n	Range number (0,1,2,3 for ranges A,B,C,D)	
	Ics	Current generated by the current source	[A]
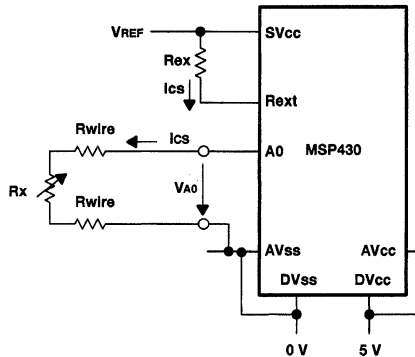
Figure 3. Current Supply for the Sensor Rx

If the resistance of the wires may be neglected (Rx >> Rwire) then the above formulas simplify to (14-bit conversion):

$$N = \frac{Rx}{Rex} \times 2^{12} \qquad\qquad Rx = Rex \times \frac{N}{2^{12}}$$

For the 12-bit conversion the formulas become:

$$N = \left(\frac{Rx}{Rex} - n\right) \times 2^{12} \qquad\qquad Rx = Rex \times \left(\frac{N}{2^{12}} + n\right)$$

2.1.2 Voltage Supply for Sensors

The ADC formula for the resistor Rx in figure 4 (Rsens2 in Figure 2) which is connected to Vref through the series resistor Rv is (14-bit conversion):

$$N = \frac{V_{A1}}{V_{REF}} \times 2^{14} = \frac{Rx + 2 \times Rwire}{Rv + Rx + 2 \times Rwire} \times 2^{14} \rightarrow Rx = Rv \times \frac{N}{2^{14} - N} - 2 \times Rwire$$

For the 12-bit conversion the formula is:

$$N = \left(\frac{V_{A1}}{V_{REF}} - n \times 0.25\right) \times 2^{14} = \left(\frac{Rx + 2 \times Rwire}{Rv + Rx + 2 \times Rwire} - n \times 0.25\right) \times 2^{14}$$

This leads to:

$$Rx = Rv \times \frac{1}{\dfrac{2^{14}}{N + n \times 2^{12}} - 1} - 2 \times Rwire$$

Where: Rv   Resistance of the series resistor $[\Omega]$
        V_{A1} Voltage at the analog input A1   [V]
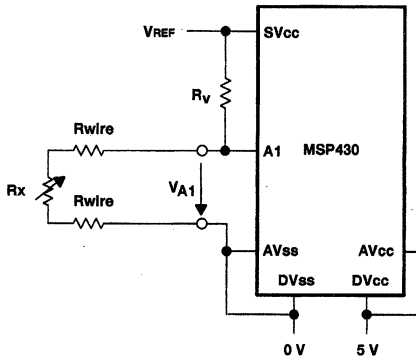
Figure 4. Voltage Supply for the Sensor Rx

If the resistance of the wires can be neglected (Rx >> Rwire), the above formulas simplify for the 14-bit conversion to:

$$N = \frac{Rx}{Rv + Rx} \times 2^{14} \rightarrow Rx = Rv \times \frac{N}{2^{14} - N}$$

For the 12-bit conversion the formula becomes:

$$N = \left(\frac{Rx}{Rv + Rx} - n \times 0.25\right) \times 2^{14} \rightarrow Rx = Rv \times \frac{1}{\dfrac{2^{14}}{N + n \times 2^{12}} - 1}$$

2.1.3 Four-Wire Sensors Circuit

Four-wire circuits eliminate errors due to the voltage drop caused by the connection lines (Rwire) to the sensor. Instead of two lines, four are used—two for the measurement current, and two for the sensor voltages. The two sensor lines do not carry current; the current at the analog inputs is in the nanoamp range, so no voltage drop falsifies the measured values. The four-wire circuit is used with a heat volume counter shown in the Section 4.5, *Heat Volume Counter.*

Figure 5 shows the four-wire circuit with its current supply.
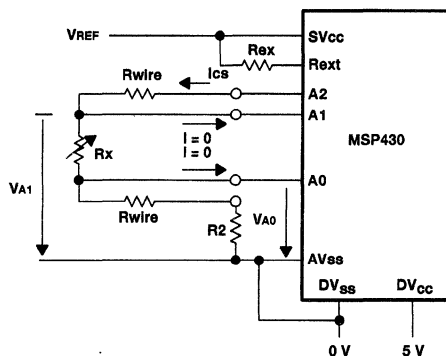
Figure 5. Four-Wire Circuit With Current Supply

The difference ΔN of the two measurement results for the analog inputs A1 and A0 is:

$$\Delta N = (V_{A1} - V_{A0}) \times \frac{2^{14}}{V_{REF}} = Ics \times ((Rx + Rwire + R2) - (Rwire + R2)) \times \frac{2^{14}}{V_{REF}}$$

$$\Delta N = \frac{0.25 \times V_{REF}}{Rex} \times Rx \times \frac{2^{14}}{V_{REF}} = \frac{Rx}{Rex} \times 2^{12}$$

This gives for Rx:

$$Rx = Rex \times \frac{\Delta N}{2^{12}}$$

Where: ΔN Difference of the two ADC results (here NA1 – NA0)

As the two final equations for ΔN and Rx show, the influence of the Rwire resistances disappears completely.

NOTE:

The two formulas above are valid for 14-bit and 12-bit conversions. If the 12-bit ADC results are measured in different ADC ranges, then the 12-bit results need a correction (the missing two MSBs—13th and 14th bits—of the ADC results must be added):

Range A: 0   Range B: 1000h   Range C: 2000h   Range D: not possible

Resistor R2 is necessary, because the ADC cannot measure down to AVss (0 V) due to saturation effects. R2 may be quite small; it is only necessary to get above the saturation voltage—normally less than 30 ADC steps.

The software to measure ∆N is shown next. The hardware of Figure 5 is used:

```
; Measure upper leg of Rx at input A1 and store ADC value.
; The Current Source is connected to A2
;
    MOV     #RNGAUTO+CSA2+A1+VREF,&ACTL   ; Define ADC
    CALL    #MEASR                        ; Upper leg voltage of Rx (A1)
    MOV     &ADAT,R5                       ; Store A1 value in R5
;
; Measure lower leg of Rx at input A0. Current Src to A2
;
    MOV     #RNGAUTO+CSA2+A0+VREF,&ACTL   ; Define ADC
    CALL    #MEASR                        ; Lower leg voltage of Rx (A0)
;
; The difference delta N of the 2 measurements is proportional
; to the value Rx: Rx = Rext x deltaN x 2^-12
;
    SUB     &ADAT,R5                       ; R5 contains delta N
    ...                                    ; Calculate Rx
```

2.1.4 Connection of Bridge Assemblies

Bridge assembly sensors are best known for pressure measurement. The voltage difference (Vp – Vm) between the two bridge legs changes with the pressure to be measured. For clarity, the temperature measurement circuitry that is normally necessary is not included.
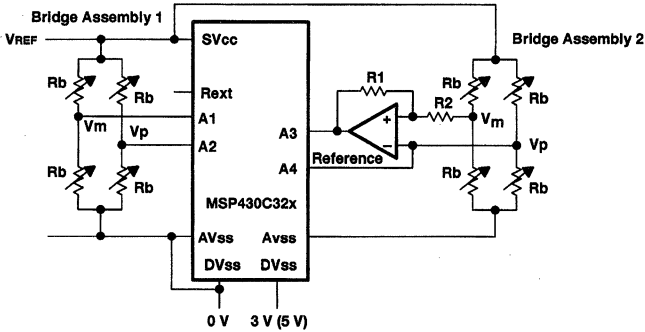
Figure 6. Connection of Bridge Assemblies

On the left side of Figure 6, a bridge assembly creates a voltage difference large enough to be measured by the ADC with appropriate resolution. The measurement result is the difference of the two ADC results measured at the A1 and A2 analog inputs.

$$\Delta N = \frac{V_{A2} - V_{A1}}{V_{REF}} \times 2^{14} \rightarrow \Delta V = \Delta N \times V_{REF} \times 2^{-14} = V_{REF} \times \frac{\Delta Rb}{Rb}$$

Where: $\quad \Delta N \quad$ Difference of the two ADC results (here N$_{A2}$–N$_{A1}$)

$\qquad\quad$ $VAx \quad$ Voltage at the ADC input Ax measured to AVss $\qquad$ [V]

$\qquad\quad$ $\Delta V \quad$ Difference of the two bridge leg voltages (here V$_{A2}$–V$_{A1}$) $\quad$ [V]

$\qquad\quad$ $\Delta Rb \quad$ Change of a single bridge resistance due to measured value

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [Ω]

$\qquad\quad$ $Rb \quad$ Nominal value of a single bridge resistor $\qquad\qquad\qquad$ [Ω]

With the above equations, the interesting bridge output value $\Delta Rb/Rb$ becomes:

$$\frac{\Delta Rb}{Rb} = \Delta N \times 2^{-14}$$

If the difference of the two measurements is too small to be used, an operational amplifier may be used as shown on the right of Figure 6. Here the possibility to measure the reference voltage (one of the two bridge legs) is shown too: analog input A4 measures the reference that can be used for a better result together with the input A3.

The voltage difference ΔV between the analog inputs A3 and A4 is:

$$\Delta V = V_{A3} - V_{A4} = (v \times (Vp - Vm) + Vp) - Vp = v \times (Vp - Vm)$$

$$\Delta V = \frac{R1}{R2} \times (Vp - Vm) = \frac{R1}{R2} \times \frac{V_{REF}}{2 \times Rb}((Rb + \Delta Rb) - (Rb - \Delta Rb)) = \frac{R1}{R2} \times V_{REF} \times \frac{\Delta Rb}{Rb}$$

The same voltage difference ΔV described with the ADC equation is:

$$\Delta V = V_{A3} - V_{A4} = \left(\frac{N_{A3}}{2^{14}} - \frac{N_{A4}}{2^{14}}\right) \times V_{REF} = (N_{A3} - N_{A4}) \times \frac{V_{REF}}{2^{14}}$$

Combining the two equations above delivers the interesting two equations:

$$\Delta N = v \times \frac{\Delta Rb}{Rb} \times 2^{14} = \frac{R1}{R2} \times \frac{\Delta Rb}{Rb} \times 2^{14}$$

For the bridge output value $\Delta Rb/Rb$, the following equation is used: the value $\Delta Rb/Rb$ is necessary for the final calculation of the measured item, e.g., pressure $p = f(\Delta Rb/Rb)$:

$$\frac{\Delta Rb}{Rb} = \frac{\Delta N}{v \times 2^{14}} = \frac{R2}{R1} \times \frac{N_{A3} - N_{A4}}{2^{14}}$$

Where: $\quad \Delta N \quad$ Difference of the two ADC results (here NA3–NA4)

$\qquad\quad$ $\Delta V \quad$ Voltage difference of analog inputs A3 And A4

$\qquad\qquad\qquad$ (VA3–VA4) $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [V]

$\qquad\quad$ $v \quad$ Amplification of the operational amplifier: v=R1/R2

$\qquad\quad$ $Vp \quad$ Voltage of the bridge leg connected to the

$\qquad\qquad\qquad$ noninverting input $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [V]

$\qquad\quad$ $Vm \quad$ Voltage of the bridge leg connected to the

$\qquad\qquad\qquad$ inverting input $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [V]

If the reference input (analog input A4 in Figure 6) is not implemented, then the difference of two measurements at the amplifier output (analog input A3 in Figure 6) is used. The voltage difference ΔV between two measurements is:

$$\Delta V = V_{A31} - V_{A30} = (v + 0.5) \times Vref \times \frac{\Delta Rb1 - \Delta Rb0}{Rb}$$

The same voltage difference ΔV described with the ADC equation is:

$$\Delta V = V_{A31} - V_{A30} = \left(\frac{N_{A31}}{2^{14}} - \frac{N_{A30}}{2^{14}}\right) \times V_{REF} = (N_{A31} - N_{A30}) \times \frac{V_{REF}}{2^{14}}$$

The two equations above deliver the equation for ΔN, e.g., the ADC value representing the difference of two weights:

$$\Delta N = N_{A31} - N_{A30} = (v + 0.5) \times \frac{\Delta Rb1 - \Delta Rb0}{Rb} \times 2^{14} = \left(\frac{R1}{R2} + 0.5\right) \times \frac{(\Delta Rb1 - \Delta Rb0)}{Rb} \times 2^{14}$$

And for the difference of the two bridge output values that represent for example a weight difference. The value $\Delta Rb/Rb$ is used for the final calculation of the measured item, e.g., the weight $G = f(\Delta Rb/Rb)$:

$$\frac{\Delta Rb}{Rb} = \frac{\Delta Rb1 - \Delta Rb0}{Rb} = \frac{N_{A31} - N_{A30}}{(v + 0.5) \times 2^{14}} = \frac{N_{A31} - N_{A30}}{\left(\frac{R1}{R2} + 0.5\right) \times 2^{14}}$$

Where:

ΔN	Difference of the two ADC results (here N_{A31}–N_{A30})	
N_{A30}	ADC result of the 1st measurement, e.g., the zero point of the bridge	
N_{A31}	ADC result of the 2nd measurement, e.g., a weight measurement	
ΔV	Voltage difference of two analog measurements (VA31–VA30)	[V]
V_{A30}	Voltage at the analog input A3, e.g., for the zero point of bridge	[V]
V_{A31}	Voltage at the analog input A3, e.g., a weight measurement	[V]
v	Amplification of the operational amplifier: v=R1/R2	
$\Delta Rb0$	Resistor deviation (Rb0–Rb) of the 1st measurement	[Ω]
$\Delta Rb1$	Resistor deviation (Rb1–Rb) of the 2nd measurement	[Ω]
ΔRb	Resistor difference (Rb1–Rb0)	
Rb	Nominal value of a single bridge resistance	[Ω]

2.1.5 Reference Measurements

The simplest way to get a reference voltage is to use the supply voltage of the MSP430. If this is not possible, and a stable reference voltage is needed, e.g. for voltage measurements, then a reference diode can be used. Figure 7 shows two ways to connect a reference diode to the MSP430:

- The reference diode Dr1 is fed via the series resistor Rvd
- The reference diode Dr2 is fed by the current source of the MSP430
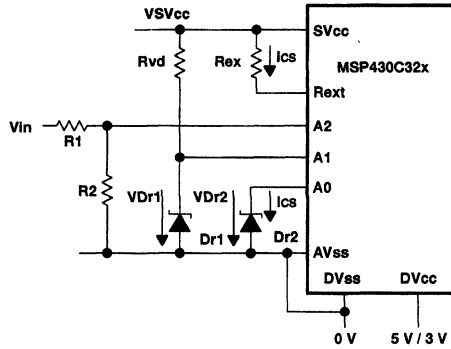
Figure 7. Connecting Reference Elements

If the external voltage Vin shown in Figure 7 is to be measured, then the following equations may be used. For reference purposes the voltage VDr is used, not the *unknown* supply voltage Vsvcc:

$$Vin = \frac{R1 + R2}{R2} \times \frac{Nin}{2^{14}} \times Vsvcc$$

The unknown voltage Vsvcc is fixed by the measurement of the reference voltage VDr:

$$V_{Dr} = \frac{N_{Dr}}{2^{14}} \times Vsvcc \rightarrow Vsvcc = \frac{2^{14}}{N_{Dr}} \times V_{Dr}$$

This leads to:

$$Vin = \frac{R1 + R2}{R2} \times \frac{Nin}{N_{Dr}} \times V_{Dr}$$

Where:
Vin	Input voltage to be measured	[V]
Vsvcc	Supply voltage at terminal SVcc	[V]
V$_{Dr}$	Voltage of the reference diode Dr	[V]
Nin	ADC measurement result for the input voltage Vin	
N$_{DR}$	ADC measurement result for the reference voltage V$_{Dr}$	
R1, R2	Voltage divider for input voltage Vin	[Ω]

If the supply voltage Vsvcc is overlaid by hum (mains driven supply), then the referencing method shown above gives much better results if the reference diode Dr is measured twice—once before the input voltage Vin (N$_{Dr0}$), and once afterwards (N$_{Dr1}$). The two ADC results, N$_{Dr0}$ and N$_{Dr1}$, are used as follows:

$$Vin = \frac{R1 + R2}{R2} \times \frac{2 \times Nin}{N_{Dr0} + N_{Dr1}} \times V_{Dr}$$

The calculation above uses the mean value of the measured values of the voltage Vsvcc (linear correction).

2.2 14-Bit Analog-to-Digital Conversion With Signed Signals

The MSP430 ADC measures unsigned signals from Vref, the voltage applied to the terminal SVcc (internal or external), to AVss. If signed measurements are necessary then a virtual zero point has to be provided. Signals above this zero point are treated as positive signals; signals below it are treated as negative ones. Four possibilities for a virtual zero point are shown in this chapter:

- Virtual ground IC: The zero point is provided by a special IC
- Split power supply: The zero point is provided by two power supplies
- Current source: The zero point is provided by the current source and a drop resistor
- Resistor divider: The zero point is provided by a resistor divider

The signal source is connected to the virtual zero point with its reference potential (first two solutions) or to the AVss potential (last two solutions).

2.2.1 Virtual Ground IC

With the *phase splitter* TLE2426, a common zero point is provided which lies exactly in the middle of the voltage between the Vref and the AVss potential. The reference voltage Vref may be internal (AVcc) or external. All signed input voltages are connected to this virtual ground with their reference potential. The virtual ground voltage (at analog input A0 in Figure 8) is measured after regular time intervals, and the measured ADC value is stored and subtracted from the measured analog input signal V1 (here at input A1). This results in a signed, offset corrected ADC value for the signal at the analog input A1. The virtual ground method is used with some electronic electricity meters shown in Section 4.1, *Electricity Meters.*

NOTE: The V1 range is –1.5 V to 1.5 V for V_{ref} = 3.0 V
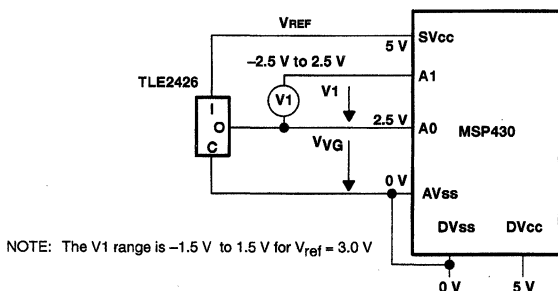
Figure 8. Virtual Ground IC for Signed Voltage Measurement

The formula for the difference of the ADC results ΔN is:

$$\Delta N = (N_{A1} - N_{A0}) = \frac{V1 + Vvg}{V_{REF}} \times 2^{14} - \frac{Vvg}{V_{REF}} \times 2^{14} = \frac{V1}{V_{REF}} \times 2^{14}$$

This leads to the formula for V1:

$$V1 = V_{REF} \times \frac{\Delta N}{2^{14}}$$

Where: *V1*   Voltage to be measured   [V]
        ΔN Difference of the two ADC results (here $N_{A1} - N_{A0}$)
        VREF   Voltage at the SVcc terminal measured against AVss
                   terminal   [V]
        Vvg   Voltage at the A0 terminal (0.5 × Vref)   [V]

EXAMPLE: The virtual ground voltage at A0 is measured and stored in location VIRTGR (register or RAM). The value of VIRTGR is subtracted from the ADC value measured at input A1; this gives the signed, offset corrected value for the input signal at the A1 input. The measurement subroutine MEASR shown in Section 4.1 is used.

```
VIRTGR .EQU  R6            ; Virtual Ground ADC value
;
; Measure virtual ground voltage at input A0 and store value
; for reference. MCLK = 3MHz: divide MCLK by 2
;
        MOV    #ADCLK2+RNGAUTO+CSOFF+A0+VREF,&ACTL
        CALL   #MEASR       ; Measure A0 (virtual ground)
        MOV    &ADAT,VIRTGR ; Store result: 14-bit value
        ...;
; Measure analog input signal V1 (0 ...03FFFh) and compute
; a signed, offset corrected value for V1 (0E000h ...01FFFh)
;
        MOV    #ADCLK2+RNGAUTO+CSOFF+A1+VREF,&ACTL
        CALL   #MEASR       ; Measure A1 (input voltage V1)
        MOV    &ADAT,R5     ; Read ADC value for V1
        SUB    VIRTGR,R5    ; R5 contains signed delta N
        ...                 ; V1 = Vref x deltaN x 2^-14
```

2.2.2 Split Power Supply

With two power supplies, for example with 2.5 V and −2.5 V, a potential in the middle of the MSP430 ADC range can be created. Figure 9 shows this arrangement. All signed input voltages are connected to this voltage with their reference potential (0 V). The mid range voltage (at analog input A0) is measured after regular time intervals and the measured ADC value is stored and subtracted from the measured signal (here at analog input A1). This gives a signed, offset corrected result for the analog input A1. The split power supply method is used with some of the electronic electricity meters shown in Section 4.1, *Electricity Meters*.
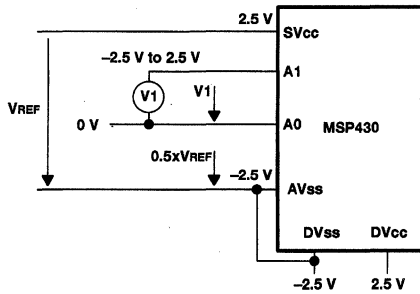
Figure 9. Split Power Supply for Signed Voltage Measurement

The formula for the difference of the ADC results ΔN is:

$$\Delta N = (N_{A1} - N_{A0}) = \frac{V1 + (0.5 \times V_{REF})}{V_{REF}} \times 2^{14} - \frac{(0.5 \times V_{REF})}{V_{REF}} \times 2^{14} = \frac{V1}{V_{REF}} \times 2^{14}$$

This leads to the formula for V1:

$$V1 = V_{REF} \times \frac{\Delta N}{2^{14}}$$

Where: $V1$ Input voltage to be measured   [V]

        ΔN Difference of the two ADC results (here $N_{A1} - N_{A0}$)

        V_{REF} Voltage between the SVcc and the AVss terminals   [V]

The same software example can be used as shown before with the virtual ground IC.

2.2.3 Use of the Current Source

With the current source method shown in Figure 10, a voltage that is partially or completely below the AVss potential can be shifted into the middle of the used ADC range of the MSP430. This is accomplished by a drop resistor Rh whose voltage drop shifts the input voltage accordingly. This method is especially useful if differential measurements are necessary, because the ADC value of the signal's midpoint (zero point) is not available as easily as with the two methods shown previously. If absolute measurements are necessary, then a calibration or a measurement with a known input voltage equal to the zero point is needed.
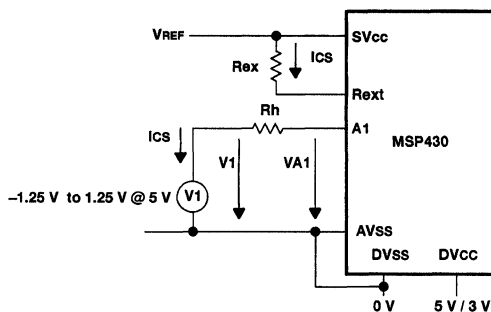
NOTE: The V1 range is –0.75 V...+0.75 V for Vref = 3 V

Figure 10. Current Source Used for Level Shifting

The example of Figure 11 shows an input signal V1 ranging from –1.25 V to 1.25 V. To shift the signal's zero voltage (0 V) to the midpoint voltage Vzv of the usable ADC range (this range is approximately 0.5 × Vref, so Vzv is 0.25 × Vref) a current Ics is used. The necessary current Ics to shift the input signal is:

$$Ics = \frac{Vzv}{Rh} \rightarrow Rh = \frac{Vzv}{Ics} = \frac{Vzv}{\dfrac{0.25 \times V_{REF}}{Rex}}$$
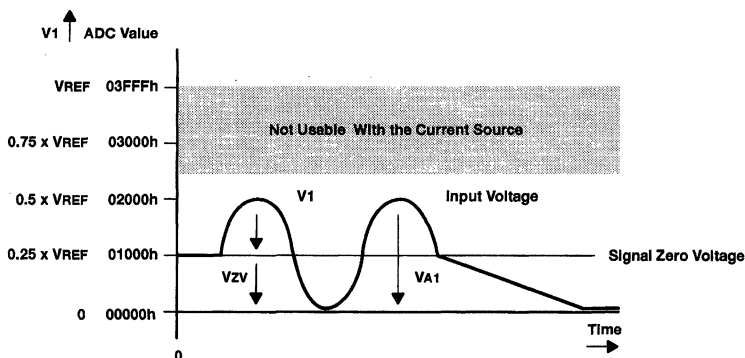
Figure 11. Signed Signals Shifted With the Current Source

Therefore the necessary shift resistor Rh is (Rh includes the internal resistance of the voltage source V1):

$$Rh = \frac{Vzv \times Rex}{0.25 \times V_{REF}}$$

with Vzv chosen to: $Vzv = 0.25 \times V_{REF} \rightarrow Rh = Rex$

Where: *Vzv*   Voltage of the signal midpoint (signal zero voltage)   [V]
          VREF   Voltage at the SVcc terminal (external or AVcc)   [V]
          Rex   Resistor between SVcc and Rext terminal (defines Ics)   [Ω]
          Rh   Shift resistor   [Ω]

The voltage V_{A1} at the analog input A1 is:

$$V_{A1} = V1 + Rh \times Ics = V1 + Rh \times \frac{0.25 \times VREF}{Rex}$$

The offset part (*Rh x Ics*) of the last equation is typically measured during a time when *V1* is known to be zero. This offset is stored in the RAM and subtracted from any measured value for *V1*. This leads to signed, offset corrected values for *V1*.

The unknown voltage *V1* is:

$$V1 = V_{A1} - Rh \times \frac{0.25 \times VREF}{Rex} = VREF \times \left(\frac{N}{2^{14}} - \frac{Rh \times 0.25}{Rex}\right)$$

With Rh=Rex: $V1 = VREF \times \left(\frac{N}{2^{14}} - 0.25\right)$

Figure 12 gives two practical examples for dc and ac measurements using the current source. Both applications measure signed voltages that are partially (the negative parts) out of the ADC range of the MSP430.
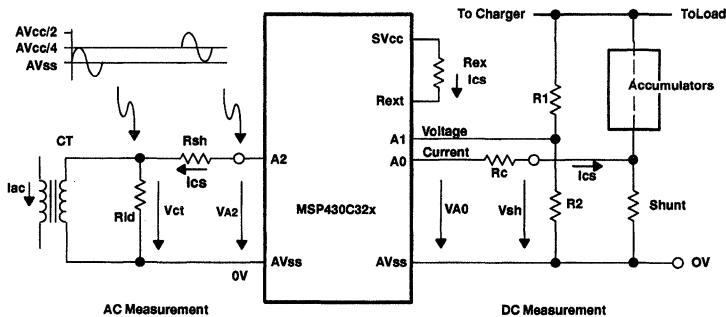
Figure 12. Signed Current Measurement With Level Shifting (Current Source)

AC Measurement: A current transformer CT is shown. Its output voltage is shifted into the ADC range by the current Ics of the current source and the resistor Rsh. The tolerable range for Ics is:

$$Ics_{min} < Ics < Idc_{max}$$

Icsmin is defined by the ADC specification, and Idcmax is given by the current transformer specification. Current transformers normally are sensitive to dc bias currents. Rcu is the resistance of the transformer's secondary winding (normally Rld >> Rcu).

$$V_{A2} = Vct + (Rsh + Rcu) \times Ics$$

This leads to:

$$Vct = V_{A2} - Rsh \times Ics = V_{REF} \times \left(\frac{N}{2^{14}} - \frac{0.25 \times (Rsh + Rcu)}{Rex} \right)$$

DC Measurement: The charge and discharge currents of an accumulator cause a voltage drop at the shunt resistor. This signed voltage drop *Vsh* is shifted into the ADC range by the resistor Rc (normally Rshunt $\ll$ Rc).

$$V_{A0} = Vsh + Rc \times Ics$$

This leads to:

$$Vsh = V_{A0} - Rc \times Ics = V_{REF} \times \left(\frac{N}{2^{14}} - \frac{0.25 \times Rc}{Rex} \right)$$

2.2.4 *Resistor Divider*

If the input voltages are high – which means normally higher than $10 \times V_{REF}$ – then, as shown in Figure 13, a simple resistor divider may be used for the level shift into the ADC range.
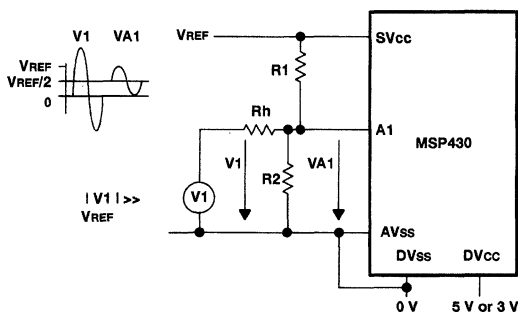
Figure 13. Resistor Divider for High Input Voltages

For input voltages V1 that are much higher than V_{REF}, the following equation is valid (Rh $\gg$ R2):

$$V_{A1} = V1 \times \frac{R1 \| R2}{R1 \| R2 + Rh} + V_{REF} \times \frac{R2}{R1 + R2} = \frac{N_{A1}}{2^{14}} \times V_{REF}$$

This leads to: $\quad V1 = V_{REF} \times \left(\frac{N_{A1}}{2^{14}} - \frac{R2}{R1 + R2} \right) \times \left(1 + \frac{Rh}{R1 \| R2} \right)$

To get the full accuracy of the ADC, the condition $R1 \| R2 < 27$ kΩ must be fulfilled.

For high input voltages V1 the resistors R1 and R2 are normally equal—it is not possible or necessary to correct the small error of the input signal—so the equation simplifies to:

$$V_{A1} = V1 \times \frac{R1}{R1 + 2 \times Rh} + 0.5 \times V_{REF} = \frac{N_{A1}}{2^{14}} \times V_{REF}$$

This leads to: $V1 = V_{REF} \times \left(\frac{N_{A1}}{2^{14}} - 0.5 \right) \times \left(1 + \frac{2 \times Rh}{R1} \right)$

The dc offset part (*0.5 × V_REF*) of the last equation is typically measured during a time when V1 is known to be zero. This measured offset is stored in the RAM and subtracted from any measured value for V1. This leads to signed, offset corrected values for V1.

For input voltages that have no dc-part (e.g., sinusoidal signals), the zero point can be calculated by an integration of the input signal. After a multiple *m* of the signal period, the integrated sum of ADC results equals *m* times the value of the zero point.

2.3   12-Bit Analog-to-Digital Conversion With Signed Signals

The asymmetrical arrangement of the four ADC ranges reduces the number of solutions that are possible with the 12-bit conversion:

- Normal phase splitter circuits are not able to shift the virtual ground into the middle of range A, B C or D as it is necessary here. See Table 2 column Vvg for the center values of the four ADC ranges.

- The split power supply method would need two voltages to get the zero point into the center of the used range: e.g., 0.625 V and 4.375 V for range A if a 5-V supply is used.

> **NOTE:** The formulas given in this section are valid only if both measurements for differences (ΔN) are measured in the same ADC range. If they are measured in different ADC ranges, then the 12–bit results need a correction (the missing two MSBs of the ADC result must be added). The correction numbers are:
> Range A: 0
> Range B: 1000h
> Range C: 2000h
> Range D: 3000h

2.3.1   Virtual Ground Circuitry

The phase splitter TLE2426 delivers only one half of the input voltage at its output terminal; it cannot be used here. With a simple op amp as shown in Figure 14, the necessary output voltages for the four ADC ranges can be obtained:
R = R1 + R2. See Table 1 for the relative resistor values.

Table 1. Resistor Ratios

ADC Range	Voltage VVG	R2	R1
A	$0.125 \times V_{REF}$	$0.125 \times R$	$0.875 \times R$
B	$0.375 \times V_{REF}$	$0.375 \times R$	$0.625 \times R$
C	$0.625 \times V_{REF}$	$0.625 \times R$	$0.375 \times R$
D	$0.875 \times V_{REF}$	$0.875 \times R$	$0.125 \times R$

Resistors R1 and R2 can have relatively high resistances. Only the offset current of the op amp limits these resistor values.
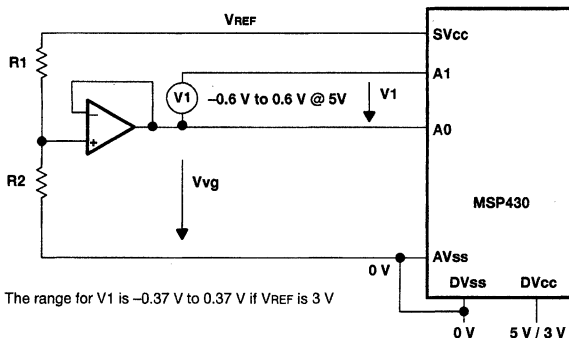
NOTE: The range for V1 is –0.37 V to 0.37 V if V$_{REF}$ is 3 V

Figure 14. Virtual Ground Circuitry for Level Shifting

The formula for the difference of the ADC results ΔN measured at the analog inputs A1 and A0 is:

$$\Delta N = (N_{A1} - N_{A0}) = \frac{V1 + Vvg}{V_{REF}} \times 2^{14} - \frac{Vvg}{V_{REF}} \times 2^{14} = \frac{V1}{V_{REF}} \times 2^{14}$$

This leads to the formula for *V1*:

$$V1 = V_{REF} \times \frac{\Delta N}{2^{14}}$$

Where: V1 Voltage to be measured inside of one ADC range   [V]
  ΔN Difference of two ADC results (here $N_{A1} - N_{A0}$)
  V$_{REF}$ Voltage at the SVcc terminal measured against AVss terminal [V]
  Vvg Voltage at the A0 input (center of the used ADC range)   [V]

EXAMPLE: The center voltage of the C range (at analog input A0) is measured and stored in location VIRTGR (register or RAM). The value of VIRTGR is subtracted from the ADC value measured at analog input A1; this gives the signed, offset corrected value for the input signal at the A1 input. The measurement subroutine MEASR of section 4.1 is used.

```
; Measure center voltage of range C at analog input A0 and
; store value for reference. MCLK = 3.3MHz: divide MCLK by 3
;
        MOV     #ADCLK3+RNGC+CSOFF+A0+VREF,&ACTL
        CALL    #MEASR                  ; Measure A0 (center voltage)
```

```
          MOV &ADAT,VIRTGR           ; Store result: 12-bit value
          ...
;
; Measure analog input signal V1 (0 ...0FFFh) and compute
; a signed, offset corrected value for V1 (0F800h...07FFh)
;
          MOV   #ADCLK3+RNGC+CSOFF+A1+VREF,&ACTL
          CALL  #MEASR              ; Measure A1 (input voltage V1)
          MOV   &ADAT,R5            ; Read ADC value for V1
          SUB   VIRTGR,R5           ; R5 contains signed delta N
          ...                       ; V1 = Vref x deltaN x 2^-14
```

2.3.2 Use of the Current Source

For signed signals it is necessary to shift the input signal V1 to the center of the ranges A or B. See Figure 15.
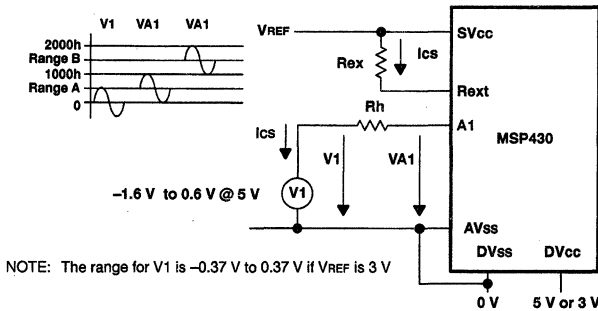
Figure 15. Current Source Used for Level Shifting

To get into the center of range *n* the necessary shift resistor *Rh* is:

$$Rh = 0.25 \times V_{REF} \times \frac{2n + 1}{2} \times \frac{Rex}{0.25 \times V_{REF}} \rightarrow Rh = (n + 0.5) \times Rex$$

The unknown voltage *V1* measured to its zero point in the center of range *n* is:

$$V1 = V_{Ax} - Rh \times Ics$$

With the above equation for *Rh* this leads to:

$$V1 = 0.25 \times V_{REF} \times \left(\frac{N}{2^{12}} + n - \frac{Rh}{Rex} \right)$$

2.3.3 Resistor Divider

The same circuitry is used as shown for the 14-bit conversion. See Figure 13. With the 12-bit conversion, it only makes sense to use the A range. This means for resistors R1 and R2, if R = R1 + R2:

$$R1 = 0.875 \times R \text{ and } R2 = 0.125 \times R.$$

For input voltages V1 that are much higher than V_{REF}, the following equation is valid (Rh >> R2):

$$V_{A1} = V1 \times \frac{R1 \| R2}{R1 \| R2 + Rh} + V_{REF} \times \frac{R2}{R1 + R2} = \frac{N_{A1}}{2^{14}} \times V_{REF}$$

With the above values for R1 and R2 this leads to:

$$V1 = V_{REF} \times 0.125 \times \left(\frac{N_{A1}}{2^{11}} - 1 \right) \times \left(1 + \frac{Rh}{0.125 \times 0.875 \times R} \right)$$

To get the full accuracy of the ADC, the condition R1||R2 < 27 kΩ must be fulfilled. This means R < 247 kΩ.

2.4 Reference Resistor Method

A system that uses sensors normally needs to be calibrated, due to the tolerances of the sensors themselves and of the ADC. A way to omit this costly calibration procedure is the use of reference resistors. Two methods can be used, depending on the type of sensor:

1. Platinum sensors (e.g., PT500, PT100): These are sensors with a precisely known temperature/resistance characteristic. Two precision resistors are used with the sensor resistances of the temperatures at the two limits of the temperature range.

2. Other sensors: Nearly all other sensors have insufficiently tight tolerances. This makes it necessary to group sensors with similar characteristics, and to select the two reference resistors according to the sensor resistances at the upper and the lower measurement range limits of these groups.

If the two reference resistors have—within the needed accuracy—the values of the sensors at the measurement range limits (or at other well-defined points) then all tolerances are eliminated during the calculation. Therefore, no calibration is necessary.

> **NOTE:** For voltage measurements, the reference method described above can be used with two reference voltages instead of two resistors. In this case, substitute voltages for the resistances used with the next equations.

2.4.1 Reference Resistor Method Without Amplification

This method can be used for the input range given by the current source—the A and B ranges and part of range C. For details, see *Architecture and Function of the MSP430 14-Bit ADC*[1]

The nominal formulas given in the previous section need to be modified if the tolerances of the ADC, the current source, the external components, and the sensor are considered. The ADC value Nx for a given resistor Rx is now:

$$Nx = \frac{Rx}{Rex} \times 2^{12} \times Slope + Offset$$

The slope and the offset are used for the correction of the measured result Nx. For the calculation of the slope and offset measurements with different resistors, Rx are necessary. With the hardware shown in figure 16 this calibration process can be omitted.
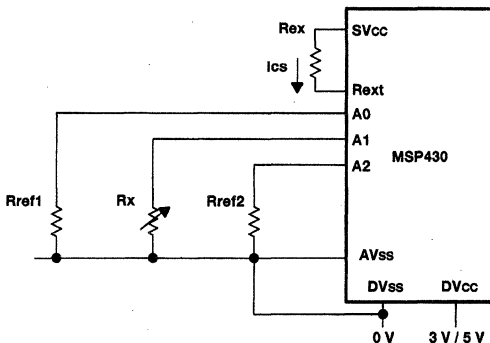
Figure 16. Referencing With Precision Resistors – No Amplification

With two known resistors Rref1 and Rref2 as shown in Figure 16, it is not necessary to know the slope and the offset to measure the value of the unknown resistor Rx exactly. Measurements are made for Rx, Rref1, and Rref2. The ADC results for these three measurements are:

$$Nx = \frac{Rx}{Rex} \times 2^{12} \qquad Nref1 = \frac{Rref1}{Rex} \times 2^{12} \qquad Nref2 = \frac{Rref2}{Rex} \times 2^{12}$$

The result of the solved equations shown above leads to:

$$Rx = \frac{Nx - Nref2}{Nref2 - Nref1} \times (Rref2 - Rref1) + Rref2$$

Where:
Nx	ADC conversion result for sensor Rx	
Nref1	ADC conversion result for reference resistor Rref1	
Nref2	ADC conversion result for reference resistor Rref2	
Rref1	Resistance of Rref1 (equals Rxmin)	[Ω]
Rref2	Resistance of Rref2 (equals Rxmax)	[Ω]

As shown, only known or measurable values are needed for the computation of Rx from Nx. Slope and offset influences of the ADC disappear completely:

- The offset disappears due to the two subtractions, one in the numerator and one in the denominator of the fraction above.

- The slope disappears due to the division

EXAMPLE: The values of these two reference resistors are chosen here for a PT1000 temperature sensor:

Rref1: 1000 Ω: The value of Rxmin. The resistance of a PT1000 sensor at 0°C (Tmin)

Rref2: 1380 Ω: The value of Rxmax. The resistance of a PT1000 sensor at 100°C (Tmax)

2.4.2 Reference Resistor Method With Amplification

If amplification is necessary to get a better resolution, then the solution shown below may be used. The full ADC range (0 to 3FFFh) can be used at analog input A1 despite the use of the current source at analog input A0. As with the section above, the offset and slope disappear; this is also true for the voltage drop at the outputs TP.x due to RDSon. The TP port of the measured resistor is switched to AVss potential; the other ones are set to Hi-Z.

The only error source of this arrangement is the difference of the internal resistances of the TP outputs (ΔRDSon). To minimize the influence of different internal resistances RDSon, only sensors with a minimum resistance should be used, e.g., PT1000 not PT100.

For the full 14-bit resolution at the analog input A1 the following design equations are valid (Rref2 > Rref1). They simplify this way if Rex is chosen to:

$$Rex = \frac{Rref2}{2}$$

This results in a maximum voltage of VREF/2—the safe maximum output voltage the current source can deliver—at the analog input A0 for the maximum resistor value Rref2.

$$Vm = \frac{Rref1}{Rref1 + Rref2} \times V_{REF} \qquad v = \frac{V_{REF}}{V_{REF} - 2 \times Vm} = \frac{R1}{R2 \| R3}$$

The calculated amplification v of the op amp needs to be reduced by 10 to 15% to be sure that V_{A1} does not saturate under worst case conditions.
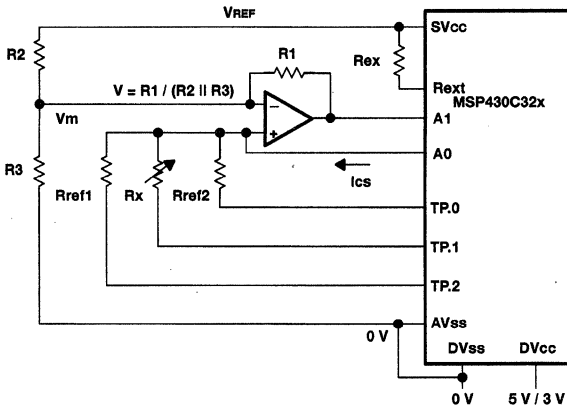
Figure 17. Referencing With Precision Resistors – With Amplification

As Figure 17 shows, with two known resistors Rref1 and Rref2 it is possible to get the values of unknown resistors exactly. The result of the solved equations gives:

$$Rx = \frac{\Delta Nx - \Delta Nref2}{\Delta Nref2 - \Delta Nref1} \times (Rref2 - Rref1) + Rref2$$

Where: ΔNx Difference of the two ADC results for Rx ($N_{A1}-N_{A0}$)
$\Delta Nref1$ Difference of the two ADC results for Rref1 ($N_{A1}-N_{A0}$)
$\Delta Nref2$ Difference of the two ADC results for Rref2 ($N_{A1}-N_{A0}$)
Vm Voltage generated by the resistor divider R2 and R3

The differences named above are the differences between the ADC conversion results measured at the analog inputs A1 and A0 for each resistor: $\Delta N = N_{A1} - N_{A0}$.

3 Hum and Noise Considerations

3.1 Connection of Long Sensor Lines

If the distance from the MSP430 to the sensor is long (>30 cm) then it is recommended to use a shielded cable between the microcomputer and the sensor. This avoids spikes at the ADC input that cause measurement errors, and also gives protection to the ADC input. Figure 18 shows this schematic on the left side. In the same way, four-wire circuitry may be connected to the MSP430.

If a shielded cable cannot be used, the circuitry shown on the right side of Figure 18 should be used; the AVss line in parallel to the signal line gives a relatively good screening. Twisting the two lines increases the protection.

To protect the measurement against spikes, hum, and other unwanted noise see Section 5.3, *Signal Averaging and Noise Cancellation.* This section shows additional possibilities for the minimization of these influences by software.
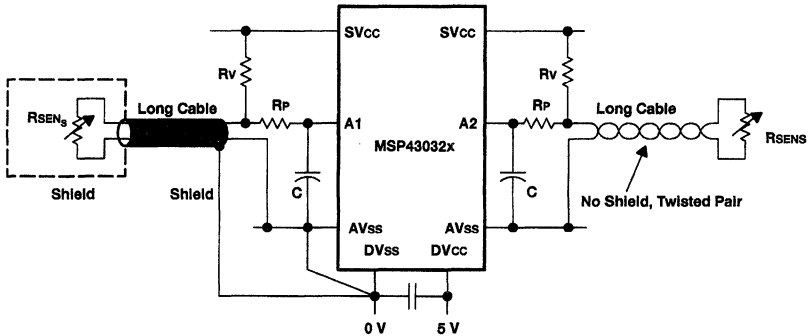
Figure 18. Sensor Connection via Long Cables With Voltage Supply

With the circuitry of figure 18, the minimum time tdelay between the switch-on of the voltage SVcc and the actual measurement—to get the full 14-bit accuracy—is:

$$t_{delay} > ln2^{14} \times \tau max = 9.704 \times \tau max \approx 10 \times \tau max$$

The value of τmax is:

$$\tau max = (Rp + Rsens_{max}\|Rv) \times C$$

If the current source is used, then:

$$Rv = \infty : \quad \tau_{max} = (Rp + Rsens_{max}) \times C$$

3.2 Grounding

Correct grounding is very important for ADCs with high resolution. There are some basic rules that need to be observed[1]. See Figure 19 also.

1. Use a separate analog and digital ground plane wherever possible: thin traces from the battery to terminals DVss and AVss should be avoided.

2. The AVss terminal should serve as a star point for all analog ground connections e.g. sensors, analog input signals. The DVss terminal should serve as a star point for all digital ground connections e.g. switches, keys, power transistors, output lines, digital input signals.

3. The battery and storage capacitor Cb should be connected close together (the capacitor Cb is needed for batteries with a relatively high internal resistance). From this capacitor two different paths go to the analog and the digital supply terminals. Two small capacitors are connected across the digital (Cd) and the analog (Ca) supply terminals. See Figure 19.

4. Rules 1 to 3 above are also true for the Vcc paths (DVcc and AVcc).

5. The AVss and DVss terminals must be connected together externally; they are not connected internally. The same is true for the AVcc and DVcc terminals. These connections should be made with the configuration shown in Figure 19.

6. The coil L should be used in very difficult cases.

7. The connections of the capacitor Cb are the star point of the complete system. This is due to the low impedance of this capacitor.
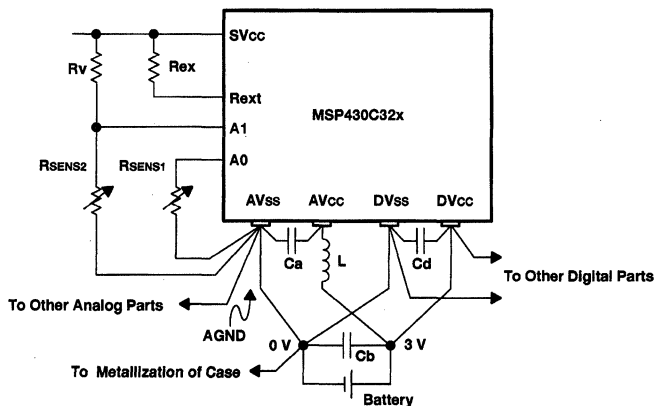
Figure 19. Analog-to-Digital Converter Grounding

If a metalized case is used around the printed circuit board containing the MSP430 then it is very important to connect the metallization to the ground potential (0 V) of the board. Otherwise the behavior is worse than without the metalization.

---

[1] These grounding rules were developed by E. Haseloff of TID.

3.3 Routing

Correct routing for a PC board is very important for minimum noise. Figure 20 shows a simplified routing that is not optimal; the gray areas receive EMI from external sources. For a minimum influence coming from external sources these areas must be as small as possible.
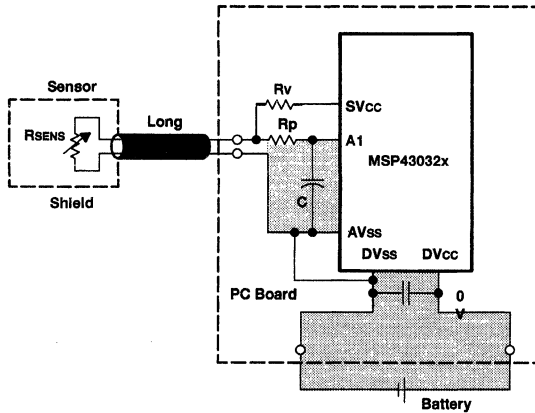
Figure 20. Routing That Is Sensitive to External EMI

Figure 21 shows an optimized routing; the areas that may fetch noise have a minimum size.
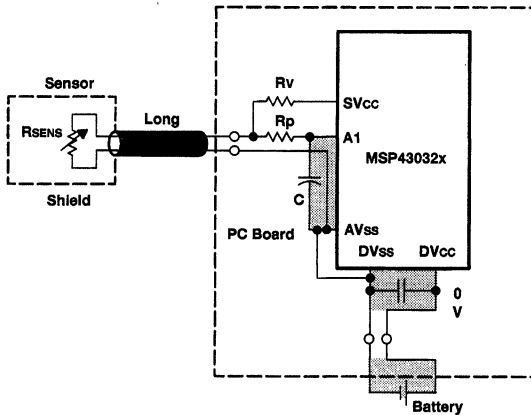
Figure 21. Routing for Minimum EMI Sensitivity

4 Enhancement of the Resolution

Many applications need a higher resolution than the 14-bit ADC can provide. For these applications the following hints may be helpful.

NOTE: These enhancements make it necessary to pay attention to the rules given in Chapter 3. Without observing these rules strictly, no enhancement will be seen.

4.1 16-Bit Mode With the Current Source

With the use of two additional output terminals (I/O-ports or TP-outputs) the 14-bit ADC may be expanded to a resolution of nearly 16 bits. The principle is simple: the resistor Rex of the current source is modified by paralleling two additional resistors (see Figure 23). These resistors have values that represent one half and one quarter of a single ADC-step. Due to the fact that these fractions of a step are accurate only at one point of the ADC-range, this enhancement gives only better resolution, not better accuracy. To get the 16-bit result, four measurements are necessary: one for every combination of the two additional resistors. If the results of these four measurements are added, a 16-bit result is reached. See Figure 22.
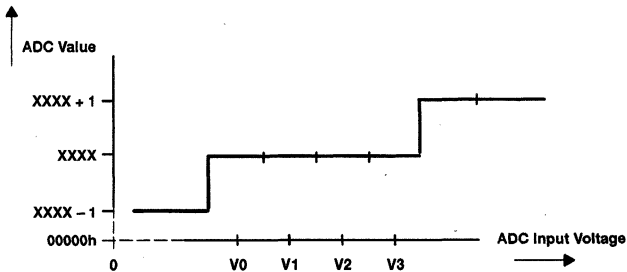
Figure 22. Dividing of an ADC-Step Into Four Steps

Table 2 shows the different results of these four measurements for the four possible input voltages V0 to V3 inside of one ADC-step; the table refers to the hardware shown in Figure 23.

Table 2. Measurement Results of the 16-Bit Method

INPUT VOLTAGE	MEASUREMENT 1 TP.1: HI-Z TP.0: HI-Z	MEASUREMENT 2 TP.1: HI-Z TP.0: HI OUT	MEASUREMENT 3 TP.1: HI OUT TP.0: HI-Z	MEASUREMENT 4 TP.1: HI OUT TP.0: HI OUT	MEAN VALUE (BINARY)
V0	XXXX	XXXX	XXXX	XXXX	XXXX.00
V1	XXXX	XXXX	XXXX	XXXX+1	xxxx.01
V2	XXXX	XXXX	XXXX+1	XXXX+1	XXXX.10
V3	XXXX	XXXX+1	XXXX+1	XXXX+1	XXXX.11

Simplified Circuity of a TP–Output
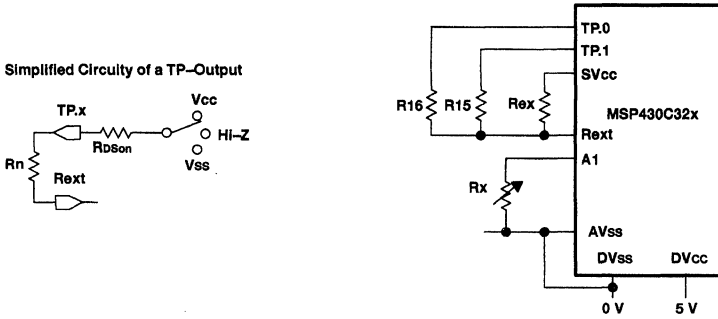
Figure 23. Hardware for a 16-Bit ADC

The values for resistors R16 and R15 are:

$$Rp = \frac{2^{14} \times 0.25 \times Rx0}{m} = \frac{2^{12} \times Rx0}{m}$$

Where: Rp Parallel resistor to Rex (here R14 and R15)   [Ω]

           $Rx0$ Sensor resistance at the point of the highest accuracy   [Ω]

           m Fraction of an ADC step (0.25 or 0.5)

EXAMPLE: With the hardware shown in Figure 23, four 16-bit measurements are made. The result is placed into R5. The software may also be written with a loop. The software assumes ascending order for the two TP outputs.

```
        MOV     #RNGAUTO+CSA1+A1+VREF,&ACTL  ; Define ADC
        BIC.B   #TP1+TP0,&TPE       ; TP.0 and TP.1 to Hi-Z
        BIS.B   #TP1+TP0,&TPD       ; Set TPD.0 and TPD.1 to Hi
        CALL    #MEASR              ; Measure with R15 = R16 =
                                    ; Hi-Z
        MOV     &ADAT,R5            ; 14-bit value to result
        ADD.B   #TP0,&TPE           ; Set R16 to Hi-Out
        CALL    #MEASR              ; Measure
        ADD     &ADAT,R5            ; Add 14-bit value to
                                    ; result
        ADD.B   #TP0,&TPE           ; Set R15 to Hi-Out,R16 to
                                    ; Hi-Z
        CALL    #MEASR              ; Measure
        ADD     &ADAT,R5            ; Add 14-bit value to
                                    ; result
        ADD.B   #TP0,&TPE           ; Set R15 and R16 to Hi-Out
        CALL    #MEASR              ; Measure
        ADD     &ADAT,R5            ; Add 14-bit value to
                                    ; result
        BIC.B   #TP1+TP0,&TPE       ; TP.n off
        ...                         ; 16-Bit result 4N in R5
;
; The measurement routine used above:
;
MEASR   BIC.B   #ADIFG,&IFG2        ; Clear EOC flag
        ...                         ; Insert delays here (NOPs)
        BIS     #SOC,&ACTL          ; Start measurement
M0      BIT.B   #ADIFG,&IFG2        ; Conversion completed?
        JZ      M0                  ; No
        RET                         ; Result in ADAT
```

4.2 Enhanced Resolution Without Current Source

The principle is explained in the last section. Figure 24 shows a hardware proposal for the measurement part of a scale using the MSP430C32x. With the resistor Rn, the resolution of the MSP430 ADC is increased to 15 bits:

- TP.0 is off (Hi-Z): normal measurement
- TP.0 is switched to Vcc: the current into the right bridge leg increases the voltage at A1 by 0.5 steps of the ADC

Two differential ADC measurements ($N_{A0} - N_{A1}$)—one with TP.0 off and one with TP.0 switched to Vcc—are summed-up and provide (nearly) 15-bit resolution. The result of these four measurements is $2 \times \Delta N$.

The formulas derived in the Connection of Bridge Assemblies section are valid here as well.
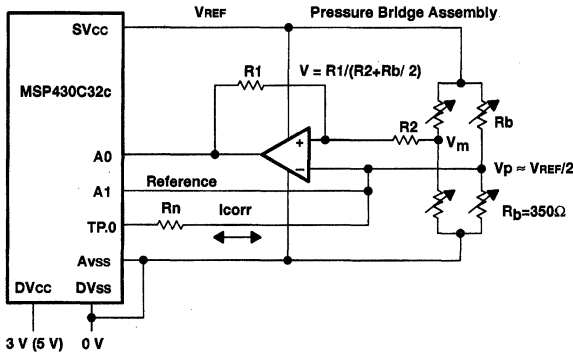
Figure 24. ADC-Resolution Expanded to 15 Bits

The formula for Rn to cause a voltage difference ΔV_{A0} (here 0.5 ADC steps) at the ADC input is:

$$\Delta V_{A0} = \frac{V_{REF}}{2^{15}} = \frac{Rb}{2 \times Rn + Rb} \times \left(V_{CC} - \frac{V_{REF}}{2} \right) \times v$$

This gives an approximate value for Rn (Vcc = VREF):

$$Rn \approx Rb \times 2^{13} \times \frac{R1}{R2}$$

Where:			
	ΔV_{A0}	Change of the input voltage at input A0 due to Rn	[V]
	Rb	Resistance of a half bridge leg (here 350 Ω)	[Ω]
	Rn	Resistance of the resistor for 15 bits resolution	[Ω]
	v	Amplification of the operational amplifier: v = R1/R2	
	VREF	Supply voltage at the SVcc terminal (int. or ext.)	[V]
	DVcc	Supply voltage at DVcc terminal (output voltage of TP.0)	[V]
	R1,R2	Resistors defining the amplification of the op amp	

Without any change to the hardware above, the resolution of the ADC can be increased to 15.5 bits (this method is only possible with sensor assemblies like those shown in Figure 24, that deliver output voltages near 0.5 x Vref):

- TP.0 is off (Hi-Z): normal measurement
- TP.0 is switched to Vcc: the current into the right bridge leg increases the voltage at A1 by 0.5 steps of the ADC
- TP.0 is switched to Vss: the current out of the right bridge leg decreases the voltage at A1 by 0.5 steps of the ADC

Three differential ADC measurements $(N_{A0} - N_{A1})$—one with TP.0 switched to Hi-Z, one with TP.0 switched to Vss, and one switched to Vcc—are summed-up and provide (nearly) 15.5-bit resolution. The calculations following these six measurements must be changed for an input value of $3 \times N$.

ΔV_{A0} is chosen to: $\quad \Delta V_{A0} = \dfrac{V_{REF}}{3 \times 2^{14}}$

The circuitry of Figure 24 leads to very high values of Rn with high amplifications v: for the above example Rn = 286 MΩ for v = 100. If these resistor values are too high, then the circuitry shown in Figure 25 should be used. The registor values of R15 and R16 have the same effect as the circuitry in Figure 24, but are much smaller.
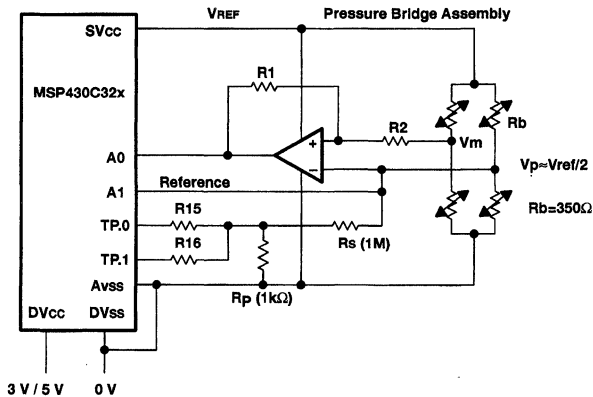
Figure 25. ADC-Resolution Expanded to 16 Bits

The formula for R15 and R16 to cause a voltage difference ΔV_{A0} of 0.5 ADC steps for R15 and 0.25 ADC steps for R16 at an ADC input is now:

$$Rn \approx 2^n \times \frac{R1}{R2} \times \frac{Rp}{Rs} \times \frac{Rb}{2}$$

Where: n Bit number of resolution resistor (15 or 16)
Rn Resistance of resolution resistor (bit n)   [Ω]
Rb Resistance of a half bridge leg (here 350 Ω)   [Ω]
Rp Parallel resistor (chosen to be 1k: small compared to 1 M)   [Ω]
Rs Serial resistor (chosen to be 1M: large compared to 350 Ω)   [Ω]

With the circuitry of Figure 25 (v = 100) R15 now becomes 573 k and R16 becomes 1.15M.

The necessary four measurements are described in Table 2. Each measurement consists of two ADC conversions that are subtracted afterwards ($\Delta N = N_{A0} - N_{A1}$). The four differences ΔN are summed and deliver a 16-bit result with nearly two bits more resolution than the normal 14-bit result. The result in R5 is $4 \times \Delta N$.

EXAMPLE: With the hardware shown in Figure 25, four differential measurements for ΔN are made ($\Delta N = N_{A0} - N_{A1}$). The four values for ΔN are summed in R5. The software assumes ascending order for the two TP outputs (TP.x and TP.x+1).

```
        BIC.B   #TP1+TP0,&TPE           ; TP.0 and TP.1 to Hi-Z
        BIS.B   #TP1+TP0,&TPD           ; Set TPD.0 and TPD.1 to Hi
        MOV     #RNGAUTO+A0+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure with R15 = R16 = Hi-Z
        MOV     &ADAT,R5                ; 14-bit value to result
        MOV     #RNGAUTO+A1+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure with R15 = R16 = Hi-Z
        SUB     &ADAT,R5                ; (NA0 - NA1) to result
;

        ADD.B   #TP0,&TPE               ; Set R16 to Hi-Out, R15 = Hi-Z
        MOV     #RNGAUTO+A0+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        ADD     &ADAT,R5                ; Add 14-bit value to result
        MOV     #RNGAUTO+A1+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        SUB     &ADAT,R5                ; (NA0 - NA1) to result
;

        ADD.B   #TP0,&TPE               ; Set R15 to Hi-Out,R16 to Hi-Z
        MOV     #RNGAUTO+A0+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        ADD     &ADAT,R5                ; Add 14-bit value to result
        MOV     #RNGAUTO+A1+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        SUB     &ADAT,R5                ; (NA0 - NA1) to result
;

        ADD.B   #TP0,&TPE               ; Set R15 and R16 to Hi-Out
        MOV     #RNGAUTO+A0+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        ADD     &ADAT,R5                ; Add 14-bit value to result
        MOV     #RNGAUTO+A1+VREF,&ACTL  ; Define ADC
        CALL    #MEASR                  ; Measure
        SUB     &ADAT,R5                ; (NA0 - NA1) to result
;

        BIC.B   #TP1+TP0,&TPE           ; TP.n off
        ...                             ; 16-Bit result 4xN in R5
;
```

4.3 Calculated Resolution of the 16-Bit Mode

4.3.1 16-Bit Mode With the Current Source

To give an idea of how much better the results of the 16-bit mode can be compared to the 14-bit mode of the ADC, the results of four calculations are shown in Table 3. The table shows the statistical results for the deviations of the corrected result in ADC-steps:

- The first column shows the statistical results for the normal 14-bit ADC
- The second column shows the statistical results for measurements that have the highest accuracy at the lowest sensor value: Rx0 = 1000 Ω
- The third column shows the statistical values if the point of highest accuracy is moved to the midpoint of the sensor resistance: Rx0 = 1190 Ω
- The fourth column shows the same as before if the highest sensor value is used for the highest accuracy: Rx0 = 1380 Ω

Calculation values and explanations:

Rxmax:	1380.0 Ω	Highest sensor resistance (100°C for PT1000)
Rxmin:	1000.0 Ω	Lowest sensor resistance (0°C for PT1000)
Rx0:		Sensor resistance for highest accuracy (3 different values)
ΔRx:	0.01 Ω	Step width for resistance value during calculation
Rex:	690.0 Ω	Calculated external resistor for the Current Source
R15:		Calculated resistor for the 15th bit
R16:		Calculated resistor for the 16th bit

Table 3. Calculation Results for Different 16-Bit Corrections

ITEM	NO CORRECTION 14-BIT	Rx0 = 1000 Ω 16-BIT	Rx0 = 1190 Ω 16-BIT	Rx0 = 1380 Ω 16-BIT
R15	N/A	8.2MΩ	9.7MΩ	11.3MΩ
R16	N/A	16.4MΩ	19.5MΩ	22.6MΩ
Mean value	−0.5001	−0.0538	−0.1250	−0.1767
Standard deviation	0.2887	0.1019	0.0841	0.0898
Variance	0.0833	0.0104	0.0071	0.0081

Table 3 shows the improved resolution especially if the best resolution is programmed for the lowest sensor resistance (Rx0 = 1000 Ω). The result is derived from 38,000 measurements with a step width of 0.01 Ω. The 14-bit results show the (correct) inherent error of minus 0.5 steps that is enhanced with the three 16-bit modes by a factor of 3 to 9.

4.3.2 16-Bit Mode Without the Current Source

Circuitry like shown in Figure 25 is normally used: this means the input voltage of the analog inputs is always near 0.5 × Vref. Therefore the results of Table 3 column Rx0 =1190 Ω (highest accuracy at the center of the resistance range) are valid.

5   Hints and Recommendations

5.1   Replacement of the First Measurement

In certain cases the first measurement is discarded. Instead, a second measurement is started and used. This method is especially useful if the settling time for the ADC is insufficient.

```
MOV     #XX,&ACTL           ; Define ADC
CALL    #MEASR              ; 1st measurement (not used)
CALL    #MEASR              ; 2nd measurement is used
MOV     &ADAT,R5            ; for calculations. Result to R5
```

5.2   Grounding and Routing

With increasing ADC accuracy and CPU frequency, the board layout becomes more important. A few hints may help to increase the performance of the ADC:

- To avoid cross talk from one ADC input line to the other one, grounded lines (AVss potential) between the analog input lines are recommended.

- Large ground planes (0V potential) should be used wherever possible. Any free space on the board should be used for this purpose.

- Analog input lines should be as short as possible. If this is not possible, input filtering may be necessary.

- To get reliable ADC results in noisy environments, additional hardware and software filtering should be used. Chapter 5 describes several methods to do this in Section 5.3, *Signal Averaging and Noise Cancellation*: over sampling, continuous averaging, weighted summation, rejection of extremes, and synchronization to hum. Tested software examples are included.

See also sections 3.2 and 3.3.

5.3   Supply Voltage and Current

Completely different environments exist for battery and mains driven systems. A few hints are given for these two supplies. More information concerning this topic is included in Section 3.8, *Power Supplies*.

5.3.1   Influence of the Supply Voltage

The supply voltage is used for reference purposes if the Vref-bit (ACTL.1) is set. This means a change of the analog supply voltage AVcc during the measurement influences the final ADC result. The same is true for an external reference voltage.

Figure 26 shows a decreasing analog supply voltage with the ADC timing. The error of the ADC result N is mainly introduced during the conversion time for the 12 LSBs of the ADC result. The input sample is taken with an AVcc voltage Vref0, the LSB is generated with an AVcc voltage Vref1. The two results have the ratio:

$$\frac{N_1}{N_0} = \frac{Vref0}{Vref1}$$

The maximum error emax in per cent is therefore:

$$e_{max} = \frac{N_1 - N_0}{N_0} \times 100 = \left(\frac{Vref0}{Vref1} - 1 \right) \times 100$$

Where: N_0 ADC result measured with a stable AVcc of Vref0

N_1 ADC result measured with a stable AVcc of Vref1

e_{max} Maximum error caused by unstable AVcc   [%]

Vref0   Value of AVcc during the sampling of the conversion

        sample   [V]

Vref1   Value of AVcc at the end of conversion   [V]
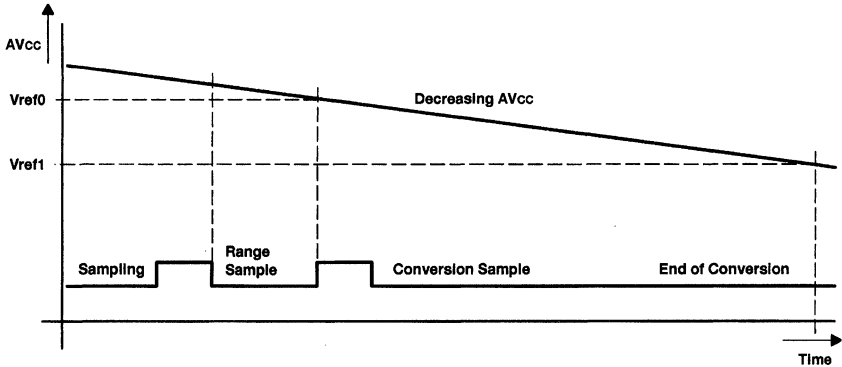
Figure 26. Influence of the Supply Voltage

The result caused by an unstable AVcc can normally be detected by its trailing series of zeroes or ones. If, during the conversion, one of the leading bits is set, or reset, and this bit has the wrong state for the changing reference voltage, then, all remaining bits will have the same value, e.g., 1 for a decreasing AVcc.

5.3.2 Battery Driven Systems

If the battery used has a high internal resistance Ri (like some long-life batteries) then the parallel capacitor Cb (see Figure 19) must have a minimum capacity Cbmin: the supply current for the measurement part—which cannot be delivered by the battery—is delivered mainly by Cb; the approximate equation includes the small current coming from the battery:

$$Cb_{min} \geq tmeas \times \frac{I_{AM}}{\Delta Vb} - \frac{1}{Ri}$$

If the battery has a high impedance Ri, then it is recommended to use the kind of measurement shown in *Architecture and Function of the MSP430 14-Bit ADC*:[1] the CPU is switched off during the ADC measurement which lowers the current out of the battery.

Between two ADC measurements, the capacitor Cb needs a time tch to become charged to Vcc potential for the next measurement. During this charge-up time the MSP430 system runs in low power mode 3 to have the lowest possible power consumption. The charge time tch to charge Cb to 99% of Vcc is:

$$tch_{min} \geq 5 \times Cb_{max} \times Ri_{max}$$

Where:
Cb	Capacitor in parallel to the battery	[F]
I_{AM}	Medium system current (MSP430 and ADC)	[A]
$tmeas$	Discharge time of Cb during measurement	[s]
ΔVb	Tolerable discharge voltage of Cb during time tmeas	[V]
Ri	Internal resistance (impedance) of the battery	[Ω]
tch	Charge-up time for the capacitor Cb	[s]

5.3.3 Mains Driven Systems

No hum, noise, or spikes are allowed for the supply voltages AVcc and DVcc. If present, the reliability of the system and the accuracy of the ADC will decrease. This is especially true for applications where the AVcc voltage is used for the ADC reference [ACTL.1 = 1 (Vref bit)]. See Section 5.3, *Signal Averaging and Noise Cancellation* for ways to overcome this problem.

5.3.4 Current Consumption

Often it is important to know the current consumption of the complete MSP430 system—which means including the supply current of the MSP430 and its ADC. The supply current of the CPU increases nearly linearly with the MCLK frequency and the applied supply voltage DVcc, but this is not the case for the ADC: the main component of the ADC supply current is drawn by the resistor divider with its 4×128 resistors. An approximate formula for the nominal current consumption Icc of the MSP430C32x is (internal ADC reference):

$$I_{CC} = ICC_{digital} + ICC_{analog} = \left(\frac{V_{DVcc}}{5V} \times \frac{f_{MCLK}}{1MHz} \times 750 \ \mu A \right) + \left(\frac{V_{AVcc}}{3V} \times 200 \ \mu A \right)$$

Where:
Icc	Complete current consumption of MSP430 (nominal)	[µA]
$ICC_{digital}$	Current consumption of the digital parts	[µA]
ICC_{analog}	Current consumption of the ADC	[µA]
V_{DVcc}	Voltage at the DVcc terminal	[V]
V_{AVcc}	Voltage at the AVcc	[V]
f_{MCLK}	Frequency of the system clock generator (MCLK)	[Hz]

5.4 Use of the Floating Point Package

For the MSP430 a Floating Point Package exists with two selectable bit lengths: 32 bit and 48 bit. For calculations with the ADC, results consisting of several multiplications and divisions, it is recommend to use this package: no decrease of accuracy is caused by the calculation itself. A detailed description of the Floating Point Package and all available mathematical functions is given in the *MSP430 Application Report*. See Section 5.6, *The Floating Point Package*.

A small example is given below: the measured ADC result—in ADC buffer ADAT—is corrected with slope and offset. The result (BCD format) is placed into the locations BCDMSD, BCDMID and BCDLSD (RAM or registers).

```
DOUBLE    .EQU    0               ; Use .FLOAT format (32 bits)
;
          MOV     #xxx,&ACTL      ; Define ADC measurement
          CALL    #MEASR          ; Measure. Result to ADAT
          CALL    #FLT_SAV        ; Save registers R5 to R12
```

```
                SUB       #4,SP              ; Allocate stack for FP result
                MOV       #ADAT,RPARG        ; Load address of ADC buffer
                CALL      #CNV_BIN16U        ; Convert ADC result to FP
;
; Calculate: ADCcorr = (ADC result x Slope) + Offset
;
                MOV       #Slope,RPARG       ; Load address of slope
                CALL      #FLT_MUL           ; ADC result x Slope
                MOV       #Offset,RPARG      ; Load address of offset
                CALL      #FLT_ADD           ; ADC result x Slope + Offset
                ...                          ; Continue with calculations
;
; The final result is converted to BCD format for the display
;
                CALL      #CNV_FP_BCD        ; Convert FP result to BCD
                JN        CNVERR             ; Result too big for BCD buffer
                POP       BCDMSD             ; BCD number: sign and MSDs
                POP       BCDMID             ; BCD digits MSD-4 to LSD+4
                POP       BCDLSD             ; BCD digits LSD+3 to LSD
                                             ; Stack is corrected by POPs
                CALL      #FLT_REC           ; Restore registers R12 to R5
                ...                          ; Continue with program

Slope           .FLOAT    -1.2345            ; Slope (fixed, RAM, EEPROM)
Offset          .FLOAT    14.4567            ; Offset (fixed, RAM, EEPROM)
CNVERR          ...                          ; Start error handler
```

6 Additional Information

This application report is complemented by the *Additive Improvement of the MSP430 14-Bit ADC Characteristic* application report[5] that explains several methods to minimize the error of the 14-Bit ADC. For all methods (linear, quadratic, cubic and others) the actual improvement for a measured ADC characteristic is shown. The enhancement methods discussed are compared completely with statistic results, advantages and disadvantages, necessary CPU cycles, and storage needs.

7 References

1. *Architecture and Function of the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA045

2. *MSP430 Application Report*, 1998, Literature #SLAAE10C

3. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B

4. *MSP430C325, MSP430P323 Data Sheet*, 1999, Literature #SLAS219

5. *Additive Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA047

6. *Linear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA048

7. *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA050

Appendix A   Definitions Used With the Application Examples

```
; HARDWARE DEFINITIONS
;
AIN       .equ   0110h    ; Input register (for digital inputs)
AEN       .equ   0112h    ; 0: analog input   1: digital input
;
ACTL      .equ   0114h    ; ADC control register: control bits
SOC       .equ   01h      ; Conversion start
VREF      .equ   02h      ; 0: ext. reference    1: SVcc on
A0        .equ   00h      ; Input A0
A1        .equ   04h      ; Input A1
A2        .equ.  08h      ; Input A2
A3        .equ   0Ch      ; Input A3
A4        .equ   10h      ; Input A4
A5        .equ   14h      ; Input A5
CSA0      .equ   00h      ; Current Source to A0
CSA1      .equ   40h      ; Current Source to A1
CSA2      .equ   80h      ; Current Source to A2
CSA3      .equ   0C0h     ; Current Source to A3
CSOFF     .equ   100h     ; Current Source off
CSON      .equ   000h     ; Current Source on
RNGA      .equ   000h     ; Range select A (0 ... 0.25xSVcc)
RNGB      .equ   200h     ; Range select B (0.25..0.50xSVcc)
RNGC      .equ   400h     ; Range select C (0.5...0.75xSVcc)
RNGD      .equ   600h     ; Range select D (0.75..SVcc)
RNGAUTO   .equ   800h     ; 1: range selected automatically
PD        .equ   1000h    ; 1: ADC powered down
ADCLK1    .equ   0000h    ; ADCLK = MCLK
ADCLK2    .equ   2000h    ; ADCLK = MCLK/2
ADCLK3    .equ   4000h    ; ADCLK = MCLK/3
ADCLK4    .equ   6000h    ; ADCLK = MCLK/4
;
ADAT      .equ   0118h    ; ADC data register (12 or 14-bit)
;
IFG2      .equ   03h      ; Interrupt flag register 2
ADIFG     .equ   04h      ; ADC "EOC" bit (IFG2.2)
;
IE2       .equ   01h      ; Interrupt enable register 2
ADIE      .equ   04h      ; ADC interrupt enable bit (IE2.2)
;
TPD       .equ   04Eh     ; TP-port: address data register
TPE       .equ   04Fh     ; TP-port: address of enable register
TP0       .equ   1        ; Bit address of TP.0
TP1       .equ   2        ; Bit address of TP.1
```

Additive Improvement of the MSP430 14-Bit ADC Characteristic

Lutz Bierl

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

Additive Improvement of the MSP430 14-Bit ADC Characteristic

Lutz Bierl

ABSTRACT

This application report shows different simple methods to improve the accuracy of the 14-bit analog-to-digital converter of the MSP430 family. They all use only addition for the correction of the analog-to-digital converter characteristic. Different correction methods are explained—all without the need for multiplication—which makes them usable for real time systems like electronic electricity meters. The methods used differ in RAM and ROM allocation, reachable improvement, and complexity. The external hardware for the measurement of the analog-to-digital converter characteristic is also described. For all correction methods, proven, optimized software examples are given. The *References* section at the end of the report lists related application reports in the MSP430 14-bit ADC series.

1 Introduction

The application report *Architecture and Function of the MSP430 14-Bit ADC*[1] gives a detailed overview to the architecture and function of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. The principle of the ADC is explained and software examples are given. Also included are the explanation of the function of all hardware registers contained in the ADC.

The application report *Application Basics for the MSP430 14-Bit ADC*[2] shows several applications of the 14-bit ADC of the MSP430 family. Proven software examples and basic circuitry are shown and explained.

Figure 1 shows the block diagram of the 14-bit analog-to-digital converter of the MSP430 family.
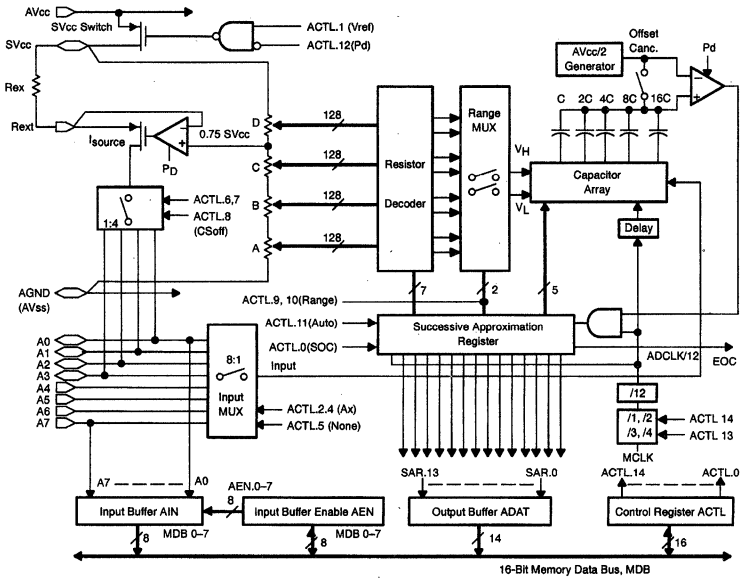
Figure 1. The Hardware of the 14-Bit Analog-to-Digital Converter

The methods for the improvement of the ADC described in the next sections are:

- Correction with the mean value of the full ADC range
- Correction with the mean values of the four ranges
- Correction with the centers of the four ranges
- Correction with multiple sections

Linear, quadratic, and cubic corrections are explained in *Linear Improvement of the MSP430 14-Bit ADC Characteristic*[3] and nonlinear improvements are discussed in the *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*[4].

2 The External Calibration Hardware for the ADC

All of the methods of improvement discussed in this report need to know the actual errors of the ADC at different points of the four ADC ranges. See Figure 7 for an example of a noncorrected ADC characteristic.

2.1 Measurement Methods for the ADC Reference Samples

The characterization of the ADC for this report is made with three different methods:

- External digital-to-analog converter (DAC): an accurate DAC—controlled by the measured MSP430—produces precise analog output voltages that are measured with the 14-bit ADC. The difference of the two numbers is the absolute error of the ADC.
- External discrete, precise voltages: the MSP430 controls its input voltage via an external analog multiplexer. If only a few accurate input voltages are needed, then this method is best.
- External precision resistors: the MSP430 controls which resistor is measured. For systems that measure the resistance of sensors, this method is best.

Several other methods exist to measure the errors of different reference points for improvement of the ADC characteristic including:

- Measurement of a single ADC sample: fastest way, but not recommended due to statistical reasons.
- Multiple measurements of the same point and calculation of the mean value: e.g. 16 measurements.
- Multiple measurements of the errors around a given point and calculation of the mean value: e.g. 16 measurements ±8 (or ±32) around the center point of interest.
- External 16-bit DAC: measurement of all possible four points (xxx.00, xxx.01, xxx.10, xxx.11 for the 14-bit value xxx) and summing them up. This gives an additional 2 bits of resolution.
- Sophisticated statistical methods.
- Measurement of 12 samples for the same ADC point and rejection of the two extreme values. The remaining 10 samples are averaged.

These error measurement methods may be used for all of the given improvement methods in this report. However, they are not discussed with the description of the improvement methods. See also Section 5.3, *Signal Averaging and Noise Cancellation*.

This application report only uses simple measurement methods.

2.2 External Digital-to-Analog Converter

The external hardware connected to the MSP430-PC board (see Figure 3) is used to obtain the necessary information about the characteristic of the ADC. Its main part is a precise 14- or 16-bit digital-to-analog converter (DAC). Figure 2 shows the calibration process:
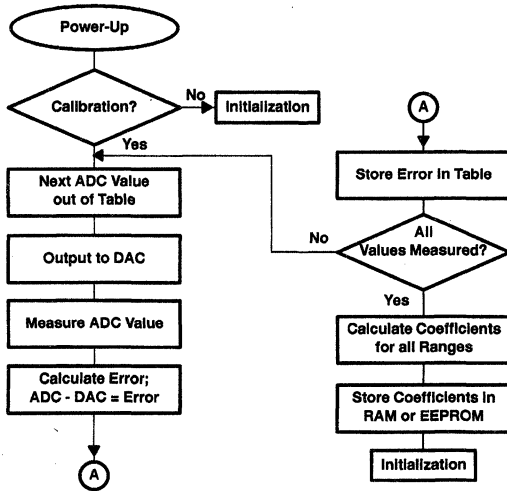
Figure 2. Flowchart 1: Calibration With an External Digital-to-Analog Converter

The measurement sequence for an ADC point is as follows (see also Figure 2):

- The MSP430 outputs via its select lines (parallel DAC) or via an output line (serial DAC) a 14- or 16-bit number. This number programs the DAC. The LCD is not damaged, due to the short duration of the signals (microseconds).

- The external DAC converts the digital number into a precise output voltage that corresponds to the input number.

- The MSP430 measures the output voltage of the DAC and compares the result with the number that was the output. The difference (measured ADC value – output DAC value) is the absolute error of the ADC at that given point.

- The measured errors are used for the calculation of the correction values. These are stored in the RAM or in an EEPROM and are used for the correction of the ADC characteristic. The format and the number of the stored correction values depend on the correction method used: 1 to 64 bytes for the examples given here.
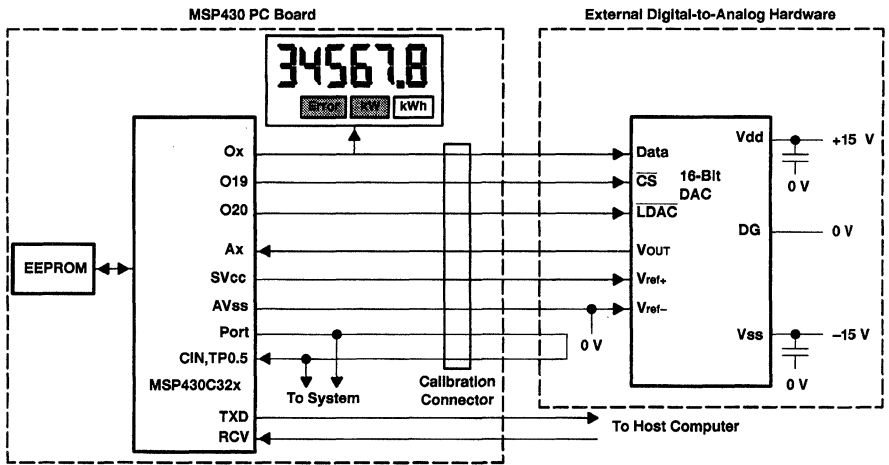
Figure 3. External, Serially Controlled DAC for ADC Measurement

The loop from Port to CIN that is closed by the external hardware indicates to the MSP430 during the initialization that the measurement of the ADC characteristic is active. Like the other DAC control lines, these two I/Os may be used for other system tasks when not in calibration mode.

It is also possible to use a parallel DAC for the calibration of the MSP430 ADC. The time needed for the measurement of the ADC characteristic is shorter than with a serial DAC, but the number of connections between the MSP430 board and the calibration unit are much higher than for a serially controlled DAC. Figure 4 shows this arrangement.

MSP430 PC Board **External Digital-to-Analog Hardware**
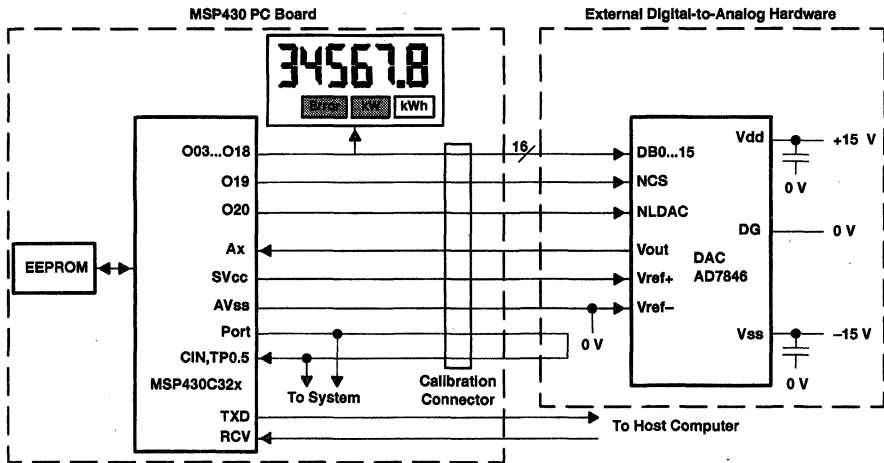
Figure 4. External, Parallel Controlled DAC for ADC Measurement

2.3 External Discrete, Precise Voltages

If only a few points of the ADC characteristic need to be known, then only a few discrete input voltages are necessary for the calibration process. These few points can be generated with a precise, external reference voltage or the supply voltage of the MSP430 and a resistor divider providing some defined output voltages. Figure 5 shows both possibilities.

MSP430 PC Board **External Digital-to-Analog Hardware**
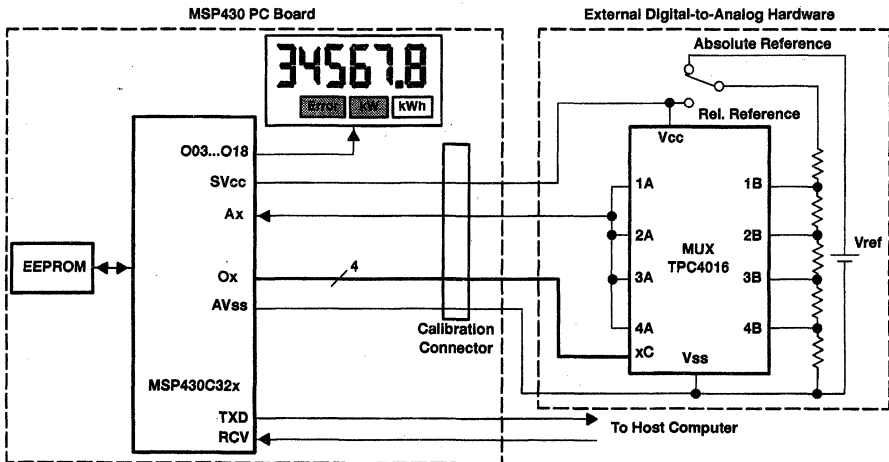
Figure 5. External, Precise Voltages for Calibration

2.4 External Discrete Precision Resistors

If the task for the MSP430 ADC is to precisely measure resistance—for example resistive sensors or platinum—and not external voltages, then this method should be considered. The external hardware is a multiplexer that connects precision resistors to one of the analog inputs of the MSP430. For external resistors with low resistance it may be necessary to use reed relays for this task due to the Rdson resistance of the multiplexer paths. Figure 6 shows this solution for two external reference resistors: the current source outputs the current Ics at the analog input Ax, the voltage drop at the selected external reference resistor is measured with the same analog input. The number of the external precision resistors may be adapted to the application needs.

This calibration method includes all onboard error sources such as Rext and the ADC characteristic.
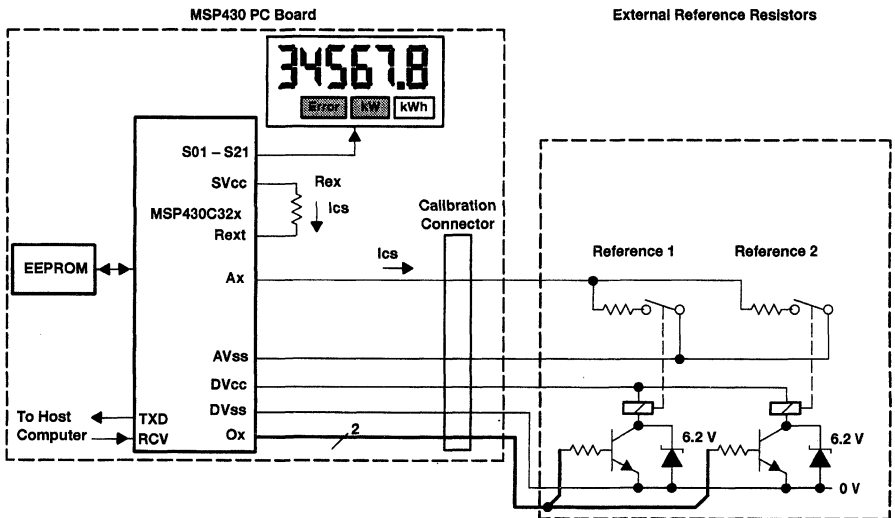
Figure 6. External Precision Resistors for Calibration

2.5 Storage of the Correction Data

The correction coefficients as calculated by the MSP430 or a host computer are stored in the RAM or in an external EEPROM:

- The RAM may be used if a battery is permanently connected to the MSP430 system.

- An EEPROM is necessary if the supply voltage of the MSP430 system can be interrupted e.g. due to the mains supply or a switch.

The format of the used 8-bit coefficients is given in *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*, SLAA050 [4]. If the accuracy that can be reached with these 8-bit numbers is insufficient, then 16-bit numbers—with doubled RAM space and calculation time—may be used. Also the MSP430 floating point package can be a solution in this case.

3 Different Improvement Methods

To allow a comparison between the different improvement methods, the mean value, the range, the standard deviation, and the variance of the corrected ADC characteristic are given. The nearer these values are to zero, the better the performance of the used improvement method.

The mathematical equations for the used statistical methods follow. They are applied to every fourth value of the 16383 corrected samples.

The mean value x is calculated by summing all of the errors (ei) of all corrected samples Ni and dividing this sum by the number of samples k. The mean value $\bar{x}$ is:

$$\bar{x} = \frac{\sum_{i=1}^{i=k} ei}{k}$$

The range R is the difference between the largest error e_{max} and the smallest error e_{min} (e.g. the most negative error value). The range R is defined as:

$$R = e_{max} - e_{min}$$

The standard deviation S is defined as:

$$S = \sqrt{\frac{\sum_{i=1}^{i=k} ei^2 - \frac{\left(\sum_{i=1}^{i=k} ei\right)^2}{k}}{k-1}} = \sqrt{V \times \frac{k}{k-1}}$$

The formula for the variance V is:

$$V = \frac{\sum_{i=1}^{i=k} ei^2 - \frac{\left(\sum_{i=1}^{i=k} ei\right)^2}{k}}{k} = \frac{\sum_{i=1}^{i=k} ei^2 - \bar{x} \times \sum_{i=1}^{i=k} ei}{k}$$

Where:

k   = Number of included ADC errors ei
ei   = ADC error at ADC step i, ranging from e1 to ek   [Steps]
i   = Index for ADC errors

NOTE: Each measured ADC value needs to be corrected individually to get a correct result. If differences are measured (ΔN) then both values have to be corrected and then the subtraction executed. A correction of the difference ΔN alone leads to false results.

It is important to note the different scaling that is used for the y-axis of the graphs with the corrected ADC characteristic. They differ significantly, dependent on the amount of improvement.

The correction coefficients for all improvement methods are calculated in such a way that allows addition for the final correction of the measured ADC result. This saves execution time and program space.

All of the calculations used for the correction are made with a floating point package (like the MSP430 FPP4 software). If—as is necessary in real-time systems—an integer package is used, then small rounding errors will occur. In *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*[4] the software routines and their influence on the accuracy of the final result are explained.

The improvement methods and their results for this report are demonstrated with the characteristic of device 1 due to its worst characteristic compared to the other three devices shown in *Architecture and Function of the MSP430 14-Bit ADC*[1].

The ADC samples used for the following improvement methods and calculations were measured the following way:

- Twelve samples with the same ADC input voltage—generated by a 16-bit DAC—were measured and stored.
- The maximum and the minimum value of these twelve samples were rejected (rejection of extremes).
- Out of the remaining ten samples the mean value was calculated and used afterwards.

The improvement methods are always shown for the full ADC range (ranges A, B, C, and D). If the current source is active, then only ranges A, B, and part of C can be used: the same improvement methods with the same formulas are valid but with less needed RAM or EEPROM space for the correction coefficients. Due to the importance of the current source for several applications, the statistical results are also shown for ranges A and B only.

The 14-bit oriented correction software is also usable if the 12-bit ADC mode is used: only the correction coefficients of the applied ADC range are used in this case.

The orientation of this application report to the ADC ranges (single or multiple corrections per range) is applicable, due to the visibly different slopes of the four ranges inside of an ADC characteristic. See the noncorrected ADC characteristics of device 1 (Figure 7) and devices 2 to 4 in [1] for examples.

3.1 The ADC Characteristic of Device 1 Without Correction

The noncorrected ADC characteristic of device 1 is shown in Figure 7. Its statistical values are given in the table below Figure 7.

The circle in Figure 7 indicates the irregularity located in range B. This irregularity is the reason why more sophisticated methods sometimes have worse results than simpler ones.

Device 1 Uncorrected

ADC Steps [0 to 16383]
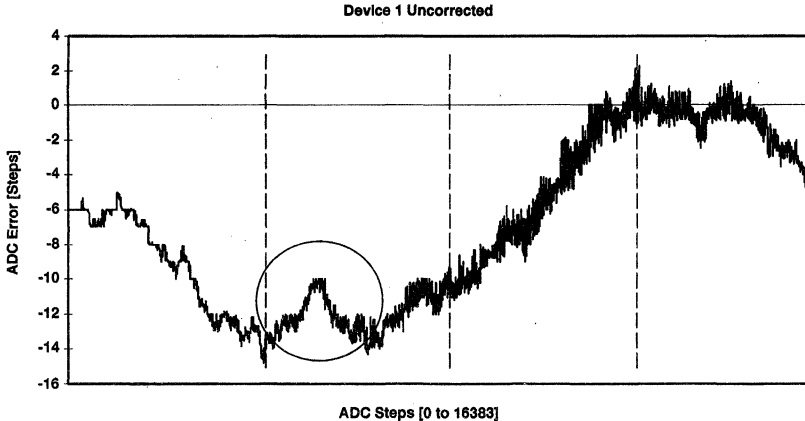
Figure 7. The Noncorrected Characteristic of Device 1

The statistical results of the original ADC characteristic of device 1 are:

	Full range	Ranges A and B only
Mean Value:	−6.95 Steps	−10.51 Steps
Range:	17.00 Steps	10.80 Steps
Standard Deviation:	4.74 Steps	2.61 Steps
Variance:	22.51 Steps	6.80 Steps

3.2 Correction Methods Using Addition Only

These four methods are the fastest because they omit the multiplication. The main disadvantages are the gaps between the ADC ranges e.g. from ADC step 4095 to 4096, and the amount of RAM used, but these methods not only show speed advantages but also the best results. The four methods explained below are best for real-time applications, where the 50 to 100 cycles that are necessary for a correction that uses multiplication cannot be spent: they are the fastest way possible for correction.

3.2.1 Correction With the Mean Value of the Full ADC Range

The ADC is measured at k equally spaced points. The errors of these k measurements are calculated and the mean value of these errors is stored and used for the correction of the ADC. The correction formula for each ADC sample Ni to get the corrected value Nicorr is:

$$Nicorr = Ni + \frac{\sum\limits_{i=1}^{i=k} - ei}{k}$$

Where:

Nicorr	= Corrected ADC sample	[Steps}
Ni	= Measured ADC sample (noncorrected)	[Steps]
k	= Number of included ADC errors ei	
ei	= ADC error i, ranging from e1 to ek	[Steps]

The principle is shown in Figure 8, the full ADC range is corrected with its mean value. As with all future principle figures in this report, the black straight line indicates the correction value, the scribbled black line indicates the noncorrected ADC characteristic, and the white line shows the corrected ADC characteristic. The small circles indicate the measured ADC points (the 128 circles of Figure 8 are not shown).

Device 1
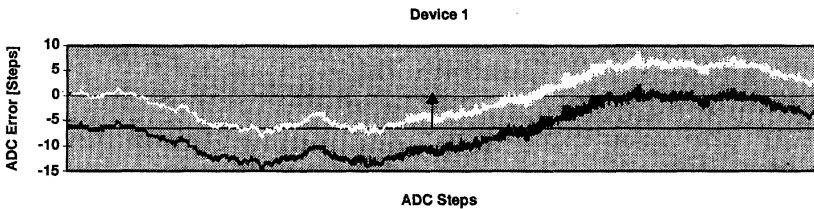
Figure 8. Principle of the Error Correction by the Mean Value of the Full Range

For k = 128—which means 128 samples over the complete ADC range—the statistical results are:

	Full range	Ranges A and B only
Mean Value:	−0.44 Steps	0.15 Steps
Range:	17.10 Steps	10.80 Steps
Standard Deviation:	4.74 Steps	2.61 Steps
Variance:	22.51 Steps	6.80 Steps

Figure 9 shows the result in a graph. The corrected characteristic is displayed for the full range and for the ranges A and B only:

Device 1 Corrected with the Mean Value of the used Range
(Full Range and A and B only)

ADC Steps [0 to 16383]
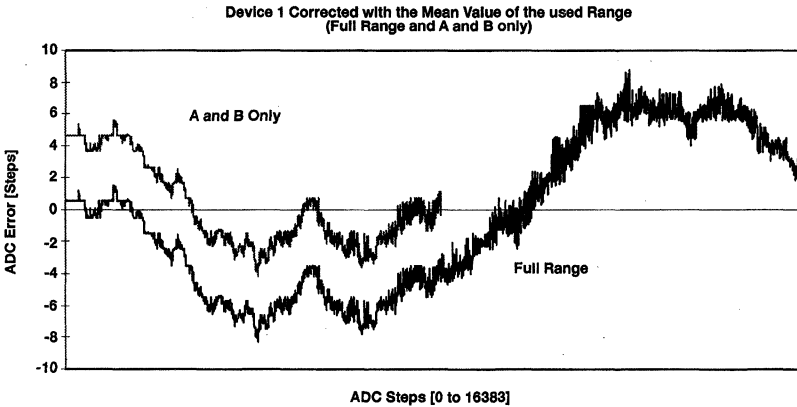
Figure 9. Error Correction With the Mean Value of the Used Range

Advantages: Only one addition is necessary
Very fast due to no missing multiplication or shifts
No gaps; the monotonicity of the ADC characteristic remains
Only one byte of RAM is needed for the correction coefficient

Disadvantages: Range, standard deviation and variance are not improved
Many calibration measurements are necessary

NOTE: Within the software examples, the format of the integer number
is noted at the right margin. The meaning of the different notations is:
    0.7   Zero integer bits, 7 fraction bits. Unsigned number
    ±4.3   Four integer bits, 3 fraction bits. Signed number
    8.0   Eight integer bits, no fraction bits. Unsigned integer number
    ±7.0   Seven integer bits, no fraction bits. Signed integer number

The software part after each ADC measurement is as follows:

```
; Correction with the mean value of the full range. 7 cycles
;
      MOV.B    TAB,R5      ; Correction for full range    ±7.0
      SXT      R5          ; Sign extend byte to word      ±15.0
      ADD      &ADAT,R5    ; Corrected ADC value in R5     14.0
      ...                  ; Proceed with corrected ADC value
;
; The RAM byte TAB contains the correction for the full range:
; the negated mean value
;
      .bss     TAB,1       ; Signed 8-bit number                    ±7.0
```

EXAMPLE: The ADC is measured at nine points (rather than 128 to keep the example under control) and the calculated mean value is used for the correction of the full ADC range. The measured (k+1) errors (for device 1) are shown below. The numbers used for the correction are slightly shaded.

ADC Step	50	2048	4096	6144	8192	10240	12288	14336	16330
Error [Steps]	−6	−8	−13	−13	−10	−5	0	0	−3

Correction:
$$\frac{\sum_{i=1}^{i=k} - ei}{k} = \frac{6 + 8 + 13 + 13 + 10 + 5 - 0 - 0 + 3}{9} = \frac{58}{9} = + 6.5$$

Corrected ADC sample: $Nicorr = Ni + 6.5$ Valid for the Full ADC range

Format: ±7.0 $6.5/2^0 = 6.5 ≈ 07h$

7								0
0	0	0	0	0	1	1	1	1

±6.1 $6.5/2^{-1} = 13 = 0Dh$

7								0
0	0	0	0	1	1	1	0	1
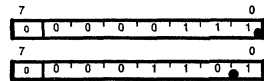
3.2.2 Correction With the Mean Values of the Four Ranges

The ADC is measured at (4×k) equally spaced points. The mean value of the k errors per range is calculated and used individually for the correction of the four ranges A to D. The correction formula for each one of the four ranges is:

$$Nicorr = Ni + \frac{\sum_{i=1}^{i=k} - ei}{k}$$

The principle is shown in Figure 10, each range is corrected with its mean value (the eight used samples are drawn only in the range A):

Device 1 Corrected with the centers of the four ranges

ADC Steps
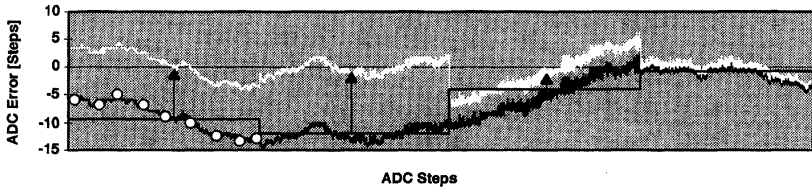
Figure 10. Principle of the Error Correction With the Mean Values of the Four Ranges

For k = 8 (8 samples per range) the statistical results are:

	Full range	Ranges A and B only
Mean Value:	−0.31 Steps	0.15 Steps
Range:	13.5 Steps	9.80 Steps
Standard Deviation:	2.49 Steps	2.10 Steps
Variance:	6.20 Steps	4.41 Steps

Figure 11 shows the graph for k = 8 (eight samples per range, 32 samples over the full ADC range):

Device 1 Corrected with the Mean Values of the Four Ranges
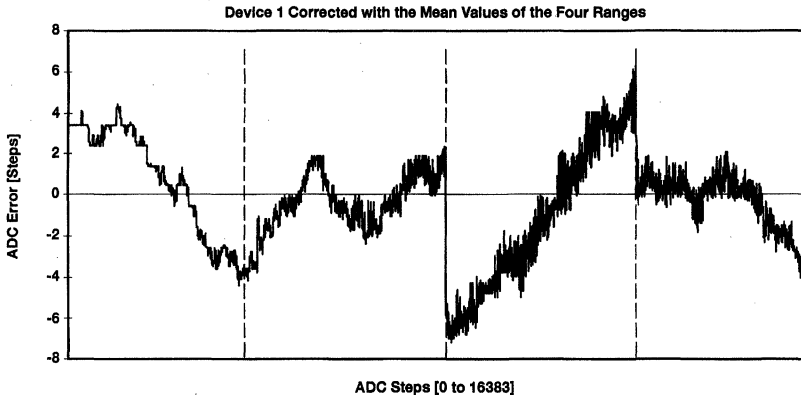
Figure 11. Error Correction With the Mean Values of the Four Ranges

Advantages: Only one addition is necessary for the correction
Fast due to no multiplication
Only four bytes are needed for the storage of the correction values

Disadvantages: Range, standard deviation and variance are only slightly improved
Monotonicity is not preserved: gaps appear at the range borders.

The software part after each ADC measurement is as follows:

```
; Correction with the mean values of the four ranges. 16 cycles
; The four signed correction values are located in four RAM
; bytes starting at label TAB
;
      MOV      &ADAT,R5        ; ADC result Ni to R5 (0...3FFFh)
      MOV      R5,R6           ; Copy result for correction     14.0
      SWPB     R6              ; Range bits of result to low byte
      RRA.B    R6              ; Calc. byte address for corr.   5.0
      RRA.B    R6              ; Shift two range bits to LSBs    4.0
      RRA.B    R6              ;                                 3.0
      RRA.B    R6              ; Range bits now 0 to 3           2.0
      MOV.B    TAB(R6),R6      ; Correction from table TAB      ±7.0
      SXT      R6              ; Signed byte to signed word     ±15.0
      ADD      R6,R5           ; Corrected result Nicorr in R5  14.0
      ...                      ; Proceed with corrected Nicorr  14.0
;
; The four signed correction values are located in four RAM bytes
; starting at label TAB.
;
      .bss     TAB,4           ; Signed 8-bit numbers           ±7.0
```

EXAMPLE: Range A of the ADC is measured at four points and the mean value is used for the correction of this ADC range. The corrections for the other three ranges (B, C and D) are calculated the same way. The measured errors for range A are shown below (for device 1):

ADC Step	1024	2048	3072	4096
Error [Steps]	−6	−8	−12	−13

$$\text{Correction:}\quad \frac{\displaystyle\sum_{i=1}^{i=k} - ei}{k} = \frac{6 + 8 + 12 + 13}{4} = \frac{39}{4} = +9.75$$

Corrected ADC sample: $Nicorr = Ni + 9.75$ Valid for range A

Format: ± 7.0 $9.75/2^0 \approx 10 = 0Ah$

± 6.1 $9.75/2^{-1} = 19.5 \approx 14h$

± 5.2 $9.75/2^{-2} = 39 = 27h$
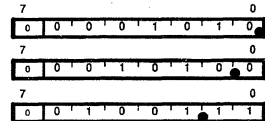
3.2.3 Correction With the Center Points of the Four Ranges

The ADC is measured at the four center points of the ranges A, B, C and D: the ADC steps 2048, 6144, 10240 and 14336. The four errors (ec) at these four center points are calculated and stored. To each measured ADC sample Ni the negated error ec of the pertaining range is added. The correction formula for each one of the four ranges is:

$$Nicorr = Ni + ec$$

Where:
ec = Negated error at the center of the actual ADC range   [Steps]

The principle is shown in Figure 12, the four A/D ranges are corrected individually with the errors of their center points:
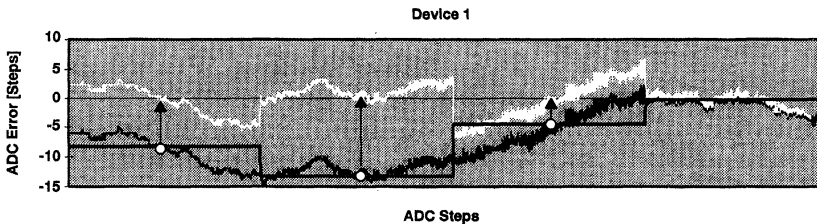
Figure 12. Principle of the Error Correction With the Centers of the Four Ranges

The statistical results of this simple kind of correction are:

	Full range	Ranges A and B only
Mean Value:	0.20 Steps	0.29 Steps
Range:	13.5 Steps	9.80 Steps
Standard Deviation:	2.56 Steps	2.27 Steps
Variance:	6.53 Steps	5.15 Steps

Figure 13 shows the resulting graph:

Device 1

ADC Steps [0 to 16383]
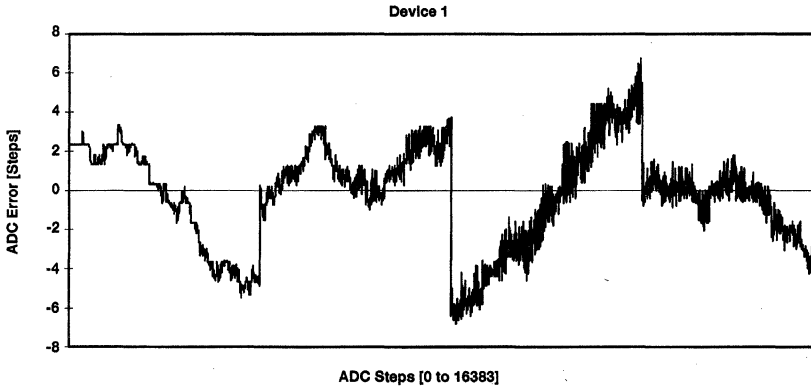
Figure 13. Correction With the Centers of the Four Ranges

Advantages: Only one addition is necessary for the correction
Fast due to no multiplication
Only four bytes are needed for the storage of the correction values

Disadvantages: The range, standard deviation and variance are only slightly improved
Monotonicity is not preserved: gaps appear at the range borders.

The software part after each ADC measurement is the same one as shown for the correction with the mean values of the four ranges.

EXAMPLE: The center point of range C (10240 steps) of the ADC is measured and the result is used for the correction of this ADC range. The other three ranges are treated the same way. The measured errors of the centers of the four ADC ranges are shown below (for device 1):

ADC Step	2048	6144	10240	14336
Error [Steps]	−6	−13	−5	0

Correction: $ec = - (-5) = 5$

Corrected ADC sample: $Nicorr = Ni + 5$ Valid for range C

Format: ±7.0 $5/2^0 = 5 = 05h$
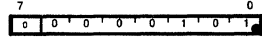
3.2.4 Correction With Multiple Sections

The ADC is measured at (p+1) equally spaced points of the full range of the ADC. This leads to p sections. The resulting errors (ek) are used to calculate the mean value for each section and the result (ekm) is stored. To each ADC sample Ni the appropriate negated error (ekm) is added. This method can be enhanced up to the measurement of all ADC points. The correction formula is:

$$Ni_{corr} = Ni + ekm \qquad ekm = - \frac{ek + 1 + ek}{2}$$

Where:

ekm	= Mean value of the ADC errors at the borders of ADC section ek [Steps}	
k	= Index for ADC sections (length 2^{14}/p), ranging from 0 to p	
p	= Number of sections $(1 \le p < 2^{14})$	
ek	= ADC error at the ADC step Ni = k × 2^{14}/p	[Steps]
ek+1	= ADC error at the ADC step Ni = (k +1) × 2^{14}/p	[Steps]

The principle is shown in Figure 14. The full ADC range is divided into eight sections (p = 8). The nine measured ADC samples are indicated with circles.

Device 1

ADC Steps
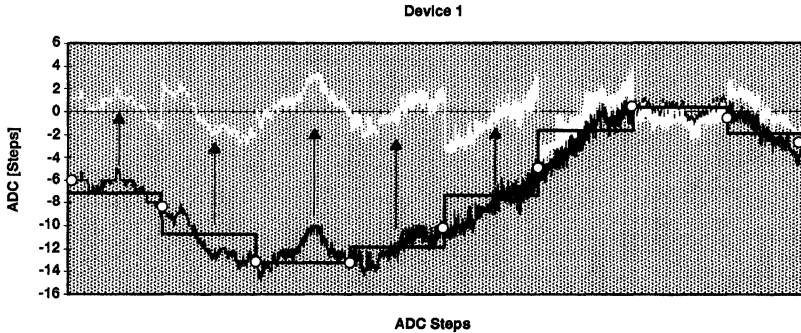
Figure 14. Principle of the Additive Correction With Multiple Sections (8 sections)

For p = 8 (section length is 2048 steps) the statistical results are:

	Full range	Ranges A and B only
Mean Value:	−0.14 Steps	0.22 Steps
Range:	8.40 Steps	6.30 Steps
Standard Deviation:	1.47 Steps	1.37 Steps
Variance:	2.16 Steps	1.89 Steps

Figure 15 shows the resulting graph for an additive correction with 8 sections (p = 8) over the full ADC range:

Device 1 Corrected with Eight Sections over the Full ADC Range
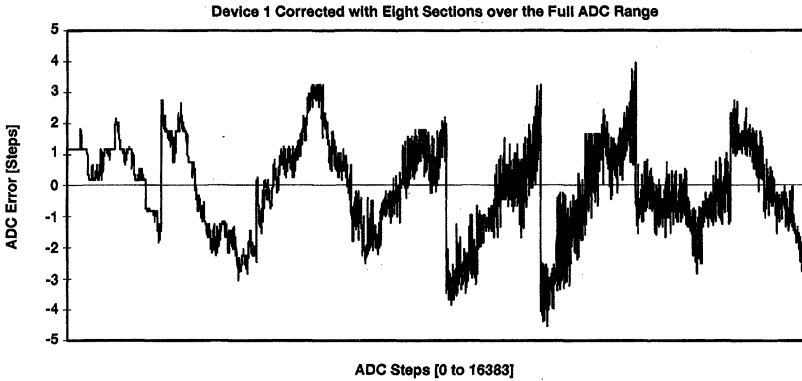
Figure 15. Additive Correction With 8 Sections Over the Full ADC Range

For p = 16 (section length is 1024 steps) the statistical results are:

	Full range	Ranges A and B only
Mean Value:	−0.29 Steps	0.05 Steps
Range:	6.40 Steps	4.85 Steps
Standard Deviation:	1.04 Steps	1.01 Steps
Variance:	1.08 Steps	1.02 Steps

Figure 16 shows the resulting graph for an additive correction with 16 sections (p = 16) over the full ADC range:

Device 1 Corrected with Sixteen Sections Over the Full ADC Range
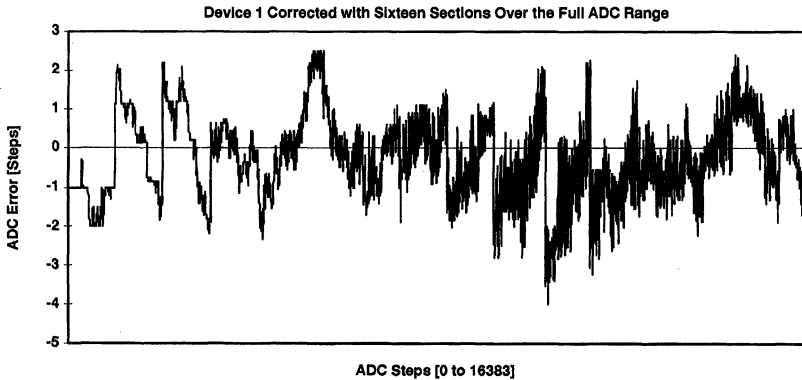
Figure 16. Additive Correction With 16 Sections Over the Full ADC Range

For p = 32 (section length is 512 steps) the statistical results are:

	Full range	Ranges A and B only
Mean Value:	–0.14 Steps	–0.05 Steps
Range:	5.20 Steps	3.65 Steps
Standard Deviation:	0.77 Steps	0.65 Steps
Variance:	0.59 Steps	0.42 Steps

Figure 17 shows the resulting graph for an additive correction with 32 sections (p = 32) over the full ADC range:

Device 1 Corrected with 32 Sections Over the Full ADC Range

ADC Steps [0 to 16383]
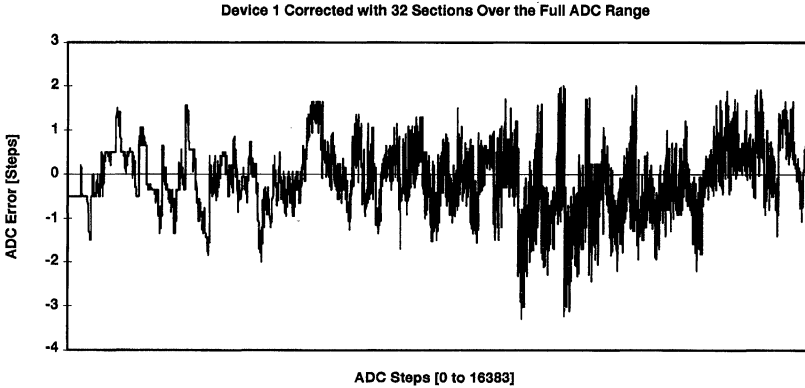
Figure 17. Additive Correction With 32 Sections Over the Full ADC Range

For p = 64 (section length is 256 steps) the statistical results are:

Mean Value:	–0.08 Steps	0.02 Steps
Range:	4.60 Steps	3.10 Steps
Standard Deviation:	0.64 Steps	0.53 Steps
Variance:	0.41 Steps	0.28 Steps

Figure 18 shows the resulting graph for an additive correction with 64 sections (p = 64) over the full ADC range. Note the scaling of Figure 18: only ±3 steps!

Device 1 Corrected with 64 Sections Over the ADC Range

ADC Steps [0 to 16383]
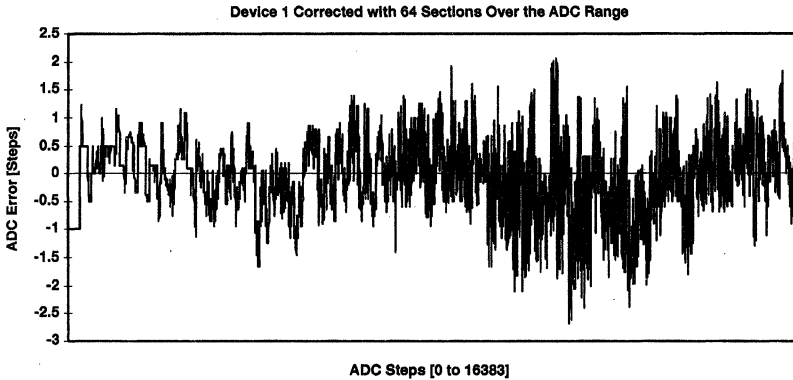
Figure 18. Additive Correction With 64 Sections Over the Full ADC Range

Advantages: Very good improvement with large section counts p
Fast due to no multiplication
The section count p is adaptable to specific applications.

Disadvantages: Relative large RAM storage is needed for a large section count p
Gaps appear at the section borders: they get smaller with increasing p

The results for the additive correction with multiple sections are summarized below for section counts p ranging from 8 to 64. For comparison purposes, the results for p = 4 (the *center of ranges* method is used) are given as well.

	p = 4	p = 8	p = 16	p = 32	p = 64
Mean Value:	+0.2	−0.14	−0.29	−0.14	−0.08 Steps
Range:	13.5	8.40	6.40	5.20	4.60 Steps
Standard Deviation:	2.56	1.47	1.04	0.77	0.64 Steps
Variance:	6.53	2.16	1.08	0.59	0.41 Steps

The improvement of the statistical results with increasing section count p can be clearly seen. Figure 19 illustrates this.
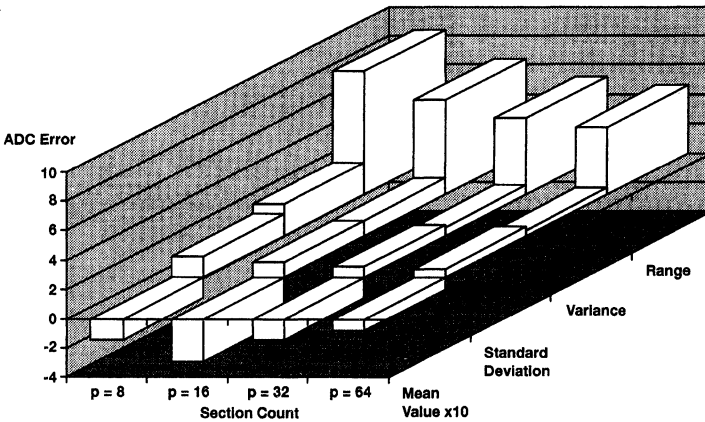
Figure 19. Improvement of the ADC Results With Increasing Section Count p

The software part after each ADC measurement follows. The addressing of the correction byte can be adapted easily also to 4, 8, 16, and 32 sections.

```
; Additive correction for 64 sections over the full ADC range.
; The 64 signed correction values are located in the RAM
; bytes starting at label TAB. 11 cycles
;
        MOV     &ADAT,R5     ; ADC result Ni to R5 (0...3FFFh)
        MOV     R5,R6        ; Copy Ni for correction      14.0
        SWPB    R6           ; MSBs of result to low byte  6.0
        MOV.B   R6,R6        ; 00h...3Fh to R6 (0..63)     6.0
        MOV.B   TAB(R6),R6   ; Corr. eim from table TAB    ±4.0
        SXT     R6           ; Extend sign of correction   ±4.0
        ADD     R6,R5        ; Nicorr = Ni + eim           14.0
        ...                  ; Proceed with corrected Nicorr
;
; The 64 RAM bytes starting at label TAB contain the corrections
; eim for the 64 sections: each one for 256 ADC points.
; The bytes are loaded during initialization Signed 8-bit numbers
;
        .bss    TAB,64       ; 0..255..511....16127..16383 ±4.0
```

EXAMPLE: The ADC is measured at nine points (8 sections) like shown in Figure 14. The measured errors for device 1 are shown below. The correction coefficient ekm of the 2nd section (2048 to 4095 ADC steps, upper half of range A) is calculated.

ADC Step	50	2048	4096	6144	8192	10240	12288	14336	16330
Error [Steps]	−6	−8	−13	−13	−10	−5	0	0	−3

Correction: $\quad ekm = -\dfrac{ek{+}1 + ek}{2} = -\dfrac{-13 - 8}{2} = +10.5$

Corrected ADC sample: $Ni_{corr} = Ni + 10.5$ $\qquad$ Valid for the 2nd section

Format: $\pm7.0 \quad 10.5/2^0 = 10.5 \approx 0Bh$
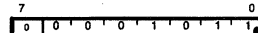
$\qquad\quad \pm6.1 \quad 10.5/2^{-1} = 21 = 15h$
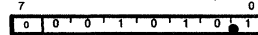
3.2.5 Summary of the Additive Corrections

Figure 20 gives an overview of all of the described additive correction methods. The results are given for different section counts p:

- N.C.: the noncorrected device 1
- p = 1: correction with the mean value of the full ADC range
- p = 2: correction with the mean values of ranges A/B and C/D
- p = 4: correction with the center values of the four Ranges
- p = 8...64: correction with 8 to 64 (multiple) sections over the full ADC range

As can be seen, the improvement increases significantly from the noncorrected device 1 to the additive correction with 64 sections.
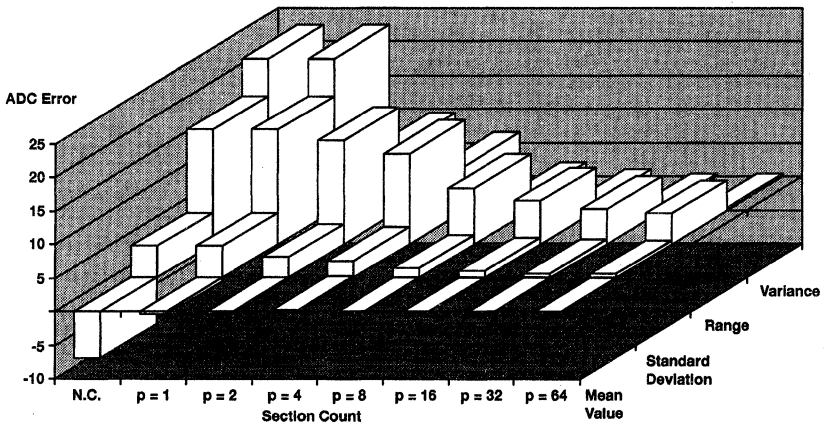
Figure 20. Overview of the Additive Correction Methods

3.3 Additional Information

The *Linear Improvement of the MSP430 14-Bit ADC Characteristic*[3] shows linear methods for the correction of the 14-bit analog-to-digital converter of the MSP430. Different correction methods are explained: some with guaranteed monotonicity and some using linear regression. The methods discussed differ in RAM and ROM allocation, calculation speed, reachable improvement, and complexity.

4 References

1. *Architecture and Function of the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA045

2. *Application Basics for the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA046

3. *Linear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA048

4. *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA050

5. *MSP430 Application Report*, 1998, Literature #SLAAE10C

6. Data Sheet MSP430C325, MSP430P323, 1997, Literature #SLASE06

7. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B

Appendix A Definitions Used With the Application Examples

```
; HARDWARE DEFINITIONS

;
; ADAT       .equ  0118h ; ADC data register (12 or 14-bits)
; ACTL       .equ  0114h ; ADC control register: control bits
```

Linear Improvement of the MSP430 14-Bit ADC Characteristic

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

List of Tables

Linear Improvement of the MSP430 14-Bit ADC Characteristic

Lutz Bierl

ABSTRACT

This application report shows different linear methods to improve the accuracy of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. Different correction methods are explained: some with monotonicity and some using linear regression. The methods used differ in RAM and ROM allocation, calculation speed, reachable improvement, and complexity. For all correction methods, proven, optimized, software examples are given with 8-bit and 16-bit arithmetic. The *References* section at the end of the report lists related application reports in the MSP430 14-bit ADC series.

1   Introduction

The application report *Architecture and Function of the MSP430 14-Bit ADC*[1] gives a detailed overview to the architecture and function of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. The principle of the ADC is explained and software examples are given. Also included are the explanation of the function of all hardware registers contained in the ADC.

The application report *Application Basics for the MSP430 14-Bit ADC*[2] shows several applications of the 14-bit ADC of the MSP430 family. Proven software examples and basic circuitry are shown and explained.

The application report *Additive Improvement of the MSP430 14-Bit ADC Characteristic*[3] explains the external hardware that is needed for the measurement of the characteristic of the MSP430's analog-to-digital converter. This report also demonstrates correction methods that use only addition. This allows the application of these methods in real time systems, were execution time can be critical.

Figure 1 shows the block diagram of the 14-bit analog-to-digital converter of the MSP430 family.

Figure 1. The Hardware of the 14-Bit Analog-to-Digital Converter

The methods for the improvement of the ADC described in the next sections are:

- Linear equations with border fit: single linear equation per range
- Linear equations with border fit: multiple linear equations per range
- Linear equations with linear regression: single linear equation per range
- Linear equations with linear regression: multiple linear equations per range

Quadratic and cubic corrections are explained in the application report *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*[4].

1.1 Correction With Linear Equations

A good error correction with low RAM requirements is possible if not only the offset error—like with the additive methods—but also the slope error of the ADC characteristic can be corrected. However, this requires the use of a multiplication. The multiplication subroutine used here is is optimized for real time environments: it terminates immediately after the unsigned operand—the ADC result—becomes zero due to the right shift during the multiplication. The subroutine is appended to the first software example (see section 1.2.1.1). The full code with explanations and timing is contained in *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*, SLAA050[4].

The generic correction formula for linear correction, which is valid for floating point or 16-bit integer arithmetic, is:

$$Ni_{corr} = Ni + (Ni \times a1 + a0)$$

The optimized 16-bit multiplication subroutine for the above formula—including the calculation software—is included in section 1.2.2.1, *Linear Regression: Single Linear Equation per Range*. The full code is described in Section 5.1, *Integer Calculation Subroutines*.

The floating point example given for the cubic correction—see *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*[4]—may be adapted easily to the calculation of linear equations: the unused terms—the quadratic and cubic terms—are simply left out.

The formulas to calculate the correction coefficients a1 (slope) and a0 (offset) out of the two known errors e2 and e1 of the ADC steps N2 and N1 are:

$$a1 = -\frac{e2 - e1}{N2 - N1} \qquad a0 = -\frac{e1 \times N2 - e2 \times N1}{N2 - N1}$$

The advantages of the negated correction coefficients a1 and a0 are:

- Shorter and faster software: the INV (invert) and INC (increment) instructions for the negation of the corrections are not necessary
- The ADAT register (ADC result register) is a read-only register and can be used for additions directly. If the correction needs to be subtracted from the ADAT register, then an intermediate step is necessary.

All principle figures of this report—as in *Additive Improvement of the MSP430 14-Bit ADC Characteristic*[3]—have the same structure:

- The black straight line indicates the negated correction value (this is to show the precision of the correction).
- The scribbled black line indicates the noncorrected ADC characteristic.
- The white line shows the corrected ADC characteristic.
- The small circles indicate the measured ADC points (not all measured samples are shown).

An example using the 16-bit arithmetic is given in section 1.2.2.1, *Linear Regression: Single Equation per Range*.

All other given equations in the following sections assume the use of the 8-bit arithmetic as described in *Nonlinear Improvement of the MSP430 14-Bit ADC Characterisitc*,SLAA050[4]. Therefore the correction formulas are adapted to the limited, but fast 8-bit arithmetic. This reduced arithmetic makes relative addresses for the ADC steps necessary: the full ADC range is divided into sections and the ADC value is adapted to 128 subdivisions for the full section. The equations for the 8-bit arithmetic are given and explained with each method.

1.2 Coefficients Estimation

With the maximum possible ADC error (±10 steps contained in a band of ±20 steps) the maximum values for the coefficients a1 and a0 are:

Table 1. Worst Case Coefficients With 8-Bit Arithmetic

EQUATIONS PER RANGE	SINGLE	TWO	FOUR
Linear coefficient a1:	±0.15625	±0.078125	±0.0390625
Format a1 (integer.fraction)	0.9	0.10	0.11
Constant coefficient a0:	±20.00	±20.00	±20.00
Format a0 (integer.fraction)	5.2	5.2	5.2
Sections (Equations) per Range	1	2	4
Subdivisions per Range	128	2 × 128	4 × 128
Maximum Change within Section	±20 Steps	±10 Steps	±5 Steps

The above maximum coefficients occur for a single equation per range when the ADC error changes 20 steps within an ADC range (4096 steps) e.g. from +10 to −10 steps or vice versa. For two and four equations per range, the maximum change is appropriately smaller (±10 resp. ±5 steps). This leads to smaller coefficients a1.

- The 8-bit arithmetic operates with signed 8-bit coefficients and an ADC result that is adapted to a value ranging from 0 to 127.
- The 16-bit arithmetic uses the full ADC result (0 to 16383) and signed 16-bit numbers for the calculations.
- The floating point calculation also uses the full ADC result (0 to 16383) and a 32-bit number format for the calculations.

NOTE: Within the software examples, at the right margin of the source code the format of the numbers is noted. The meaning of the different notations is:

0.7   No integer bits, 7 fraction bits. Unsigned number
±4.3   Four integer bits, 3 fraction bits. Signed number
8.0   Eight integer bits, no fraction bits. Unsigned integer number
±7.0   Seven integer bits maximum, no fraction bits. Signed integer number

The statistical results are given separately for the full ADC range (ranges A to D) and for the ranges A and B only. The reason for the second case is the internal current source that is used by many applications: with its use the ADC ranges are restricted to the ranges A, B, and the lower part of range C.

1.2.1 Linear Equations With Border Fit

If monotonicity of the corrected ADC characteristic is a requirement, then the correction methods using the border fit are the right choice. They guarantee, that the four ranges continue smoothly at its borders. This feature is important if the differences of two ADC results are used for calculations.

1.2.1.1 Single Linear Equation per Range

The ADC is measured at the five borders of the four ADC ranges (Ni = 50, 4096, 8192, 12288, 16330). These five results are used for the calculation of the offsets and the slopes of all four ADC ranges.

NOTE: The ADC points 0 and 16383 (3FFFh) including small bands cannot be measured. This is the reason for the use of steps 50 and 16330 in the above explanation.

The formula for the offset a0 and the slope a1 for each one of the four ranges is:

$$Nicorr = Ni + \left[\left(\frac{Ni}{4096} - n \right) \times 128 \times a_1 + a_0 \right]$$

$$a_1 = -\frac{eu - el}{128} \qquad a_0 = -el$$

Where:

Nicorr	= Corrected ADC sample	[Steps}
Ni	= Measured ADC sample (noncorrected)	[Steps]
n	= Range number (0...3 for ranges A...D)	
a1	= Slope of the correction	
a0	= Offset of the correction	[Steps]
eu	= Error of the ADC at the upper border of the range	[Steps]
el	= Error of the ADC at the lower border of the range	[Steps]

The term $\left(\frac{Ni}{4096} - n \right) \times 128$ of the equation above is the adaptation of a complete section—here a full range—to 128 subdivisions. The calculation of the term is made by simple shifts and logical AND instructions and not a division and a multiplication. See the initialization part of the software example.

The principle of the correction with a linear equation for each range (border fit) is shown in Figure 2. Border fit means, that the borders of the four ranges A to D fit together without gaps from one range to the other one: the border value is used for both ranges.

The improvement methods and their results for this report are demonstrated with the characteristic of device 1 and device 2 due to their worst characteristic compared to the other three devices shown in *Architecture and Function of the MSP430 14-Bit ADC*.

Device 1
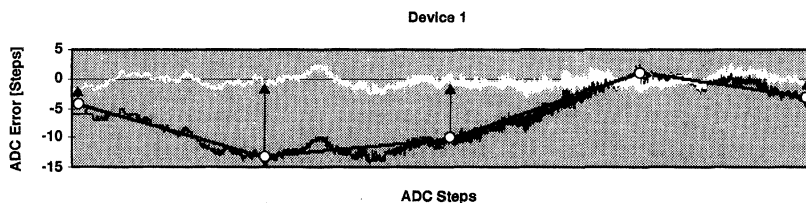
Figure 2. Principle of the Correction With Border Fit (single linear equation per range)

The statistical results for this simple correction method are:

	Full range	Ranges A and B only
Mean Value:	−0.32 Steps	−0.33 Steps
Range:	5.6 Steps	5.1 Steps
Standard Deviation:	0.94 Steps	0.99 Steps
Variance:	0.88 Steps	0.98 Steps

Figure 3 shows the results of this method in a graph.

Device 1 Corrected With a Single Linear Equation per Range (Border Fit)

ADC Steps [0 to 16383]
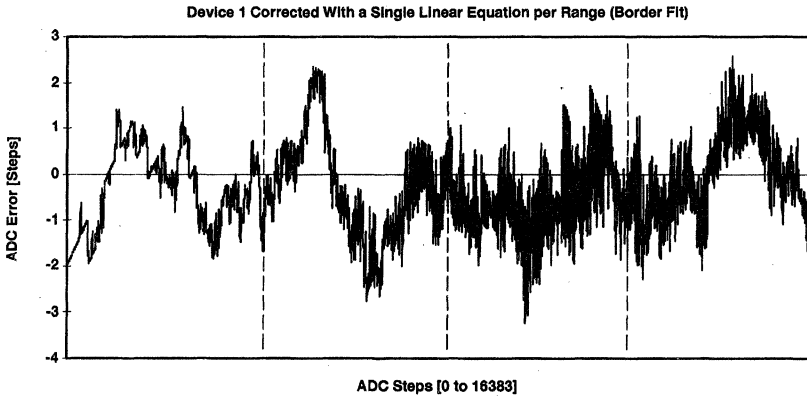
Figure 3. Error Correction With Border Fit (single linear equation)

Advantages: Only five measurements are necessary
No gaps; the monotonicity of the ADC characteristic remains
Low memory needs: 8 bytes only (four slopes and four offsets)
Good improvement of the ADC characteristic despite low expense

Disadvantages: Multiplication is necessary

The software part after each ADC measurement is as follows. For lower accuracy needs the algorithm may be simplified by the use of less accurate slopes and offsets (fewer fraction bits).

A more detailed description for the 8-bit multiplication is given in *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*.[4] The numbers at the right margin indicate the used number format (integer.fraction)

```
; Error correction with a single equation per range
; 8-bit arithmetic. Cycles needed:
; Subdivision = 0:    51 cycles
; Subdivision > 3Fh: 100 cycles
;
;                                                          int.frct
        MOV     &ADAT,R5        ; ADC result Ni to R5          14.0
        MOV     R5,R6           ; Address info for correction  14.0
```

```
        AND    #0FFFh,R5          ; Delete range bits              12.0
        RLA    R5                 ; Calculate subdivision          13.0
        RLA    R5                 ; Prepare (Ni/4096-n)x128        14.0
        RLA    R5                 ; 7 bit ADC info to high byte    15.0
        SWPB   R5                 ; ADC info to low byte 0...7Fh    7.0
        MOV.B  R5,IROP1           ; To MPY operand register         7.0
        SWPB   R6                 ; MSBs to low byte 0...3Fh        6.0
        RRA.B  R6                 ; Calculate coeff. address        5.0
        RRA.B  R6                                                   4.0
        RRA.B  R6                 ; 2n (Range) in R6  0...07h       3.0
        BIC    #1,R6              ; 0...06h: address of slope a1    3.0
        MOV.B  TAB1(R6),IROP2L    ; Slope a1                        0.9
        CALL   #MPYS8             ; (Ni/4096-n)x 128 x a1          ±5.9
        RLA    IRACL              ; Slope part to a0 format        ±5.10
        SWPB   IRACL              ;                                ±5.2
        SXT    IRACL              ; To 16-bit format               ±5.2
        MOV.B  TAB0(R6),R5        ; Offset a0                      ±5.2
        SXT    R5                 ; To 16-bit format
        ADD    R5,IRACL           ; Ni + correction                ±5.2
        RRA    IRACL              ;                                ±5.1
        RRA    IRACL              ; Carry is used for rounding     ±5.0
        ADDC   &ADAT,IRACL        ; Corrected result Nicorr        14.0
        ...                       ; Use Nicorr in IRACL
;
; The 8 RAM bytes starting at label TAB1 contain the correction
; info a1 and a0. The bytes are loaded during the calibration
;
        .bss   TAB1,1             ; Range A a1: lin. coefficient   ±0.9
        .bss   TAB0,1             ; a0: constant coefficient       ±5.2
        .bss   TABx,6             ; Ranges B, C, D: a1, a0. (like above)
; Run time optimized 8-bit Multiplication Subroutines
;
IROP1  .EQU   R14                 ; Unsigned ADC result (7Fh max.)
IROP2L .EQU   R13                 ; Signed factor (80h...7Fh)
IRACL  .EQU   R12                 ; Signed result word
;
; Signed multiply subroutine: IROP1 x IROP2L -> IRACL
;
MPYS8  CLR IRACL                  ; 0 -> 16 bit RESULT
       TST.B  IROP2L              ; Sign of factor (slope a1)
       JGE    MACU8               ; Positive sign: proceed
       SWPB   IROP1               ; Negative
       SUB    IROP1,IRACL         ; Correct result
       SWPB   IROP1
```

```
;
MACU8   BIT.B   #1,IROP1            ; Test actual bit (LSB)
        JZ      L$01               ; If 0: do nothing
        ADD     IROP2L,IRACL       ; If 1: add multiplier to result
L$01    RLA     IROP2L             ; Double multiplier IROP2
        RRC.B   IROP1              ; Next bit of IROP1 to LSB
        JNZ     MACU8              ; If IROP1 = 0: finished
        RET
```

EXAMPLE: The ADC is measured at the five borders of the ADC ranges. The measured errors—device 1 is used—are shown below. The correction coefficients for the range C are calculated. The correction coefficients for the other three ranges may be calculated the same way, using the appropriate border errors.

ADC Step	50	4096	8192	12288	16330
Error [Steps]	–6	–13	–10	0	–3

Error coefficients for the range C:

$$a1 = -\frac{eu - el}{128} = -\frac{0 - (-10)}{128} = -\frac{10}{128} = -0.078125$$

$$a0 = -el = -(-10) = +10$$

Correction: $\left(\dfrac{Ni}{4096} - n\right) \times 128 \times a1 + a0 = \left(\dfrac{Ni}{4096} - 2\right) \times 128 \times (-0.078125) + 10.0$

The correction for the ADC step 11000—located in range C—is calculated:

$$\left(\frac{11000}{4096} - 2\right) \times 128 \times (-0.078125) + 10.0 = +3.1$$

Corrected ADC sample: $Nicorr = Ni + 3.1$ Valid for the ADC step 11000

Format: a1: ±0.9   –0.078125/2⁻⁹ = –40 = D8h

a0: ±5.2   +10.0/2⁻² = +40 = 28h

Format: a1: ±0.9 $-0.078125/2^{-9} = -40 = $ D8h
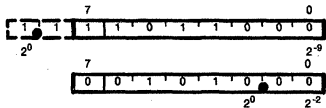
a0: ±5.2 $+10.0/2^{-2} = +40 = $ 28h

The number of fractional bits for a1 is derived from the following consideration: a1 is maximally ±0.15625 (see Table 1). This value must be possible with the largest number that can be expressed with a signed 8-bit number (7Fh):

$$7Fh \times 2^{-9} > 0.15625 = \frac{20}{128} > 7Fh \times 2^{-10}$$

$$0.24805 > 0.15625 = \frac{20}{128} > 0.124025$$

This leads to a valency of 2^{-9} for the LSB of the 8-bit number. The detailed explanation for the calculation of the correction coefficients is given in *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*,[4] *Calculation of the 8-Bit Numbers*.

1.2.1.2 Multiple Linear Equations per Range

The ADC is measured at (p+1) equally distributed points over the full ADC range ($p = 2^m \geq 8$). These (p+1) results are used for the calculation of the offset and the slope of p linear equations valid for the p sections. The formula for the offset a0 and the slope a1 for each of the p linear equations is (8-bit arithmetic):

$$Ni_{corr} = Ni + \left[\left(\frac{Ni \times p}{2^{14}} - n1 \right) \times 128 \times a1 + a0 \right]$$

$$a1 = -\frac{(eu-el)}{128} \qquad a0 = -el$$

Where:

Ni_{corr}	= Corrected ADC sample	[Steps}
Ni	= Measured ADC sample (noncorrected)	[Steps]
$n1$	= Value of the MSBs of Ni (0 to p–1)	
p	= Number of sections over the full ADC range (8 for Figure 4)	
$a1$	= Slope of the correction	
$a0$	= Offset of the correction	[Steps]
eu	= Error of the ADC at the upper border of the section	[Steps]
el	= Error of the ADC at the lower border of the section	[Steps]

n1 ranges from 0 to (p–1) and has a length of $log_2\ p$ bits. This means for n1:

- Two linear equations per range (Figure 4): value is 0...7, length is $log_2\ 8 = 3$ bits;
- Four linear equations per range (Figure 6): value is 0...15, length is $log_2\ 16 = 4$ bits;

The term $\left(\frac{Ni \times p}{2^{14}} - m1 \right) \times 128$ in the equation above is the adaptation of a complete section—here a half range—to 128 subdivisions. The calculation is made by simple shifts and logical AND instructions

1.2.1.3 Two Linear Equations per Range

The principle for two linear equations per range (p=8) is shown in Figure 4.

Device 1 Corrected With Eight Linear Equations (Border Fit)
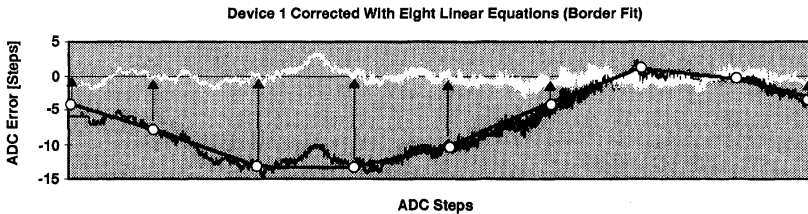
Figure 4. Principle of the Correction With Border Fit (two linear equations per range)

The statistical results for two linear equations per range are:

	Full range	Ranges A and B only
Mean Value:	–0.29 Steps	–0.02 Steps
Range:	6.49 Steps	5.3 Steps
Standard Deviation:	0.97 Steps	1.06 Steps
Variance:	0.94 Steps	1.12 Steps

Figure 5 shows the result in a graph.

Device 1 Corrected With Eight Linear Equations (Border Fit)
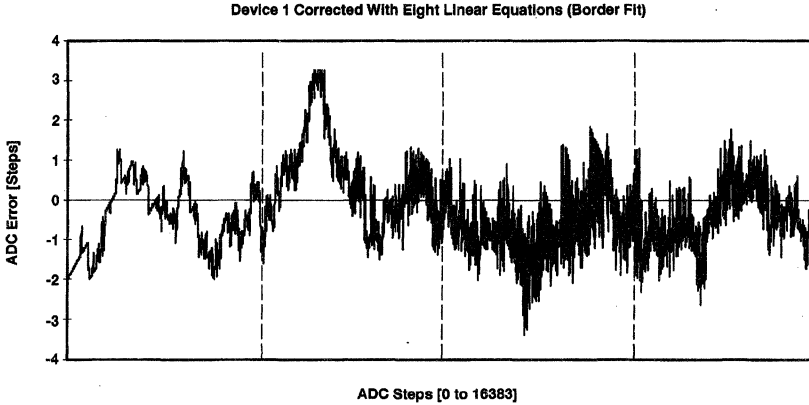
Figure 5. Error Correction With Border Fit (two linear equations per range)

Advantages: Only few measurements are necessary (p+1). Nine for the example above
No gaps; the monotonicity of the ADC characteristic is preserved
Better correction than with a single linear equation per range
Low memory needs: 2 × p bytes (16 for the example)

Disadvantages: Multiplication is necessary

The software is the same as shown in section 1.2.2.2, *Multiple Linear Equations per Range*.

EXAMPLE: The ADC is corrected with eight sections, each one with a length of 2048 steps (p = 8). The measured errors—(device 1 of *Architecture and Function of the MSP430 14-Bit ADC*, SLAA045 is used)—are shown below. The correction coefficients for the lower section of range C—ADC steps 8192 to 10240 (n1 = 4)—are calculated. The correction coefficients for the other seven sections are calculated the same way.

ADC Step	50	2048	4096	6144	8192	10240	12288	14336	16330
n1	0	1	2	3	4	5	6	7	7
Error [Steps]	–6	–8	–13	–13	–10	–5	0	0	–3

Error coefficients for the lower section of range C:

$$a_1 = -\frac{e_u - e_l}{128} = -\frac{-5 - (-10)}{128} = -\frac{5}{128} = -0.0390625$$

$$a_0 = -e_l = -(-10) = +10$$

Correction: $\left(\dfrac{N_i \times p}{2^{14}} - n_1\right) \times 128 \times a_1 + a_0 = \left(\dfrac{N_i \times 8}{2^{14}} - 4\right) \times 128 \times (-0.03906) + 10.0$

Lower section of range C

The correction for the ADC step 9000—located in the lower section of range C—is calculated ($p = 8$, $n_1 = 4$):

$$\left(\frac{9000 \times 8}{2^{14}} - 4\right) \times 128 \times (-0.0390625) + 10.0 = 50.5 \times (-0.0390625) + 10.0 = +8.027$$

Corrected ADC sample: $N_{icorr} = N_i + 8.03$ Valid for the ADC step 9000 (range C)

Format: a_1: ±0.10 $-0.039625/2^{-10} = -40 = D8h$

       a_0: ±5.2 $+10.0/2^{-2} = 40 = 28h$
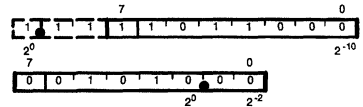
1.2.1.4 Four Linear Equations per Range

The principle for four linear equations per range ($p=16$) is shown in Figure 6.

Device 1

ADC Steps
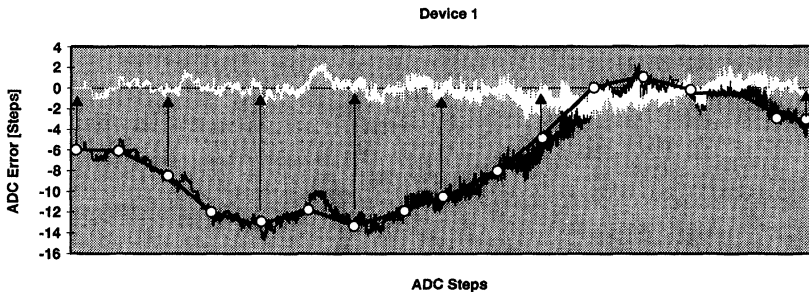
Figure 6. Principle of the Correction With Border Fit (four linear equations per range)

The statistical results for four linear equations per range are:

	Full range	Ranges A and B only
Mean Value:	−0.22 Steps	0.07 Steps
Range:	5.36 Steps	4.07 Steps
Standard Deviation:	0.83 Steps	0.65 Steps
Variance:	0.69 Steps	0.42 Steps

Figure 7 shows the result in a graph.

Device 1 Corrected With Sixteen Equations (Border Fit)
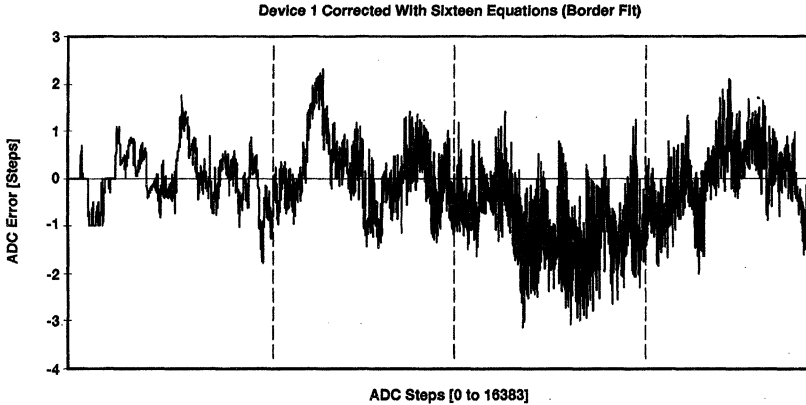
Figure 7. Error Correction With Border Fit (four linear equations per range)

Advantages: Only a few measurements are necessary (p+1). Seventeen for the example above

No gaps; the monotonicity of the ADC characteristic is preserved

Better correction than with one or two linear equations per range

Low memory requirements if p is small: $2 \times p$ bytes (32 for above example)

Disadvantages: Multiplication is necessary

For 16 linear equations for the full ADC range, the software for each ADC measurement is as follows:

```
; Error correction with four linear equations per range
; (16 for the full ADC range) 8-bit arithmetic. Cycles needed:
; Subdivision = 0:    49 cycles
; Subdivision > 3Fh: 101 cycles
;
      MOV     &ADAT,R5        ; ADC result Ni to R5          4.0
      MOV     R5,R6           ; Address info for correction
      AND     #03FFh,R5       ; Delete 4 MSBs (n1 bits)      10.0
      RLA     R5              ; Calculate subdivision        11.0
      RLA     R5              ;                              12.0
      RLA     R5              ; Prepare                      13.0
      RLA     R5              ; ((Ni x p/2^14)-n1)x 128      14.0
      RLA     R5              ; 7 bit ADC info to high byte   15.0
      SWPB    R5              ; ADC info to low byte 0...7Fh  7.0
      MOV.B   R5,IROP1        ; To MPY operand register       7.0
```

```
      SWPB    R6                  ; Calculate coeff. address       6.0
      RRA.B   R6                  ; 2 x n1 in R6  0...01Fh         5.0
      BIC     #1,R6               ; 0...01Eh: address of slope a1  5.0
      MOV.B   TAB1(R6),IROP2L     ; Slope a1                       ±0.11
      CALL    #MPYS8              ; ((Ni x p/2^14)-n1)x 128 x a1   ±3.11
      RRA     IRACL               ; MPY result to a0 format        ±3.10
      SWPB    IRACL               ;                                ±3.2
      ADD.B   TAB0(R6),IRACL      ; Offset a0                      ±5.2
      SXT     IRACL               ;                                ±5.2
      RRA     IRACL               ;                                ±5.1
      RRA     IRACL               ; Carry is used for rounding     ±5.0
      ADDC    &ADAT,IRACL         ; Corrected result Nicorr        14.0
      ...                         ; Proceed with Nicorr in IRACL
;
; The 32 RAM bytes starting at label TAB1 contain the corr.
; coefficients a1 and a0. The bytes are loaded during the
; initialization. 8-bit, signed numbers
;
      .bss    TAB1,1              ; Range A lowest quarter: a1 ±0.11
      .bss    TAB0,1              ;                         a0 ±5.2
      .bss    TABx,30             ; Ranges A (3), B, C, D: a1, a0.
```

EXAMPLE: The ADC is corrected with sixteen sections, each one with a length of 1024 steps (p = 16). The measured errors (device 1 of *Architeture and Function of the MSP430 14-Bit ADC*, SLAA045 is used.) are shown below. The correction coefficients for the lowest section of range C—ADC steps 8192 to 9216 (n1 = 8)—are calculated. The correction coefficients for the other seven sections are calculated the same way.

ADC Step	8192	9216	10240
n1	8	9	10
Error [Steps]	–10	–7	–5

Error coefficients for the lower section of range C:

$$a1 = -\frac{eu - el}{128} = -\frac{-7 - (-10)}{128} = -\frac{3}{128} = -0.0234375$$

$$a0 = -el = -(-10) = +10$$

Correction:
$$\left(\frac{Ni \times p}{2^{14}} - n1\right) \times 128 \times a0 + a1 = \left(\frac{Ni \times 16}{2^{14}} - 8\right) \times 128 \times (-0.0234375) + 10.0$$

Lower section of range C

The correction for the ADC step 9000—located in the lower section of range C—is calculated ($p = 16$, $n_1 = 8$):

$$\left(\frac{9000 \times 16}{2^{14}} - 8 \right) \times 128 \times (-0.0234375) + 10.0 = 101.0 \times (-0.0234375) + 10.0 = +7.63$$

Corrected ADC sample: $Nicorr = Ni + 7.63$ Valid for the ADC step 9000 (range C)

Format: a_1: ±0.11 $-0.0234375/2^{-11} = -48 - D0h$
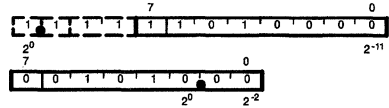
a_0: ±5.2 $+10.0/2^{-2} = 40 = 28h$

1.2.2 Linear Equations With Linear Regression

With linear regression the linear equations that best fit the measured ADC characteristic are used. This leads to good results within the ranges but may produce gaps at the borders.

The linear regression formulas (*Least Squares Method*) for the correction coefficients a1 (slope) and a0 (offset) are given below. To simplify the real time calculations, the negative values of the coefficients are used. The reasons for this are the same ones as described in section 1.1.

$$a_1 = - \frac{\dfrac{\sum\limits_{i=1}^{i=k} N \times \sum\limits_{i=1}^{i=k} ei}{k} - \sum\limits_{i=1}^{i=k} N \times ei}{\dfrac{\left(\sum\limits_{i=1}^{i=k} N\right)^2}{k} - \sum\limits_{i=1}^{i=k} N^2} \qquad a_0 = - \left(\overline{e} - a_1 \times \overline{N} \right)$$

The mean values of N and e are defined as:

$$\overline{N} = \frac{\sum\limits_{i=1}^{i=k} N}{k} \qquad \overline{e} = \frac{\sum\limits_{i=1}^{i=k} ei}{k}$$

Where:

N	=	Measured ADC sample (noncorrected)	[Steps]
ei	=	Error of the ADC sample i	[Steps]
a1	=	Slope of the correction (negated)	
a0	=	Offset of the correction (negated)	[Steps]
k	=	Number of the measured samples	
i	=	Sample index running from 1 to k	

The value N represents different values depending on the calculation method:
- 8-bit arithmetic: the subdivision of Ni within the appropriate section. The range for N is 0...127. See the explanation given in section 1.2.1.1
- Floating Point and 16-bit arithmetic: N equals the full 14-bit ADC value Ni

The examples used are simplified due to the amount of data involved.

1.2.2.1 Single Linear Equation per Range

The ADC is measured at k points inside each of the four ranges. Out of these (4×k) results, four linear equations are calculated using the *Least Squares Method* (see above formulas). The four slopes and offsets are stored in the RAM or in EEPROM. The formula for the corrected value Nicorr is:

$$Nicorr = Ni + \left[\left(\frac{Ni}{4096} - n \right) \times 128 \times a_1 + a_0 \right]$$

Where:

n	= Range number (0...3) for ADC ranges A...D)	
a_1	= Slope calculated by the host or MSP430	
a_0	= Offset calculated by the host or MSP430	[Steps]
k	= Number of samples for each linear equation (range)	

The term $\left(\frac{Ni}{4096} - n \right) \times 128$ of the above equation is the adaptation of a complete section—here a full range—to 128 subdivisions. The calculation is made by simple shifts and logical AND instructions. See the initialization part of the example below.

The principle of this method is shown in Figure 8, the eight measured samples are drawn only in range A (k = 8):

Device 1
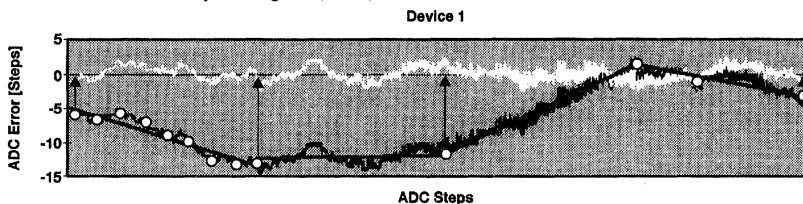
Figure 8. Principle of the Linear Regression Method (single linear equation per range)

	Full range	Ranges A and B only
Mean Value:	0.03 Steps	0.12 Steps
Range:	5.09 Steps	4.85 Steps
Standard Deviation:	0.94 Steps	1.00 Steps
Variance:	0.88 Steps	1.00 Steps

The statistical results for 8 and 16 measurements per range are shown below: as it can be seen, 16 samples per range improve the final result only marginally.

	8 Samples per range	16 Samples per Range
Mean Value:	0.03 Steps	0.07 Steps
Range:	5.09 Steps	5.04 Steps
Standard Deviation:	0.94 Steps	0.92 Steps
Variance:	0.88 Steps	0.85 Steps

Figure 9 shows the result of this method: eight samples per range are measured (k=8). Note the small range of only ±3 steps.
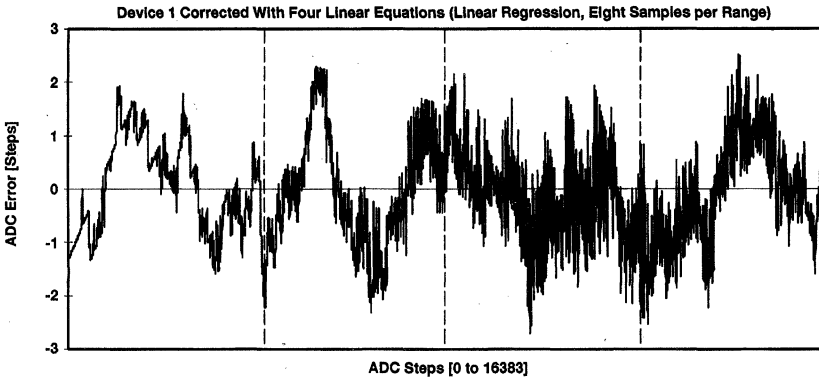
Figure 9. Error Correction With Linear Regression (single linear equation per range)

Advantages: Good adaptation to the ADC characteristic

Disadvantages: One multiplication is necessary
Small gaps at the borders of the four ranges
Calculation of the linear regression is necessary during the calibration

Device 1 does not show gaps at the borders of the four ranges—which is purely random—therefore another device that shows this disadvantage of the method more clearly is included in Figure 10. Note the gaps between the ranges A and B and the ranges B and C.
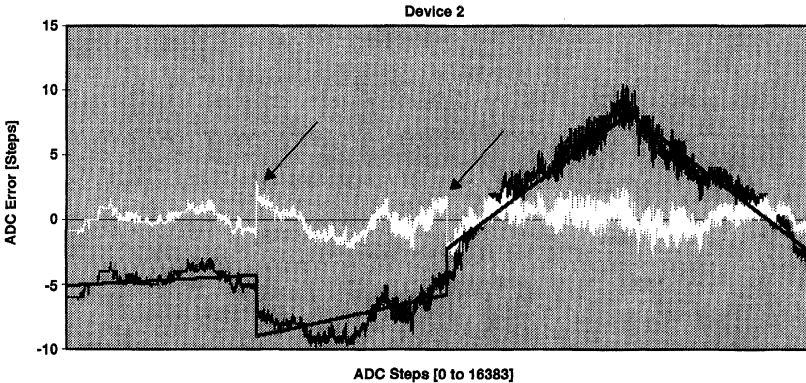
Figure 10. Device 2[1] Showing the Typical Gaps at the Range Borders

---

[1] This is device 2 from *Archtecture and Function of the MSP430 14-Bit ADC Application Report #SLAA045.*

The correction software for the 8-bit arithmetic is identical to the one shown in section *Single Linear Equation per Range (Border Fit)*, section 1.2.1.1.

Here an additional solution with 16-bit integer arithmetic is given.

```
; Error correction with a single equation per range
; 16-bit arithmetic. Cycles needed:
; ADAT value = 0000h:  47 cycles
; ADAT value = 3FFFh: 178 cycles
;
        MOV     &ADAT,IROP1     ; ADC result Ni to MPY reg.   14.0
        MOV     IROP1,R6        ; Calculation of coeff. address  14.0
        SWPB    R6              ; MSBs to low byte 0...3Fh     6.0
        RRA.B   R6              ;                              5.0
        RRA.B   R6              ; 4n (Range) in R6  0...0Fh    4.0
        BIC     #3,R6           ; 0...0Ch: address of slope a1  4.0
        MOV     TAB1(R6),IROP2L ;  Slope a1                    0.22
        CALL    #MPYS           ; Ni x a1                     ±4.22
        RRA     IRACM           ; Only HI result is used      ±4.5
        RRA     IRACM           ; To format 4.3 of offset a0  ±4.4
        RRA     IRACM           ;                             ±4.3
        ADD     TAB0(R6),IRACM  ; Add Offset a0               ±5.3
        RRA     IRACM           ; Nicorr = Ni x a1 + a0       ±5.2
        RRA     IRACM           ;                             ±5.1
        RRA     IRACM           ; Carry is used for rounding  ±5.0
        ADDC    &ADAT,IRACM     ; Nicorr in IRACM             14.0
        ...                     ; Proceed with corr. result Nicorr
;
; The 16 RAM bytes starting at label TAB1 contain the
; correction info a1 and a0 for all four ranges. The bytes
; are loaded during the calibration
;
        .bss    TAB1,2          ; Range A a1: lin. coefficient  ±0.22
        .bss    TAB0,2          ; a0: constant coefficient      ±5.3
        .bss    TABx,12         ; Ranges B, C, D: a1, a0.
; Run time optimized 16-bit Multiplication Subroutines
;
IROP1  .EQU    R11             ; Unsigned ADC result (0...3FFFh)
IROP2L .EQU    R12             ; Signed factor (8000h...7FFFh)
IROP2M .EQU    R13             ; High word of signed factor (0)
IRACL  .EQU    R14             ; Result word low
IRACM  .EQU    R15             ; Result word high
;
; Signed multiply subroutine: IROP1 x IROP2L -> IRACM|IRACL
;
```

```
MPYS    CLR   IRACL            ; 0 -> result word low
        CLR   IRACM            ; 0 -> result word high
        TST   IROP2L           ; Sign of factor a1
        JGE   MACU             ; Positive sign: proceed
        SUB   IROP1,IRACM      ; Correct result
MACU    CLR   IROP2M           ; Clear MSBs multiplier
L$002   BIT   #1,IROP1         ; Test actual bit (LSB)
        JZ    L$01             ; If 0: do nothing
        ADD   IROP2L,IRACL     ; If 1: add multiplier to result
        ADDC  IROP2M,IRACM
L$01    RLA   IROP2L           ; Double multiplier IROP2
        RLC   IROP2M           ;
;
        RRC   IROP1            ; Next bit of IROP1 to LSB
        JNZ   L$002            ; If IROP1 = 0: finished
        RET
```

EXAMPLE: (8-bit arithmetic). The ADC is measured at five points of the ADC range A (n = 0). The measured errors—device 1 is used—are shown below. The correction coefficients for the range A are calculated with the linear regression method. The correction coefficients for the other three ranges may be calculated the same way. The used numbers are shaded.

ADC Step	50	1024	2048	3072	4096
Subdivision	1.56	32	64	96	128
Error [Steps]	–6	–6	–8	–12	–13

The correction coefficients for the range A (n=0), are calculated with the formulas shown in section 1.2.2.

$$a_1 = + 0.06326 \qquad \textit{Negated result of linear coefficient}$$
$$a_0 = + 4.9312 \qquad \textit{Negated result of constant coefficient}$$

Correction: $\left[\left(\dfrac{Ni}{4096} - 0 \right) \times 128 \times a_1 + a_0 \right] = \left(\dfrac{Ni}{32} \times 0.06326 + 4.9312 \right)$

The correction for the ADC step 2000—located in range A—is calculated:

$$\frac{Ni}{32} \times 0.06326 + 4.93 = \frac{2000}{32} \times 0.06326 + 4.93 = + 8.88$$

Corrected ADC sample: $Nicorr = Ni + 8.9$

Format: Valid for the ADC step 2000

a1: ±0.9   +0.06326/2⁻⁹ = +32.4 ≈ 20h $+0.06326/2^{-9} = +32.4 \approx 20h$

a0: ±5.2   +4.93/2⁻² = +19.7 ≈ 14h $+4.93/2^{-2} = +19.7 \approx 14h$

EXAMPLE: (16-bit arithmetic). The ADC is measured at five points of the ADC range C. The measured errors—device 1 is used—are shown in the table below. The correction coefficients for the range C are calculated with the linear regression method. The correction coefficients for the other three ranges may be calculated the same way. The used numbers are shaded.

ADC Step	8192	9216	10240	11254	12288
Error [Steps]	–9.6	–8.6	–5.2	–1	+0.1

The correction coefficients for the range C are calculated with the formulas shown in section 1.2.2. The full 14-bit ADC result is used for the calculations due to the available 16 bits of resolution.

$$a_1 = -0.0026381701 \qquad \textit{Negated result of linear coefficient}$$
$$a_0 = +31.8695 \qquad \textit{Negated result of constant coefficient}$$

Correction: $Ni \times a_1 + a_0 = Ni \times (-0.00263817) + 31.8695$

The correction for the ADC step 12000—located in range C—is calculated:

$Ni \times (-0.00263817) + 31.8695 = 12000 \times (-0.00263817) + 31.8695 = +0.204$

Corrected ADC sample: $Ni_{corr} = Ni + 0.2$ Valid for the ADC step 12000

Format:

a_1: ± 0.22 $-0.0026381701/2^{-22} = -11065.3 \approx$ D4C7h
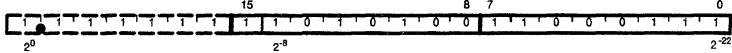
a_0: ± 5.3 $+31.86958564/2^{-3} = +254.96 \approx$ 00FFh
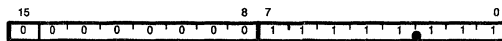
1.2.2.2 Multiple Linear Equations per Range

The ADC is measured at $(p \times k)$ points over the four ranges $(p = 2^m \geq 8)$. Out of these $(p \times k)$ results p linear equations are calculated using the Least Squares Method. The calculated slopes and offsets are stored in the RAM or in EEPROM. The formula for the correction is:

$$Ni_{corr} = Ni + \left[\left(\frac{Ni \times p}{2^{14}} - n_1 \right) \times 128 \times a_1 + a_0 \right]$$

Where:

Ni_{corr}	= Corrected ADC sample	[Steps]
Ni	= Measured ADC sample (noncorrected)	[Steps]
p	= Number of sections for the full ADC range. p is a power of 2.	
n_1	= Value of the MSBS of Ni. n_1 ranges from 0 to $(p-1)$	
a_1	= Slope of the correction	
a_0	= Offset of the correction	
k	= Number of samples for each linear equation (section)	

The value n_1 is explained in section *Multiple Linear Equations (Border Fit)*, section 1.2.1.2.

The term $\left(\dfrac{Ni \times p}{2^{14}} - n_1\right) \times 128$ in the above equation is the adaptation of a complete section—here a half range—to 128 subdivisions. The calculation is made by simple shifts and logical AND instructions.

The principle of this method—with four samples within each one of the eight sections ($k = 4$, $p = 8$)—is shown in Figure 11, the ADC samples are shown only in range A:

Device 1
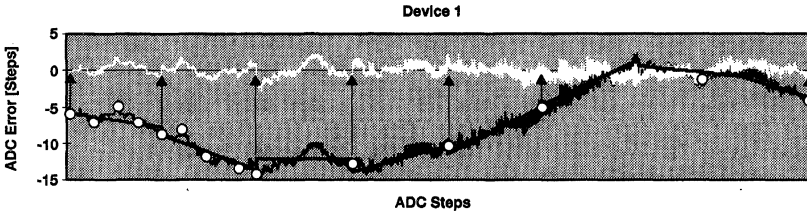
Figure 11. Principle of the Linear Regression Method (two linear equations per range)

The statistical results for 16 points per range—eight samples for each one of the eight linear equations ($k = 8$, $p = 8$)—are:

	Full range	Ranges A and B only
Mean Value:	−0.03 Steps	+0.09 Steps
Range:	4.84 Steps	4.80 Steps
Standard Deviation:	0.78 Steps	0.79 Steps
Variance:	0.61 Steps	0.63 Steps

The result is shown in Figure 12. Note the error range of this figure: only ±3 ADC steps.

Device 1 Corrected With Eight Linear Equations (Linear Regression Used)
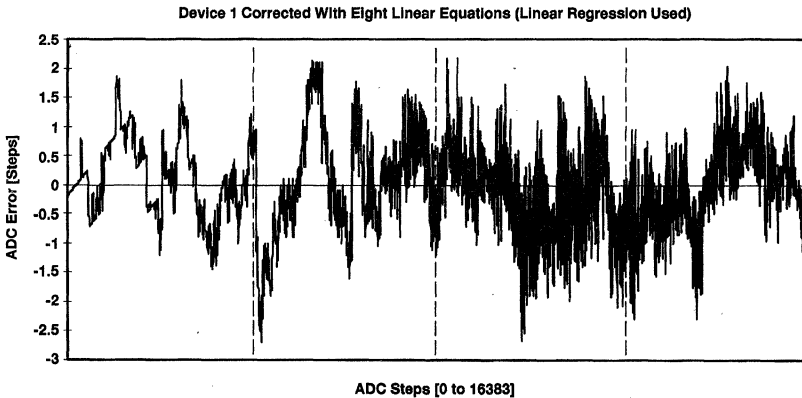
Figure 12. Error Correction With Linear Regression (two linear equations per range)

Advantages: Very good adaptation to the ADC characteristic
Method can be adapted to specific needs with more equations
per range

Disadvantages: Multiplication is necessary
Small gaps at the borders of the four ranges
Calculation of linear regression is necessary during calibration
Many measurements are necessary during calibration (64 with
the above example)

The correction software for the 8-bit arithmetic:

```
; Error correction with two linear equations per range
; (8 for the full ADC range) 8-bit arithmetic. Cycles needed:
; Subdivision = 0:    48 cycles
; Subdivision > 3Fh:  97 cycles
;
    MOV    &ADAT,R5       ; ADC result Ni to R5              14.0
    MOV    R5,R6          ; Address info for correction      14.0
    AND    #07FFh,R5      ; Delete 3 MSBs (n1 bits)          11.0
    RLA    R5             ; Calculate subdivision
    RLA    R5             ; Prepare                          13.0
    RLA    R5             ; ((Ni x p/2^14)-n1)x 128          14.0
    RLA    R5             ; 7 bit ADC info to high byte      15.0
    SWPB   R5             ; ADC info to low byte 0...7Fh      7.0
    MOV.B  R5,IROP1       ; To MPY operand register           7.0
;
    SWPB   R6             ; Calculate coeff. address          6.0
    RRA.B  R6             ; 0...3Fh to 0...1Fh                5.0
    RRA.B  R6             ; 2 x n1 in R6  0...0Fh             4.0
    BIC    #1,R6          ; 0...0Eh: address of slope a1      4.0
    MOV.B  TAB1(R6),IROP2L ; Slope a1    ±0.10
    CALL   #MPYS8         ; ((Ni x p/2^14)-n1)x 128 x a1    ±4.10
    SWPB   IRACL          ; MPY result to a0 format ±         4.2
    ADD.B  TAB0(R6),IRACL ; (nnn)x 128 x a1 + a0            ±5.2
    SXT    IRACL          ;                                 ±5.2
    RRA    IRACL          ; To integer format               ±5.1
    RRA    IRACL          ; Carry is used for rounding       ±5.0
    ADDC   &ADAT,IRACL    ; Corrected result Nicorr          14.0
    ...                   ; Proceed with Nicorr in IRACL
;
; The 16 RAM bytes starting at label TAB contain the correction
; coefficients a1 and a0. The bytes are loaded during the
; initialization. 8-bit, signed numbers
;
    .bss   TAB1,1         ; Range A: a1                      ±0.9
```

```
.bss   TAB0,1        ; a0                        ±5.2
.bss   TABx,14       ; Range B, C, D: a1, a0.
```

EXAMPLE: The ADC ranges are split into two sections each. The measured errors of five points located in the upper section of range B—device 1 is used—are shown below ($k = 4$, $p = 8$). The correction coefficients for this section are calculated with the linear regression method. The correction coefficients for the other seven sections may be calculated the same way.

ADC Step	6144	6656	7168	7680	8192
Subdivision	0	32	64	96	128
Error [Steps]	−14	−13.6	−12	−10.5	−9.6

The correction coefficients a_1 and a_0 for the upper section of range B ($n_1 = 3$) are calculated with the formulas shown in section 1.2.2. The subdivision of the ADC step (0 to 127) is used (8-bit arithmetic).

$$a_1 = + 0.03719 \qquad \textit{Negated value}$$
$$a_0 = + 14.32 \qquad \textit{Negated value}$$

Correction:

$$\left[\left(\frac{Ni \times p}{2^{14}} - m_1 \right) \times 128 \times a_0 + a_1 \right] = \left[\left(\frac{Ni \times 8}{2^{14}} - 3 \right) \times 128 \times (- 0.03719) + 14.32 \right]$$

The correction for the ADC step 7000—located in the upper section of range B—is calculated:

$$\left(\frac{7000 \times 8}{2^{14}} - 3 \right) \times 128 \times (- 0.03719) + 14.32 = + 12.33$$

Corrected ADC sample: $Ni_{corr} = Ni + 12.3$ Valid for ADC step 7000 range B

Format:

a1: ±0.10 $-0.03719/2^{-10} \approx -38$ = DAh
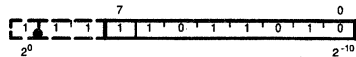
a0: ±5.2 $+14.32/2^{-2} = 57.3 \approx 39h$
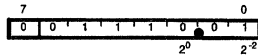
2 Additional Information

The application report *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic*[4] shows nonlinear methods such as quadratic and cubic corrections for the improvement of the 14-bit analog-to-digital converter of the MSP430. Also included are the integer multiplication subroutines for the fast correction software and considerations to the obtainable accuracy with the 8-bit software. Finally all explained correction methods presented are compared by ROM and RAM needs, accuracy improvement, and required CPU cycles.

3 References

1. *Architecture and Function of the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA045
2. *Application Basics for the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA046
3. *Additive Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA047
4. *Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA050
5. *MSP430 Application Report*, 1998, Literature #SLAAE10C
6. Data Sheet MSP430C325, MSP430P323, 1997, Literature #SLASE06B
7. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B

Appendix A   Definitions Used With the Application Examples

```
; HARDWARE DEFINITIONS
;
ACTL    .equ   0114h ; ADC control register: control bits
ADAT    .equ   0118h ; ADC data register (12 or 14-bits)
```

Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic

TEXAS INSTRUMENTS

Contents

List of Figures

List of Tables

Nonlinear Improvement of the MSP430 14-Bit ADC Characteristic

Lutz Bierl

ABSTRACT

This application report shows nonlinear methods—with quadratic and cubic equations —to improve the accuracy of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. The methods used differ in RAM and ROM requirements, calculation speed, achievable improvement, and complexity. The influence of the restricted calculation accuracy for 8-bit coefficients is compared to the accuracy of floating-point calculations. Finally, a comparison of all improvement methods is given. The *References* section at the end of the report lists related application reports in the MSP430 14-bit ADC series.

1   Introduction

The application report *Architecture and Function of the MSP430 14-Bit ADC*[1] gives a detailed overview of the architecture and function of the 14-bit analog-to-digital converter (ADC) of the MSP430 family. The principle of the ADC is explained and software examples are given. Also included are the explanation of the function of all hardware registers contained in the ADC.

The application report *Application Basics for the MSP430 14-Bit ADC*[2] shows several applications of the 14-bit ADC of the MSP430 family. Proven software examples and basic circuitry are shown and explained.

The application report *Additive Improvement of the MSP430 14-Bit ADC Characteristic*[3] explains the external hardware that is needed for the measurement of the characteristic of the MSP430's analog-to-digital converter. This report also demonstrates correction methods that use addition only: no multiplication is needed. This allows the application of these methods in real-time systems, where execution time can be critical.

The application report *Linear Improvement of the MSP430 14-Bit ADC Characteristic*[4] shows linear improvements using linear equations with border fit and correction by linear regression methods.

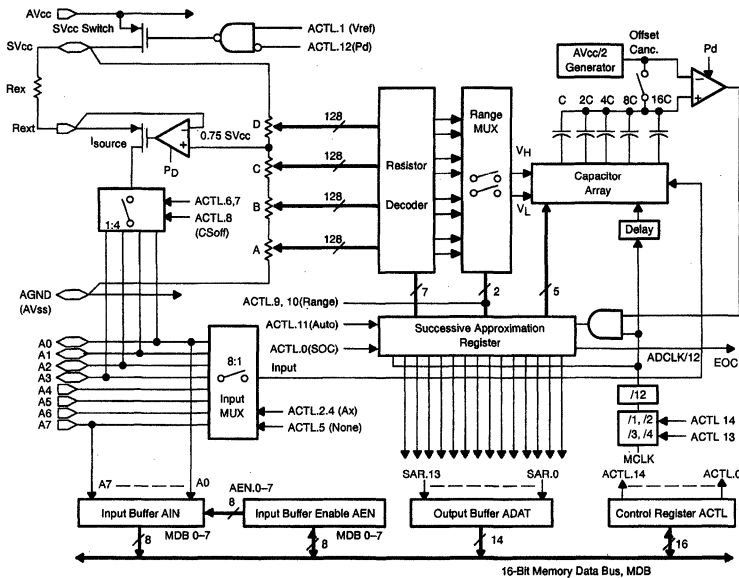Figure 1 shows the block diagram of the 14-bit analog-to-digital converter of the MSP430 family.
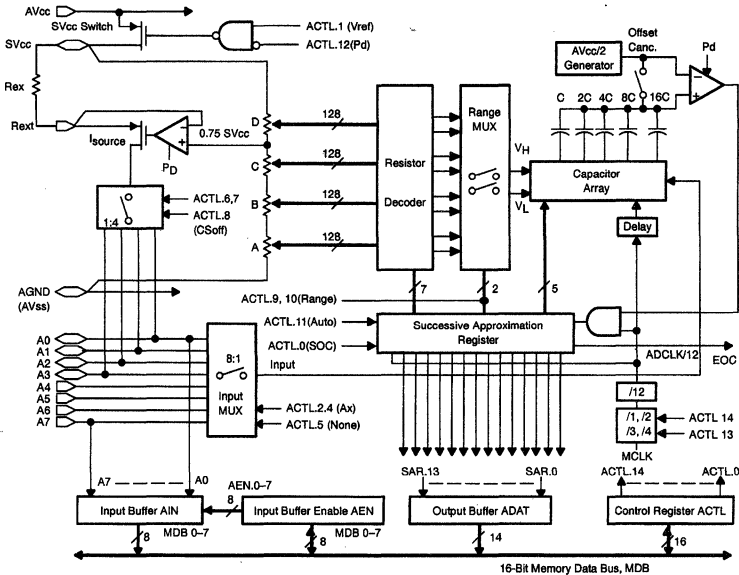
Figure 1. The Hardware of the 14-Bit Analog-to-Digital Converter

The methods for the improvement of the ADC discussed in this report are:

- Correction with nonlinear equations
- Correction with one quadratic equation per range
- Correction with one cubic equation per range

1.1 Correction With Quadratic Equations

The ADC is measured at three points within all four ranges. These points are used for a quadratic correction (one correction for each range). It is recommended to use more than one measurement for each one of these three important points. *Additive Improvement of the MSP430 14-Bit ADC Characteristic*[3] section 2.1 for details. Normally these three points are the two borders and the center of the actual range. This has two advantages:

- The ranges continue smoothly at the common borders.
- Only nine points of the ADC characteristic need to be measured for the full ADC range during the calibration. This is due to the common range border points.

But other arrangements are possible.

The improvement methods and their results for this report are demonstrated with the characteristic of device 1 due to its worst characteristic compared to the other three devices shown in *Architecture and Function of the MSP430 14-Bit ADC Application Report.*[1]

The formula used for each range with separate factors *ax* for 8-bit integer calculations is:

$$Nicorr = Ni + \left(\left(\left(\frac{Ni}{4096} - n \right) \times 256 \right)^2 \times a2 + \left(\frac{Ni}{4096} - n \right) \times 256 \times a1 + a0 \right)$$

Where:
Nicorr	Corrected ADC sample	[Steps]
Ni	Measured ADC sample (non-corrected)	[Steps]
N	Subdivision representing the ADC sample (0...255)	
n	Range number (0...3 for ranges A...D)	
a2	Quadratic coefficient of the correction	[Steps^{-1}]
a1	Linear coefficient of the correction	
a0	Offset of the correction	[Steps]
i	Nominal ADC step of the ADC input (DAC output)	[Steps]

The term $N = \left(\frac{Ni}{4096} - n \right) \times 256$ of the equation above is the adaptation of a complete section—here a full range—to 256 subdivisions. The calculation of the term is made by simple shifts rather than division and a multiplication. Rounding is used to achieve better accuracy. See the initialization part of the software example.

The formula above uses the subdivisions (0 to 255) inside of an ADC range (0 to 4095 steps) instead of the full 14-bit position (0 to 16383 steps) of an ADC point. This is to maintain the accuracy of the calculation with limited coefficient length (here for 8-bit coefficients). The above formula is used with the 8-bit calculation.

If floating point calculation or 16-bit arithmetic is used, the higher resolution makes the range correction unnecessary: the full 14-bit result may be used for the calculations.

$$Nicorr = Ni + \left(Ni^2 \times a2 + Ni \times a1 + a0 \right)$$

The software example given for the cubic correction in section 1.3—which is written in floating point notation—may be adapted easily to quadratic correction: the unused cubic part is simply left out and the address calculation for the coefficients is modified to three coefficients (a2..a0) instead of the four (a3..a0).

To save multiplications, the so-called *Horner* scheme is used. This scheme is applicable for all given examples. The formula using the 8-bit arithmetic now becomes:

$$Nicorr = Ni + \left(\left(\left(\frac{Ni}{4096} - n \right) \times 256 \times a2 + a1 \right) \times \left(\frac{Ni}{4096} - n \right) \times 256 \times a0 \right)$$

The 16-bit formula and the FPP formula now require only two multiplications instead of three.

$$Nicorr = Ni + ((Ni \times a2 + a1)Ni + a0)$$

Figure 2 shows the principle of the correction with four quadratic equations: the used correction parabolas are drawn together with the non-corrected ADC characteristic. As with all principle figures in this report, the black straight line indicates the correction value, the scribbled black line indicates the non-corrected ADC characteristic and the white line shows the corrected ADC characteristic. The small circles indicate the measured ADC points.

Device 1
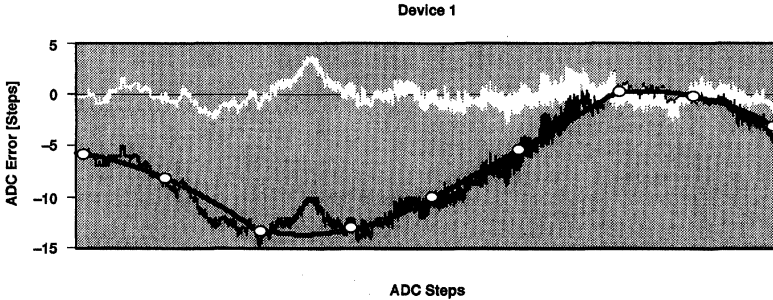
ADC Steps

Figure 2. Principle of the Error Correction With Four Quadratic Equations

The statistical results for the quadratic correction are (single measurement for each one of the nine ADC steps used for the calculation of the correction coefficients):

	Full range	**Ranges A and B only**
Mean Value:	−0.08 Steps	0.24 Steps
Range:	6.78 Steps	5.86 Steps
Standard Deviation:	1.05 Steps	1.10 Steps
Variance:	1.11 Steps	1.21 Steps

If each of the nine ADC steps used for the calculation of the correction coefficients is measured in a slightly modified way, then the statistical results change also. Now the mean value of seven measured ADC steps is taken for the calculation. The seven ADC steps are:

$Nn-12$, $Nn-8$, $Nn-4$, Nn, $Nn+4$, $Nn+8$ and $Nn+12$, where Nn is the ADC step used in the calculation formula. Now the statistical results are:

	Full range	**Ranges A and B only**
Mean Value:	−0.07 Steps	0.11 Steps
Range:	6.47 Steps	6.02 Steps
Standard Deviation:	1.00 Steps	1.12 Steps
Variance:	1.00 Steps	1.24 Steps

Figure 3 shows the resulting errors of both methods in a graph (differences cannot be seen):

Device 1 with quadratic Correction

ADC Steps [0 to 16383]
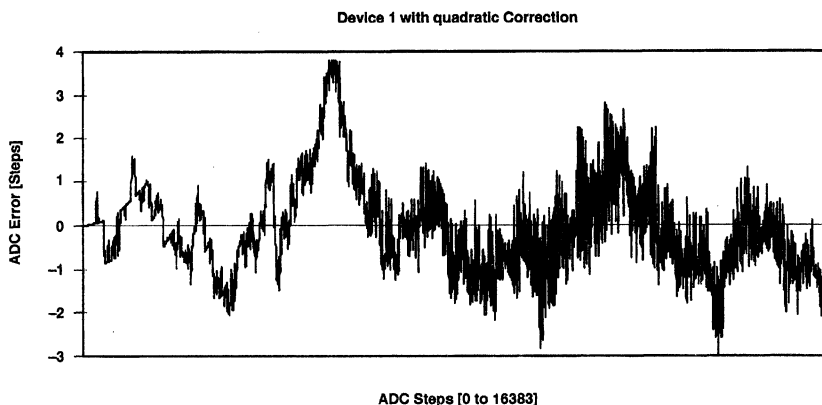
Figure 3. Error Correction With Four Quadratic Equations

Advantages: Only nine ADC measurements are necessary
No gaps at the range borders: perfect continuation
The MSP430 can calculate the correction coefficients
a2 to a0

Disadvantages: Two multiplications are necessary (with Horner
scheme)

1.2 Coefficients Estimation

With the maximum possible ADC error (± 10 steps contained in a band of ± 20 steps like shown in Figure 6) the maximum values for the coefficients a2 to a0 are shown in Table 1. Also given are the valences of the MSBs and the LSBs and the possible coefficient range. In Figure 6, the range C shows the worst case for a quadratic error curve. This curve is the basis for Table 1.

Table 1. Worst Case Coefficients for Quadratic Equations (8-Bit)

COEFFICIENT	MAXIMUM COEFFICIENT	VALENCE OF MSB (BIT 6)	VALENCE OF LSB (BIT 0)	COEFFICIENT RANGE
Quadratic coefficient a2	$\pm 6.103515E{-}4$	2^{-11}	2^{-17}	$\pm 9.7E{-}4$
Linear coefficient a1	$\pm 1.171875E{-}1$	2^{-4}	2^{-10}	$\pm 1.25E{-}1$
Constant coefficient a0	$\pm 2.00000E{+}1$	2^{+4}	2^{-2}	$\pm 3.2000E{+}1$

The integer calculation operates with signed 8-bit coefficients and ax ADC result rounded to 8 bits. The floating point calculation uses the full ADC result (0 to 16383) and a 32-bit format for the calculations.

To give an idea concerning the actual magnitudes, the twelve calculated correction coefficients *ax* for device 1 are:

Table 2. 8-Bit Coefficients for the Four Quadratic Equations of Device 1

COEFFICIENT	RANGE A	RANGE B	RANGE C	RANGE D
a2	9.155273E–5	–1.831055E–4	–2.746582E–5	1.517946E–4
a1	4.687500E–3	3.281250E–2	–3.0859375E–2	–2.0992208E–2
a0	6.000000E+0	1.320000E+1	9.600000E+0	–1.000000E–1

The three equations to calculate the correction coefficients a2, a1 and a0 out of the three known errors e3, e2 and e1 at the ADC steps N3, N2 and N1 are:

$$a1 = -\frac{(e_2 - e_1) \times \left(N_3^2 - N_2^2\right) - (e_3 - e_2) \times \left(N_2^2 - N_1^2\right)}{(N_2 - N_1) \times \left(N_3^2 - N_2^2\right) - (N_3 - N_2) \times \left(N_2^2 - N_1^2\right)}$$

$$a2 = -\frac{(e_2 - e_1) - a_1 \times (N_2 - N_1)}{N_2^2 - N_1^2} \qquad a0 = -\left(e1 - a2 \times N_1^2 - a1 \times N1\right)$$

NOTE:

N_3, N_2, and N_1 can be expressed in ADC steps (0...16383), range steps (0...4095) or subdivisions of the range (0...255) for 8-bit calculations. In the following, N represents subdivisions.

As shown with the linear improvements, using more than one quadratic parabola per ADC range is also possible. It is only necessary to adapt the 256 subdivisions to the sections of the ranges, to calculate the new coefficients a2 to a0, and to modify the addressing of the coefficients.

The software part after each ADC measurement is as follows. The numbers at the right border—below *int.frct*—indicate the maximum integer bits and the actual number of fraction bits for the result (integer.fraction). The Horner scheme is used for the calculation.

```
; Quadratic error correction with a single equation per range.
; 8-bit arithmetic. Cycles needed:
; Subdivision N =    0:  85 cycles
; Subdivision N > 7Fh: 206 cycles
;
;                                                          int.frct
      MOV     &ADAT,R5    ; ADC result Ni to R5                14.0
      MOV     R5,R6       ; Address info for correction        14.0
      RRA     R5          ; Calculate subdivision 0...FFh      13.0
      RRA     R5          ; Prepare N = (Ni/4096-n)x256        12.0
      RRA     R5          ; 8 bit ADC info to low byte         11.0
      RRA     R5          ;                                    10.0
      ADC.B   R5          ; Round subdivision 0...FFh           8.0
      JNC     L$1         ; If result overflows to 100h:
      DEC.B   R5          ; Limit subdivision to FFh            8.0
;
L$1 SWPB·     R6          ; Calculate coefficient address       6.0
      RRA.B   R6          ; 0...1Fh                             5.0
      RRA.B   R6          ; 0...0Fh                             4.0
      RRA.B   R6          ; 0...07h                             3.0
      BIC     #1h,R6      ; 0...06h                             3.0
```

```
        PUSH   R6             ; Save 0...6h                          3.0
        RRA.B  R6             ; 0...03h                              2.0
        ADD    @SP+,R6        ; 0...9h (3n) pointer to a2            4.0
;
        MOV.B  R5,IROP1       ; ADC info to MPY register             8.0
        MOV.B  TAB2(R6),IROP2L
                              ; Quadr. slope a2                      0.17
        CALL   #MPYS8         ; N x a2                             +-0.17
        RLA    IRACL          ; To a1 format                       +-0.18
        ADD    #80h,IRACL     ; Round result                       +-0.18
        SWPB   IRACL          ;                                    +-0.10
        MOV.B  IRACL,IROP2L   ; To MPY register                    +-0.10
;
        MOV.B  R5,IROP1       ; Subdivision to MPY register          8.0
        ADD.B  TAB1(R6),IROP2L
                              ; Linear slope a1 added              +-0.10
        CALL   #MPYS8         ; ((N x a2) + a1) x N                +-5.10
;
        ADD    #80h,IRACL     ; Round result                       +-5.10
        SWPB   IRACL          ; To a0 format                       +-5.2
        ADD.B  TAB0(R6),IRACL
                              ; Add a0                             +-5.2
        SXT    IRACL          ; Correction to 16 bit               +-5.2
        RRA    IRACL          ; (((N x a2) + a1) x N) + a0         +-5.1
        RRA    IRACL          ; Carry is used for rounding         +-5.0
        ADDC   &ADAT,IRACL    ; Corrected result Nicorr             14.0
        ...                   ; Use Nicorr in IRACL
;
; The 12 RAM bytes starting at label TAB2 contain the
; correction coefficients a2, a1 and a0 for the four ranges.
; The bytes are loaded during the initialization
;
        .bss   TAB2,1         ; Range A a2: quadr. coeff.          +-0.17
        .bss   TAB1,1         ;          a1: lin. coefficient       +-0.10
        .bss   TAB0,1         ;          a0: constant coeff.        +-5.2
        .bss   TABx,9         ; Ranges B, C, D: a2...a0.
```

EXAMPLE: The ADC is measured at the two borders and the middle of ADC range B (n = 1). The measured errors—device 1 is used—are shown below. The three correction coefficients a2, a1, and a0 for the range B are calculated with the formulas given before. The correction coefficients for the other three ranges may be calculated the same way; only the appertaining border and center errors need to be used. Twelve measurements were made for each ADC step, and the two extremes were discarded: this leads to one decimal fraction digit.

ADC Step	4096	6144	8192
Subdivision N	0	128	256
Error e [Steps]	−13.2	−13.4	−9.6

Error coefficients for the range B:

$a2 = -0.000183106$

$a1 = +0.0328125$

$a0 = +13.2$

For better legibility $N = \left(\dfrac{Ni}{4096} - n\right) \times 256$ is used in the following.

Correction:

$$((N \times a2 + a1) \times N + a0) = ((N \times (-0.000183106) + 0.0328125) \times N + 13.2)$$

The correction for the ADC step 7000—located in range B—is calculated:

$$\left(\left(\left(\frac{7000}{4096}-1\right)\times 256\times(-0.000183106)+0.0328125\right)\times\left(\frac{7000}{4096}-1\right)\times 256+13.2\right)=+13.3$$

Corrected ADC sample: $Nicorr = Ni + 13.3$. Valid for ADC step 7000

Format: a2: ±0.17 $-0.000183106/2^{-17} = -24 = E8h$
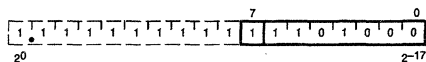
a1: ±0.10 $+0.0328125/2^{-10} = +33.6 \approx 22h$

a0: ±5.2 $+13.2/2^{-2} = +52.8 \approx 35h$

1.3 Correction With Cubic Equations

The ADC is measured at the two borders of each range (common to two ranges) and at one third and two thirds of each range, which results in 13 measurements: e.g., N = 20, 1366, 2731, and 4096 for range A. The errors of these 13 measurements are used for the calculation of four cubic equations, one for each ADC range. It is recommended to use more than one measurement for each of these thirteen points. The resulting correction coefficients a3, a2, a1, and a0 for each ADC range are stored in the RAM or EEPROM.

The above method has two advantages:

- The ranges continue smoothly at the common borders.
- Only thirteen points of the ADC characteristic need to be measured for the full ADC range during the calibration. This is due to the common range border points.

The used formula for each range with separate factors ax for an 8-bit integer calculation is:

$$Nicorr = Ni + \left(N^3 \times a3 + N^2 \times a2 + N \times a1 + a0\right)$$

Where: $N = \left(\dfrac{Ni}{4096}-n\right)\times 256$, the ADC result of a range adapted to the subdivisions 0...255.

Where:
Nicorr	Corrected ADC sample	[Steps]
Ni	Measured ADC sample (non-corrected)	[Steps]
N	Subdivision representing the ADC sample (0...255)	
n	Range number (0...3 for ranges A...D)	
a3	Cubic coefficient of the correction	[Steps^{-2}]
a2	Quadratic coefficient of the correction	[Steps^{-1}]
a1	Linear coefficient of the correction	
a0	Offset of the correction	[Steps]
i	Nominal ADC step of the ADC input (DAC output)	[Steps]

The integer formula above uses the subdivision (0..255) inside of an ADC range (0 to 4095) instead of the full 14-bit position (0 to 16383) of an ADC point. This is to increase the accuracy of the calculation also with limited coefficient length, e.g., for 8-bit coefficients.

If floating point calculation is used, the high resolution of the 24-bit mantissa makes the range correction unnecessary. The equation simplifies to:

$$Nicorr = Ni + \left(Ni^3 \times a3 + Ni^2 \times a2 + Ni \times a1 + a0 \right)$$

To save multiplications the Horner scheme is used again. This reduces the number of multiplications from six to only three. The formula using the 8-bit arithmetic now becomes (N represents the actual subdivision 0...255. See above):

$$Nicorr = Ni + (((N \times a3) + a2) \times N + a1) \times N + a0$$

The formula for 16-bit and floating point calculations now becomes:

$$Nicorr = Ni + (((Ni \times a3) + a2) \times Ni + a1) \times Ni + a0$$

Figure 4 shows the principle of the correction with four cubic equations: the correction parabolas actually used are printed together with the corrected and non-corrected ADC characteristic. The circles indicate the measured ADC points.

Device 1
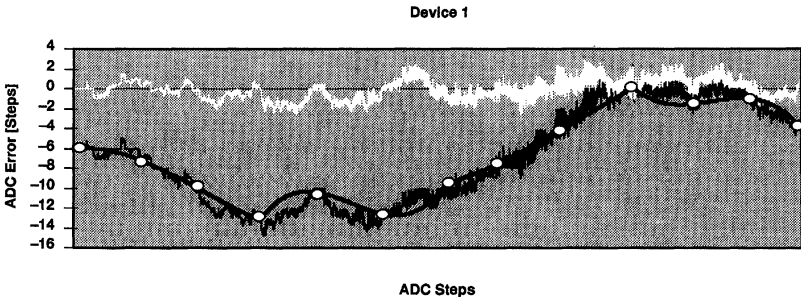
Figure 4. Principle of the Error Correction With Four Cubic Equations

The statistical results for the cubic correction method are:

	Full range	**Ranges A and B only**
Mean Value:	−0.10 Steps	−0.28 Steps
Range:	5.47 Steps	4.97 Steps
Standard Deviation:	0.93 Steps	0.97 Steps
Variance:	0.87 Steps	0.94 Steps

Figure 5 shows the resulting error correction in a graph:

Device 1 corrected with four Cubic Equations

ADC Steps [0 to 16383]
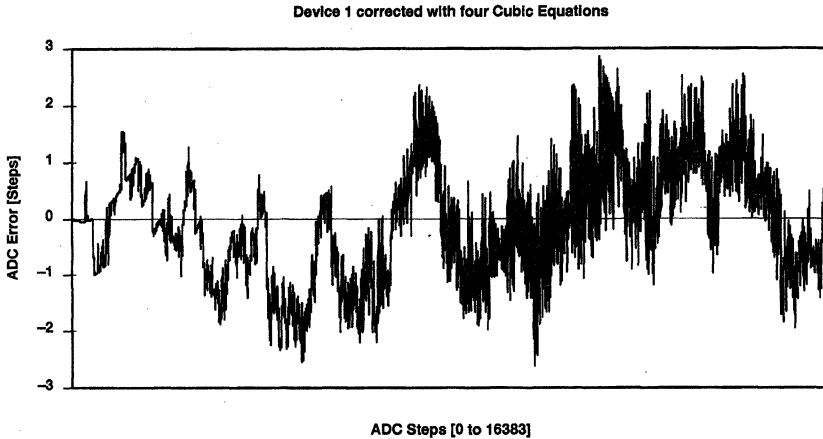
Figure 5. Error Correction With Four Cubic Equations

Advantages: Good adaptation to worst case ADC characteristics
Low storage needs: 16 bytes RAM or EEPROM
(integer calculation)
Monotonicity is ensured due to common samples at the
range borders
Only thirteen ADC measurements are necessary for the
calibration

Disadvantages: Three multiplications are necessary (with HORNER
scheme)
Host is necessary for the calculation of the correction
coefficients ax

1.4 Coefficients Estimation

With the maximum possible ADC error (± 10 steps contained in a band of ± 20
steps like shown in Figure 6) the maximum values for the coefficients a3 to a0 are
shown in Table 3 (8-bit arithmetic). Also given are the valences of the MSB and
the LSB and the possible coefficient range. In Figure 6, the range D shows the
worst case of a cubic error curve. This curve is the basis for Table 3.

Table 3. Worst Case Cubic Coefficients (8-Bit Arithmetic)

COEFFICIENT	MAXIMUM COEFFICIENT	VALENCE OF MSB (BIT 6)	VALENCE OF LSB (BIT 0)	COEFFICIENT RANGE
Cubic coefficient a3	$\pm 6.357828E{-}6$	2^{-18}	2^{-24}	$\pm 7.57E{-}6$
Quadratic coefficient a2	$\pm 2.441406E{-}3$	2^{-9}	2^{-15}	$\pm 3.88E{-}3$
Linear coefficient a1	$\pm 2.473958E{-}1$	2^{-3}	2^{-9}	$\pm 2.48E{-}1$
Constant coefficient a0	$\pm 2.00000E{+}1$	2^{+4}	2^{-2}	$\pm 3.200E{+}1$

The integer calculation operates with signed 8-bit coefficients and an ADC result
rounded to 8 bits (256 subdivisions).

The floating point calculation uses the full ADC result (0 to 16383) and a 32-bit format for the calculations. To give an example, for device 1 the calculated sixteen correction factors a3 to a0 are (12-bit ADC info is used):

Table 4. Correction Coefficients for the Cubic Equations of Device 1

COEFFICIENT	RANGE A	RANGE B	RANGE C	RANGE D
a3	−1.18E−10	−6.483598E−10	3.286326E−11	5.17398E−10
a2	1.02E−06	3.943417E−06	−3.497543E−07	−2.655711E−06
a1	−4.37E−04	−6.153471E−03	−1.486924E−03	3.83333E−03
a0	6.00E+00	1.320000E+01	9.600000E+00	−1.000000E−01

The algorithm to calculate the four correction coefficients a3 to a0 out of the four measured errors e4, e3, e2 and e1 at the ADC steps N4, N3, N2 and N1 is very complex. It is recommended to use a mathematical support software running on a host computer for this task. A simple calculation software routine is available from Texas Instruments on request.

For the cubic correction an example using the MSP430 Floating Point Package FPP4 is given below. This software example can be adapted easily to linear and quadratic correction:

- The parts not used are deleted (e.g., the parts handling the coefficients a3 and a2 if a linear correction is needed)
- The calculation of the start address of the correction coefficients (address of a3 in the example) out of the ADC result is modified slightly.

```
; Cubic error correction with a single equation per range.
; Floating point arithmetic. Cycles needed: 800 to 2400
;
DOUBLE .EQU   0              ; Use .FLOAT format (32 bit)
;
        MOV   #xxx,&ACTL     ; Define ADC measurement
        CALL  #MEASR         ; Measure. Result Ni to ADAT
        CALL  #FLT_SAV       ; Save registers R5 to R12
        SUB   #4,SP          ; Allocate stack for FP result
        MOV   #ADAT,RPARG    ; Load address of ADC buffer
        CALL  #CNV_BIN16U    ; Convert ADC result Ni to FP
        SUB   #4,SP          ; New working space for calc.
;
        MOV   &ADAT,R15      ; Calc. address of coeff. a3
        SWPB  R15
        AND   #0030h,R15     ; Range x 16: rel. address a3
        ADD   #a3,R15        ; Start address of coeff. block
        MOV   R15,RPARG      ; Points to actual a3
        CALL  #FLT_MUL       ; a3 x Ni
        ADD   #a2-a3,R15     ; Address of a2
        MOV   R15,RPARG      ; Points to actual a2
        CALL  #FLT_ADD       ; a3 x Ni + a2
        ADD   #4,RPARG       ; To Ni
        CALL  #FLT_MUL       ; (a3 x Ni+a2)Ni
        ADD   #a2-a3,R15     ; Address of a1
        MOV   R15,RPARG      ; Points to actual a1
        CALL  #FLT_ADD       ; ((a3 x Ni)+a2)Ni + a1
        ADD   #4,RPARG       ; To Ni
        CALL  #FLT_MUL       ; (((a3 x Ni) + a2)Ni + a1)Ni
        ADD   #a2-a3,R15     ; To actual a0
        MOV   R15,RPARG
        CALL  #FLT_ADD       ; (((a3 x Ni)+a2)Ni+a1)Ni+a0
        ADD   #4,RPARG       ; To Ni
        CALL  #FLT_ADD       ; Nicorr = Ni + correction
;
```

```
          POP    2(SP)        ; Result to top of stack
          POP    2(SP)        ; POPs correct the stack
          ...                 ; Continue calc. with Nicorr
          CALL   #FLT_REC     ; Restore registers R12 to R5
          ...                 ; Normal program continues
;
; Correction coefficients are loaded from EEPROM during init.
;
          .bss   a3,4         ; Range A: Cubic coefficient a3
          .bss   a2,4         ; Quadratic coefficient a2
          .bss   a1,4         ; Linear coefficient a1
          .bss   a0,4         ; Constant coefficient a0
;
          .bss   ax,48        ; Ranges B, C and D: a3...a0
```

The assembler software part after each ADC measurement is as follows. The numbers at the right border—below int.frct—indicate the maximum integer bits and the maximum number of fraction bits (integer.fraction). The Horner scheme is used for the calculation.

```
; Cubic error correction with a single equation per range.
; 8-bit arithmetic. Cycles needed:
; Subdivision N =  0:  108 cycles; Subdivision N > 7Fh: 283 cycles
;
;                                                         int.frct
          MOV    &ADAT,R5     ; ADC result Ni to R5          14.0
          MOV    R5,R6        ; Address info for correction  14.0
          RRA    R5           ; Calculate subdivision 0...FFh 13.0
          RRA    R5           ; Prepare N = (Ni/4096-n)x256   12.0
          RRA    R5           ; 8 bit ADC info to low byte    11.0
          RRA    R5           ;                               10.0
          ADC.B  R5           ; Round subdivision 0...FFh      8.0
          JNC    L$1          ; If result overflows to 100h:
          DEC.B  R5           ; Limit subdivision to FFh       8.0
;
L$1       SWPB   R6           ; Calculate coeff. address a3    6.0
          BIC.B  #0Fh,R6      ; 0...30h                        6.0
          RRA.B  R6           ; 0...18h                        5.0
          RRA.B  R6           ; 0...0Ch: address of slope a3   4.0
;
          MOV.B  R5,IROP1     ; Subdivision to MPY register    8.0
          MOV.B  TAB3(R6),IROP2L ; Cubic slope a3              0.24
          CALL   #MPYS8       ; N x a3                       +-0.24
          SWPB   IRACL        ;                              +-0.16
          RRA.B  IRACL        ; To a2 format                 +-0.15
          ADC.B  IRACL        ; Round result                 +-0.15
          MOV.B  IRACL,IROP2L ; To MPY register              +-0.15
;
          MOV.B  R5,IROP1     ; Subdivision to MPY register    8.0
          ADD.B  TAB2(R6),IROP2L ; Quadr. slope a2 added      +-0.15
          CALL   #MPYS8       ; ((N x a3 + a2) x N           +-0.15
          RLA    IRACL        ; To a1 format                 +-0.16
          RLA    IRACL        ;                              +-0.17
          ADD    #080h,IRACL  ; Round result                 +-0.17
          SWPB   IRACL        ;                              +-0.9
          MOV.B  IRACL,IROP2L ; To MPY register              +-0.9
;
          MOV.B  R5,IROP1     ; Subdivision to MPY register    8.0
          ADD.B  TAB1(R6),IROP2L ; Linear slope a1 added      0.9
          CALL   #MPYS8       ; ((N x a3 + a2) x N + a1) x N +-5.9
;
          RLA    IRACL        ; To a0 format                 +-5.10
          ADD    #80h,IRACL   ; Round result                 +-5.10
          SWPB   IRACL        ; To a0 format                 +-5.2
          ADD.B  TAB0(R6),IRACL ; Add a0                     +-5.2
```

```
        SXT    IRACL              ; Correction to 16 bit            +-5.2
        RRA    IRACL              ;                                 +-5.1
        RRA    IRACL              ; Carry is used for rounding      +-5.0
        ADDC   &ADAT,IRACL        ; Corrected result Nicorr         14.0
        ...                       ; Use Nicorr in IRACL
;
; The 16 RAM bytes starting at label TAB3 contain the
; correction coefficients a3, a2, a1 and a0. The bytes are
; loaded during the initialization
;
        .bss   TAB3,1             ; Range A a3: cubic coeff.        +-0.24
        .bss   TAB2,1             ;          a2: quadr. coeff.      +-0.15
        .bss   TAB1,1             ;          a1: lin. coefficient   +-0.9
        .bss   TAB0,1             ;          a0: constant coeff.    +-5.2
        .bss   TABx,12            ; Ranges B, C, D: a3...a0
```

As shown with the linear improvements, it is also possible to use more than one cubic parabola per ADC range. It is only necessary to adapt the 256 subdivisions to the sections of the ranges, to calculate the new coefficients, and to modify the addressing of the coefficients.

EXAMPLE: The ADC is measured at the borders and at one third and two thirds of ADC range D (n = 3). The measured errors—device 1 is used—are shown below. The four cubic correction coefficients a3 to a0 for the range D are calculated with a math package running on a PC. The correction coefficients for the other three ranges may be calculated the same way using the appertaining errors of each range. Twelve measurements were made for each ADC step, the two extremes were discarded: this leads to one decimal fraction digit.

ADC Step	1228	13653	15019	16350
Subdivision N	0	85.33	170.67	253.9
Error e [Steps]	0.1	−1.5	−1.1	−4.0

Error coefficients for the range D:

$a3 = + 0.00000146501$

$a2 = -0.000512371$

$a1 = + 0.0518045$

$a0 = -0.10$

For better legibility $N = \left(\dfrac{Ni}{4096} - n \right) \times 256$ is used in the following.

Correction: $(((N \times a3) + a2) \times N + a1) \times N + a0$

$= (((N \times 0.00000146501) - 0.000512371) \times N + 0.0518045) \times N - 0.10$

The correction for the ADC step 15000—located in range D—is calculated:

$$N = \left(\frac{15000}{4096} - 3 \right) \times 256 = 169.5 \approx 170$$

$(((170 \times 0.00000146501) - 0.000512371) \times 170 + 0.0518045) \times 170 - 0.10 = + 1.1$

Corrected ADC sample: *Nicorr = Ni* + 1.1. Valid for ADC step 15000

Format: a3: ±0.24 $+0.000146501/2^{-24} \approx +24.58 = 19h$
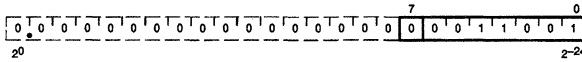
a2: ±0.15 $-0.000512371/2^{-15} = -16.79 \approx EFh$
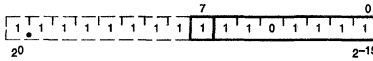
a1: ±0.9 $+0.0518045/2^{-9} = +26.52 \approx 1Bh$
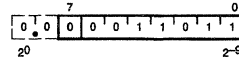
a0: ±5.2 $-0.1/2^{-2} = -0.4 \approx 00h$
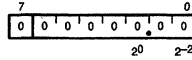
2 Considerations to the Integer Calculations

The calculations for this application report were made with a floating point package. If the 14-bit ADC is used within a real time system the floating point calculation time is normally too long. Therefore the necessary loss of accuracy needs to be known if integer calculations with their restricted bit length are used. The most time consuming parts are the multiplication subroutines, so they are shown first.

2.1 Multiplication Subroutines

To reduce the multiplication time as much as possible, two multiplication subroutines, which terminate immediately after the operand IROP1 becomes zero are shown; this means that the operand with leading zeroes should be in the register IROP1—here the subdivision representing the ADC result.[1]

2.1.1 8-Bit Multiplication Subroutine

If the operands of the multiplication subroutine are normally shorter than 8 bits, then the multiplication subroutine below saves time due to its run time optimization: the multiplication terminates immediately after IROP1 gets zero due to the right shifts during the processing.

```
; Run time optimized 8-bit Multiplication Subroutines
; Definitions
;
IROP1     .EQU  R14        ; Unsigned subdivision (00h...FFh)
IROP2L    .EQU  R13        ; Signed coefficient   (80h...7Fh)
IRACL     .EQU  R12        ; Result word
;
; Cycles for specific registers contents without CALL:
;
; TASK       MACU8 MACS8 MPYS8 IROP1 IROP2 Result (MPYS8)
;-----------------------------------------------------------
; MINIMUM      9    12    13     000h x 000h = 0000h
; MEDIUM      34    37    38     00Fh x 00Fh = 00E1h
; MAXIMUM     66    70    71     0FFh x 0FFh = FF01h
;
; Used registers IROP1, IROP2L, IRACL
;
; Signed multiply subroutine: IROP1 x IROP2L -> IRACL
;
MPYS8 CLR     IRACL        ; 0 -> 16 bit RESULT
;
; Signed multiply-and-accumulate subroutine:
; (IROP1 x IROP2L) + IRACL -> IRACL
;
MACS8 TST.B IROP2L         ; Sign of factor
      JGE   MACU8          ; Positive sign: proceed
      SWPB  IROP1          ; Negative sign: correction nec.
      SUB   IROP1,IRACL    ; Correct result word
      SWPB  IROP1
;
; Unsigned multiply-and-accumulate subroutine (MAC):
; (IROP1 x IROP2L) + IRACL -> IRACL
;
MACU8 BIT.B #1,IROP1       ; Test actual bit (LSB)
      JZ    L$01           ; If 0: do nothing
      ADD   IROP2L,IRACL   ; If 1: add multiplier to resultL$01
L$01  RLA   IROP2L         ; Double multiplier IROP2
      RRC.B IROP1          ; Next bit of IROP1 to LSB
      JNZ   MACU8          ; If IROP1 = 0: finished
      RET
```

[1]The idea for these subroutines initially came from Leslie Mable of TIL.

2.1.1.1 16-Bit Multiplication Subroutine

This multiplication subroutine is used if the 8-bit version is not accurate enough. Like the 8-bit version, the multiplication terminates immediately after IROP1 becomes zero due to the right shifts during the processing. All of the shown ADC improvement methods may be adapted to the 16-bit multiplication subroutine.

```
; Run time optimized 16-bit Multiplication Subroutines
;
IROP1  .EQU  R11       ; Unsigned ADC result (0000h...3FFFh)
IROP2L .EQU  R12       ; Signed coefficient  (8000h...7FFFh)
IROP2M .EQU  R13       ; High word of signed factor (0)
IRACL  .EQU  R14       ; Result word low
IRACM  .EQU  R15       ; Result word high
;
; Cycles for specific register contents without CALL:
; TASK       MACU   MACS   MPYS   IROP1   IROP2  Result (MPYS)
;------------------------------------------------------------------
; MINIMUM    11     14     16     0000h x 0000h = 00000000h
; MEDIUM     83     86     88     00FFh x 00FFh = 0000FE01h
; MAXIMUM    143    147    149    3FFFh x FFFFh = FFFFC001h
;
; Used registers: all of the above ones
;
; Signed multiply subroutine: IROP1 x IROP2L -> IRACM|IRACL
;
MPYS   CLR   IRACL       ; 0 -> result word low
       CLR   IRACM       ; 0 -> result word high
MACS   TST   IROP2L      ; Sign of factor a1
       JGE   MACU        ; Positive sign: proceed
       SUB   IROP1,IRACM ; Correct result
MACU   CLR   IROP2M      ; Clear MSBs of multiplier
L$002  BIT   #1,IROP1    ; Test actual bit (LSB)
       JZ    L$01        ; If 0: do nothing
       ADD   IROP2L,IRACL ; If 1: add multiplier to result
       ADDC  IROP2M,IRACM ;
L$01   RLA   IROP2L      ; Double multiplier IROP2
       RLC   IROP2M      ;
;
       RRC   IROP1       ; Next bit of IROP1 to LSB
       JNZ   L$002       ; If IROP1 = 0: finished
       RET
```

2.2 Maximum Magnitude of the 8-Bit Coefficients

To get the maximum accuracy with the limited 8-bit format used for the correction coefficients, it is necessary to calculate the worst case magnitude for each one of these coefficients. The basis for this calculation is the maximum error of the 14-bit ADC:

±10 steps within a band of ±20 steps

Figure 6 shows examples for the worst case errors of the 14-bit ADC:

- Range A shows the maximum error band of the ADC: ±20 steps; within this error band all errors of different devices are contained. The four dark boxes indicate four possible error ranges of ±10 steps: they are examples for single devices, within such an error band the errors of a single device are contained.
- Range B gives an example for the maximum linear error: within one range (4096 steps) the error changes by 20 steps.
- Range C is an example for a maximum quadratic error.
- Range D is an example for a maximum cubic error. This means within an ADC range the ADC characteristic moves from an error of +10 steps at the lower

range border to +2.5 then back to +7.5 and back to 0 steps at the upper range border.

Examples for Worst Case ADC Errors
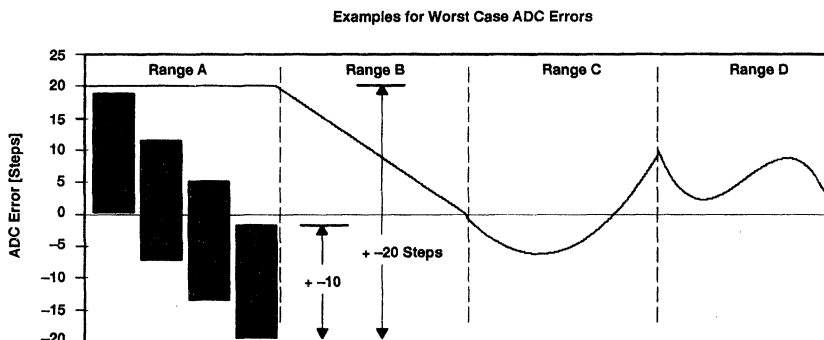
Additive, Linear,Quadratic and Cubic Worst Case Characteristics

Figure 6. Worst Case ADC Error With Different Improvement Methods

Table 5 shows the worst case values—the largest possible values—for the 8-bit correction coefficients that were calculated with the following assumptions:

- The ADC characteristic uses the full error band of ±10 steps.
- The ADC characteristic changes its direction as often as the order of the correction formula, e.g., twice for a quadratic correction.
- The correction is made for each range individually; this means the ADC result bits 13 and 12—the bits defining the ADC range—are cleared.
- The relative ADC result within each range (12 bits) is rounded to eight bits (0 to FFh) for the calculations (8-bit arithmetic).
- The linear coefficient a1 of each method must allow a ±10 step correction.
- The constant coefficient a0 of each method must allow a ±20 step correction.

The maximum correction coefficients calculated with the above assumptions are listed in Table 5. (NA means not applicable):

Table 5. Worst Case Correction Coefficients (8-Bit Arithmetic)

	CUBIC CORRECTION	QUADDRATIC CORRECTION	LINEAR CORRECTION	ADDITIVE CORRECTION
a3	±6.357828E−6	NA	NA	NA
a2	±2.441406E−3	±6.103515E−4	NA	NA
a1	±2.473958E−1	±1.171875E−1	±1.562500E−1	NA
a0	±2.000000E+1	±2.000000E+1	±2.000000E+1	±2.000000E+1

2.3 Number Formats of the 8-Bit Coefficients

The format chosen for the correction data is byte format due to its low storage needs and the speed advantages for multiplications. Figure 7 shows a signed 8-bit number with three fractional bits (±4.3). The range of this number format is −16.00 to +15.875.
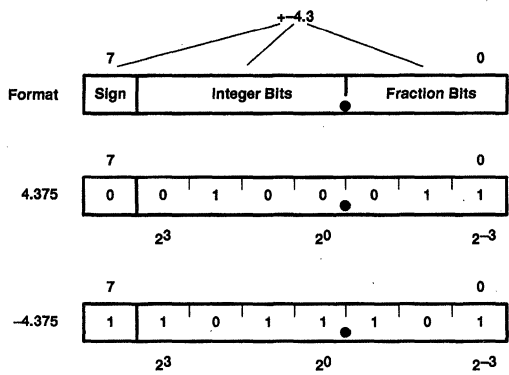
Figure 7. Number Format With Integers for 8-Bit Calculations

·   For the coefficients a3 and a2 no integer parts exist, due to the small values of the resulting numbers. This makes a different format necessary, but the philosophy is the same, to pack very small numbers with many leading zeros or ones into a single byte. Figure 8 shows the number format of the quadratic coefficient a2 used in a cubic correction. All bits of the extended sign have the same value as the sign bit (bit 7).
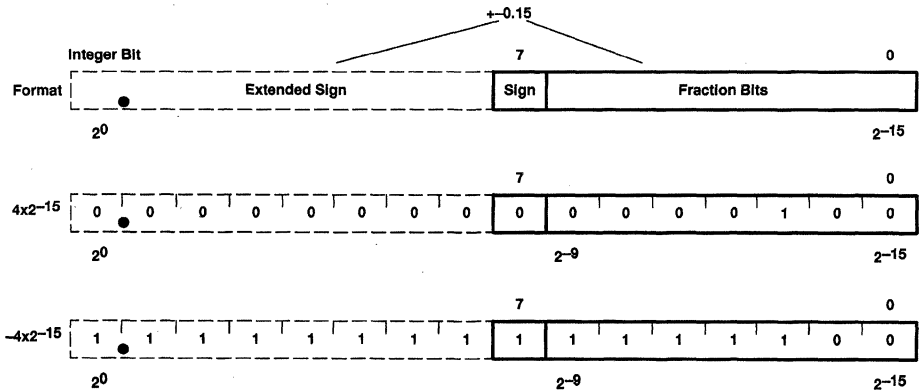
Figure 8. Number Format With Fraction Part Only for 8-Bit Calculations

2.4   Calculation of the 8-Bit Coefficients

To get the subdivision N out of the ADC value—ranging from 0 to 3FFFh—a short calculation is necessary:

$$N = \left(\frac{Ni}{4096} - n \right) \times 128 \quad \text{ranging from 0 to 128 for linear correction}$$

$$N = \left(\frac{Ni}{4096} - n\right) \times 256 \quad \text{ranging from 0 to 256 for quadratic and cubic}$$
correction

With two, three, or four subdivisions N—dependent on the used correction method—the coefficients ax are calculated. See the appropriate sections.

1. To start, it is necessary to find the minimum valence—a power of 2—of bit 6 (MSB) of the 8-bit number that is sufficient for the worst case value of the coefficient ax. The formula for this calculation is:

$$V_{MSB} \geq log2|ax|-1$$

Where: V_{MSB} Valence for the MSB (bit 6) of the 8-bit number
          V_{LSB} Valence for the LSB (bit 0) of the 8-bit number
          ax   Decimal correction coefficient (a3 to a0)

The above formula ensures that the worst case value of the coefficient ax fits into an 8-bit twos complement number.

2. The valence V_{LSB} of the LSB is for 8-bit arithmetic

$$V_{LSB} = V_{MSB} - 6$$

With this valence V_{LSB} the 8-bit coefficient ax8bit is calculated:

$$ax8bit = \frac{ax}{2^{V_{LSB}}}$$

3. The result ax8bit—which is also named ax in the following equations—is converted into a signed hexadecimal number using the twos complement format:

- A positive coefficient is simply converted.
- A negative coefficient is converted and negated afterwards (complemented and incremented).

EXAMPLE: The worst case value for the cubic correction coefficient a3 is ±6.357828E–6 (see Table 5). To find the minimum valence of the MSB of the number format the equation above is used:

$$V_{MSB} \geq log2|ax| - 1 = log2\ 6.357828E - 6 - 1 = -17.263 - 1 = -18.263$$

$$V_{MSB} \geq -18.263 \rightarrow V_{MSB} = -18$$

This result means that bit 6—the MSB—of the 8-bit coefficient a3 must have a minimum valence of 2^{-18}. For the LSB the valence V_{LSB} becomes:

$$V_{LSB} = V_{MSB} - 6 = -18-6 = -24$$

This means, a3 can cover the number range from -128×2^{-24} to $+127 \times 2^{-24}$ ($-7.57E–6$ to $+7.63E–6$) in steps of 2^{-24} (5.96E–8).

Calculation: a3: ±0.24 $+6.357828E–6/2^{-24}$ = +106.67 ≈ 6Bh

This means the worst case of the a3 value results in 107 steps out of 127 steps: good resolution and enough reserve are given. The number format—shown for the negative worst case value of a3 (–107 = 95h)—is:
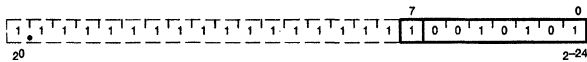
The bits 2^0 to 2^{-17} for the above example always have the same value: they contain the extended sign: the same value as the sign bit in bit 7 of the 8–bit value (2s complement arithmetic):

- Zero for a positive coefficient
- One for a negative coefficient

Information is contained only in the bits 7 to 0 (2^{-18} to 2^{-24} for the above example). This is possible due to the known maximum value of these coefficients.

2.5 Accuracy With the 8-Bit Integer Routines

To show the loss of accuracy when moving from floating point to integer calculations with 8-bit coefficients, the results of the linear and the cubic correction are given.

2.5.1 Accuracy for the Linear Correction

The linear correction—which is not sensitive to coefficient truncation due to the simple algorithm—is shown with both calculation methods. The correction coefficients a1 and a0 of the linear equation shown in section 1.2.1.1, single linear equation per range (with border fit) of *Linear Improvement of the MSP430 14-Bit ADC Characteristic*, SLAA048, [4], were recalculated to fit into signed 8-bit constants with their restricted resolution. With these 8-bit coefficients the calculations were repeated. The statistictical results in comparison to the floating point results are (full range, 8-bit results after rounding):

	32-Bit Floating Point	8-Bit Integer Calculations
Mean Value:	−0.32 Steps	−0.32 Steps
Range:	5.6 Steps	6.0 Steps
Standard Deviation:	0.94 Steps	0.98 Steps
Variance:	0.88 Steps	0.96 Steps

Figure 9 compares the corrected ADC characteristics by floating point calculation vs 8-bit arithmetic (integer result). The difference of the corrected characteristics (FPP result—8-bit result) is displayed also:

- The white, scribbled line indicates the result of the correction using floating point calculations
- The black, scribbled line indicates the result of the correction with 8–bit arithmetic
- The black line below the above two lines indicates the difference between the two corrections using the floating point and the 8-bit arithmetic. The offset is chosen as −4 steps, which means −4 steps represent the zero line of the difference

Device 1 : Comparison between Floating Point and 8–Bit Arithmetic
Single Linear Equation per Range (Border Fit)

ADC Steps [0 to 16383]
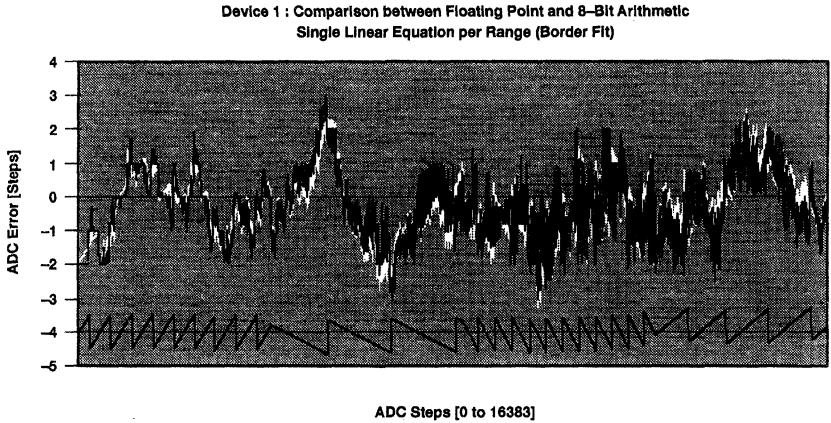
Figure 9. Comparison of Corrected ADC Characteristics. 8-Bit Results After Rounding

As can be seen, the loss of accuracy is not critical (max. ±0.5 steps), the statistical values are nearly identical. This proves that the 8-bit arithmetic is useful for this improvement method due to its speed and storage advantages.

The difference between the floating point and the 8-bit arithmetic looks even better if the 8-bit result is used before the rounding: the two fraction bits of the calculation are used as well. Figure 10 shows this:

Device 1 : Comparison between Floating Point and 8–Bit Arithmetic
Single Linear Equation per Range (Border Fit)

ADC Steps [0 to 16383]
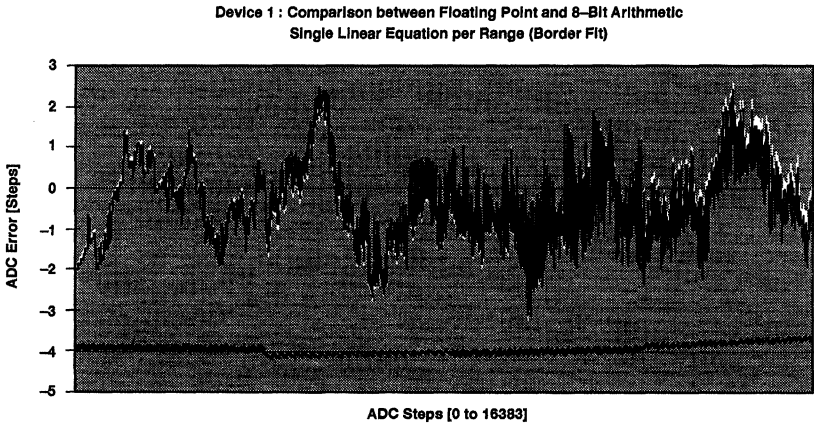
Figure 10. Comparison of Corrected ADC Characteristics. 8-Bit Results Before Rounding

Nearly no difference now exists between the floating point and the 8-bit results.

2.5.2 Accuracy for the Cubic Correction

The cubic correction—which is the most sensitive one to coefficient length due to the third power multiplication—is also shown with both calculation methods.

The correction coefficients a3 to a0 of the cubic equations shown in Table 4 were recalculated to fit into signed 8-bit constants with their restricted resolution. With these 8-bit coefficients the calculations were repeated. The results in comparison to the floating point results are (full range):

	32-Bit Floating Point	8-Bit Integer Calculations
Mean Value:	−0.10 Steps	−0.17 Steps
Range:	5.47 Steps	6.25 Steps
Standard Deviation:	0.93 Steps	1.03 Steps
Variance:	0.87 Steps	1.06 Steps

Figure 11 compares the corrected ADC characteristics by floating point calculation vs 8-bit arithmetic. The difference of the corrected characteristics (FPP result—8-bit result) is displayed as well:

- The white, scribbled line indicates the result of the floating point calculations
- The black, scribbled line indicates the result of the 8-bit arithmetic
- The black line below indicates the difference between floating point and 8-bit arithmetic. The difference is shown before the rounding of the result (two binary digits). The offset is −6 steps, which means −6 steps represent the zero line of this difference

Device 1: Comparison Between Floating Point and 8-Bit Arithmetic Four Cubic Equations

ADC Steps [0 to 16383]
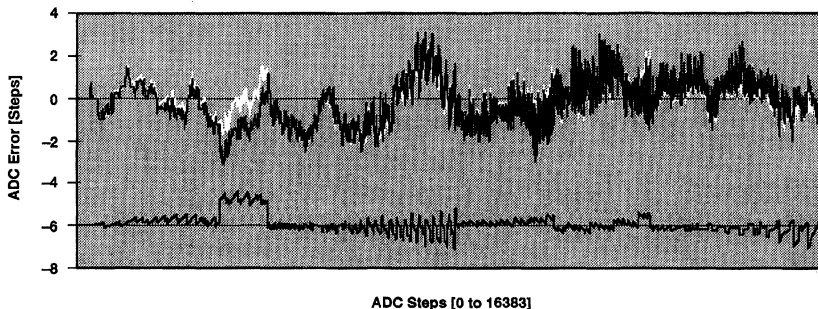
Figure 11. Comparison of Corrected ADC Characteristics. 8-Bit Results Before Rounding

The loss of accuracy is not critical (±1 LSB), but higher than with the linear improvement method. The statistical values are nearly identical to the floating point results.

3 Comparison of the Used Improvement Methods

This section gives an overview to the possible improvements including the effort that is needed to implement them (RAM, ROM, number of measurements during the calibration, calculation time).

3.1 Comparison Tables

Table 6 gives the statistical results for all of the previously described improvement methods. The definitions of the statistical values are given in the application report *Additive Improvement of the MSP430 14-Bit ADC Characteristic*.[3] The most important value of Table 6 is the range: it indicates the worst case value for the error of the ADC (here for device 1). For example, a range of 6.0 means, that the maximum difference of errors is 6.0. All values are ADC steps.

Table 6. Comparison Table for the Different Improvement Methods

CORRECTION METHOD	MEAN VALUE	RANGE	STANDARD DEVIATION	VARIANCE
Additive Corrections:				
Mean value of full range	−0.44	17.1	4.74	22.51
Mean value of 4 ranges	−0.31	13.5	2.49	6.20
Center of ranges	0.20	13.5	2.56	6.53
Multiple sections (8 sections)	−0.14	8.40	1.47	2.16
(16 sections)	−0.29	6.40	1.04	1.08
(32 sections)	0.14	5.20	0.77	0.59
(64 sections)	−0.08	4.60	0.64	0.41
Linear Equations With Border Fit:				
Single linear equation per range	−0.32	5.60	0.94	0.88
Two linear equations per range	−0.29	6.49	0.97	0.94
Four linear equations per range	−0.22	5.36	0.83	0.69
Linear Equations With Linear Regression:				
Single linear equation per range	0.03	5.09	0.94	0.88
Multiple linear equations per range (2)	−0.03	4.84	0.78	0.61
Correction With Quadratic Equations	−0.27	6.96	1.14	1.29
Correction With Cubic Equations:				
8-Bit calculation	−0.17	6.25	1.03	1.06
Floating point calculation	−0.10	5.47	0.93	0.87

Table 7 gives the memory (RAM, ROM, EEPROM) requirements and the necessary number of CPU cycles for all improvement methods. The meaning of the four columns is:

- **RAM/EEPROM:** The number of bytes in the RAM or an external EEPROM that are needed for the continuous storage of the correction coefficients if 8-bit arithmetic is used (full range). It indicates words, if 16-bit arithmetic is used for the calculations. If the correction coefficients are stored in an external memory (e.g., EEPROM), then only a part of this number (the coefficients actually used) need RAM space. If the current source is used, then only one half of the given number is needed (for the ranges A and B only).

- **ROM:** The number of ROM bytes needed for the correction algorithm. The multiplication subroutine is not included in this number.

- **Cycles:** The number of CPU cycles needed for the calculation of the correction.
- **Calibration Samples:** The number of ADC samples that are needed for the used improvement method. The number of actual measurements for each sample is not included in this number. See *Measurement Methods for the ADC Reference Samples* in the application report *Additive Improvement of the MSP430 14-Bit ADC Characteristic*[3] for examples.

Table 7. Comparison Table for the Different Improvement Methods

CORRECTION METHOD	RAM EEPROM BYTES	ROM BYTES	CYCLES	CALIBRATION SAMPLES
Additive Corrections:				
Mean value of full range	2	10	7	16...64
Mean value of 4 ranges	4	24	16	16...64
Center of ranges	4	24	16	4
Multiple sections (8 sections)	8	22	13	9
(16 sections)	16	20	12	17
(32 sections)	32	18	11	33
(64 sections)	64	18	11	65
Linear Equations With Border Fit:				
Single linear equation per range	8	60	51...100	5
Two linear equations per range	16	64	48...97	9
Four linear equations per range	32	56	49...101	17
Linear Equations With Linear Regression:				
Single linear equation per range				
8-Bit calculation	8	60	51...100	16...64
16-Bit calculation	16	44	47...178	16...64
Multiple linear equations per range (2)	16	54	48...97	32...128
Correction With Quadratic Equations	12	84	85...206	9
Correction With Cubic Equations:				
8-Bit calculation	16	102	108...283	13
Floating point calculation	64	88	800...2400	13

3.2 Comparison Graph

To give an impression of how the discussed improvement methods perform, Figure 12 shows the original (non-corrected) characteristic of device 1 and the best improvement in one figure: it is the additive correction with 64 sections described in the application report *Additive Improvement of the MSP430 14-Bit ADC Characteristic*:[3] 64 sections of the ADC range are corrected by the addition of individual constants.

Uncorrected Characteristic compared to best Correction

ADC Steps [0 to 16383]
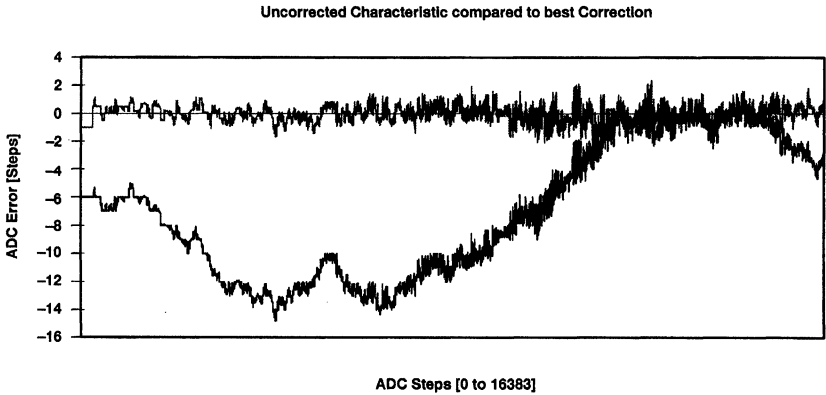
Figure 12. Comparison of the Non-Corrected ADC Characteristic and the Best Improvement

As can be seen, the non-corrected range (17 steps) reduces to a range of less than 5 steps. The statistical results are (full range):

- Best correction: Additive correction of 64 sections for the full ADC range
- Second best correction: Linear regression with two equations per ADC range

	Non-Corrected Device 1	**Best Correction**	**2nd Best Corrected**
Mean Value:	−6.95 Steps	−0.08 Steps	−0.03 Steps
Range:	17 Steps	4.60 Steps	4.84 Steps
Standard Deviation:	4.74 Steps	0.64 Steps	0.78 Steps
Variance:	22.51 Steps	0.41 Steps	0.61 Steps

4 Selection Guide

To quickly find the best improvement method for the 14-bit ADC, Table 8 gives a hint to which method is best for a given application. The selection criteria are:

- Accuracy: The range is used.
- RAM critical: The RAM needed for the coefficients is ≤ 8 bytes.
- Time critical: The calculation time takes ≤ 60 cycles on an average.

This table is taken from the results of device 1. But the table may be usable for other MSP430 devices too. With a more regular ADC characteristic than device 1, the more complex methods will show better results than the simpler ones.

Table 8. Selection for the Improvement Methods

NEEDED ACCURACY	RAM CRITICAL TIME CRITICAL	RAM CRITICAL	TIME CRITICAL	RAM AND TIME NON-CRITICAL
High (±2.5 Steps)	Not possible	Single linear equation/range (linear regression)	Multiple sections (64 sections)	Two linear equations/range (linear regression)
Medium (±3.5 Steps)	Not possible	Single linear equation/range (border fit)	Multiple sections (16 and 32 sections)	All others not named
Low (±7 Steps)	Mean Value/Range Center of Ranges Multiple Sections (8 Sections)			
Very Low (±10 Steps)	Mean value of full range			

5 Summary

The five application reports in this series show many simple-to-realize improvements for the accuracy of the 14-bit analog-to-digital converter of the MSP430. From a non-corrected error range of 17 steps, it was possible to reduce the range to less than ±2.5 steps. Even better, the standard deviation improved from 4.74 steps to 0.65 steps. With a larger effort, the results can be even better—for example if sophisticated statistical methods are applied. Solutions are possible for real-time systems also, e.g., the additive method with its simple and fast algorithm. It was also shown, that relatively simple correction methods can deliver the best results.

6 References

1. *Architecture and Function of the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA045

2. *Application Basics for the MSP430 14-Bit ADC Application Report*, 1999, Literature #SLAA046

3. *Additive Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA047

4. *Linear Improvement of the MSP430 14-Bit ADC Characteristic Application Report*, 1999, Literature #SLAA048

5. *MSP430 Metering Application Report*, 1997, Literature #SLAAE10B

6. Data Sheet MSP430C325, MSP430P323, 1998, Literature #SLASE06A

7. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B

8. *MSP430 Application Report*, 1998, Literature #SLAAE10C

Using the MSP430 Universal Timer/Port Module as an Analog-to-Digital Converter

Lutz Bierl

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

List of Tables

Using the MSP430 Universal Timer/Port Module as an Analog-to-Digital Converter

Lutz Bierl

ABSTRACT

This application report gives a detailed overview of several applications for theMSP430 family Universal Timer/Port Module when used as an analog-to-digital converter (ADC). Proven software examples and basic circuitry are shown and explained.

1 The Universal Timer/Port Module

The function of the Universal Timer/Port Module is completely different from the 14-bit ADC. The discharge times of a capacitor for different resistors are measured and compared.

The module consists of two independent parts, which work together for the measurement of resistors or voltages.

- Counter with Controller: two 8-bit counters, which can be connected in series to get a 16-bit counter. Additionally, there is a controller, a comparator input (CMPI), and a normal input (CIN).

- Input/Output Port: five outputs (TP.0...TP.4), which can be switched to Hi-Z and an I/O-port (TP.5).

Two different inputs are available with the module:

- The CIN input having a Schmitt-Trigger characteristic. It is normally used for resistor measurements. The threshold voltages are the same ones as for the other inputs (P0.x).

- The comparator input CMPI, which is used for the voltage measurement, has a threshold voltage Vref that is nominally $0.25 \times V_{CC}$ with small tolerances. The threshold voltage Vref itself is temperature independent. The input CMPI shares a pin with an LCD select line and must be switched by software to the input function. This input function is valid until the next PUC. The software for the activation of the comparator is:

```
BIS.B    #CPON,&TPD      ; Switch on the comparator
```

The comparator hardware consumes approximately 300 µA, it should be switched off therefore when not in use.

```
BIC.B    #CPON,&TPD      ; Switch off the comparator
```

2 x 8–Bit Counter or 1 x 16–BitCounter with Clock Frequency and Enable Control

| I/O Port | Counter with Controller | Control Registers |
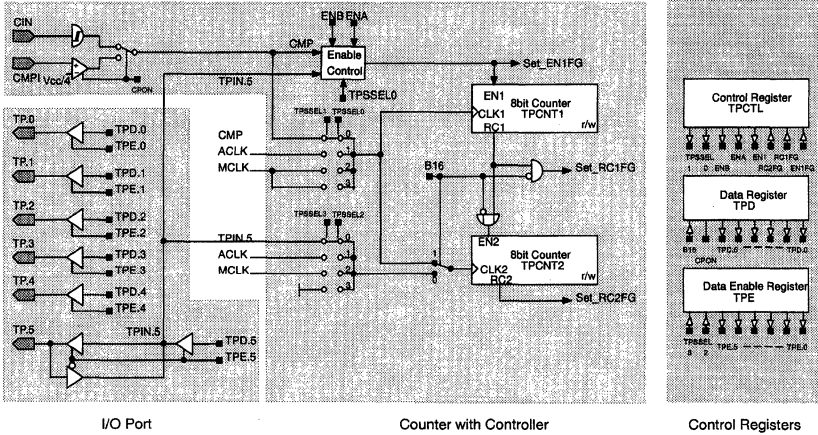
Figure 1. Block Diagram of the Universal Timer/Port Module

Figure 1 shows the minimum hardware required with one sensor Rsens and a single reference resistor Rref.
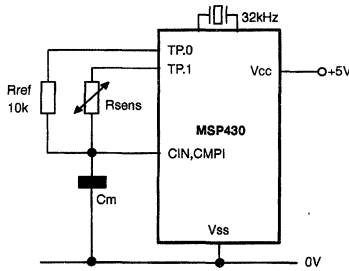
Figure 2. Minimum Sensor Circuit

The voltage at the capacitor Cm during the measurement is shown in Figure 3. The equation that describes the discharge curve for the sensor (Rsens) is:

$$Vth = Vcc \times e^{-\dfrac{tsens}{Cm \times Rsens}} \;\;-> \;\; Rsens = -\dfrac{tsens}{Cm \times ln\dfrac{Vth}{Vcc}}$$

The equation for the reference resistor (Rref) is:

$$Vth = Vcc \times e^{-\dfrac{tref}{Cm \times Rref}} \approx Rref = -\dfrac{tref}{Cm \times ln\dfrac{Vth}{Vcc}}$$
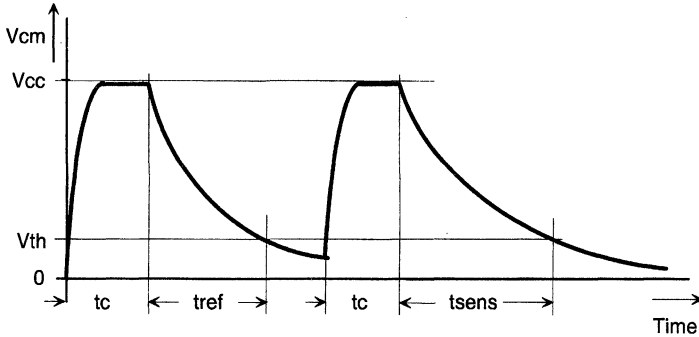
Figure 3. Timing for the Universal Timer/Port Module ADC

Where:

Vth	Threshold voltage of the comparator	[V]
Vcc	Supply voltage of the MSP430	[V]
tref	Discharge time with the reference resistor Rref	[s]
tsens	Discharge time with the sensor Rsens	[s]
tc	Charge time for the capacitor	[s]

The solving of the exponential equation leads to the simple equation in the following:

$$\dfrac{Rsens}{Rref} = \dfrac{-tsens}{Cm \times ln\dfrac{Vth}{Vcc}} \times \dfrac{Cm \times ln\dfrac{Vth}{Vcc}}{-tref} \approx Rsens = Rref \times \dfrac{tsens}{tref}$$

With two known reference resistors (Rref1 and Rref2) it is possible to compute the slope and offset and get the exact values of the unknown resistors. The result of the solved equations gives:

$$Rsens = \frac{tsens - tref2}{tref2 - tref1} \times (Rref2 - Rref1) + Rref2$$

Where:

tsens	Discharge time for sensor Rsens	[s]
tref1	Discharge time for Rref1	[s]
tref2	Discharge time for Rref2	[s]
Rref1	Resistance of reference resistor Rref1	[Ω]
Rref2	Resistance of reference resistor Rref2	[Ω]

As shown only known or measurable values are needed for the computation of Rsens from tsens. The slope and offset of the measurement disappear completely.

To get a resolution of n bits, the capacitor Cm must have a minimum capacity:

$$Cm > \frac{-2^n}{Rx_{min} \times f \times \ln\frac{Vth_{max}}{Vcc}}$$

The approximate conversion time tconv is:

$$tconv \approx \frac{2^n}{f}$$

The complete conversion time tcompl is (reference and sensor measurement):

$$tcompl = 2 \times (tconv + 5 \times Cm \times Rsens)$$

Where:

f	Measurement frequency (ACLK or MCLK)	[Hz]
Rx_{min}	Lowest resistance of sensor or reference resistor	[Ω]
Vth_{max}	Maximum value for threshold voltage Vth	[V]
tconv	Conversion time for an analog-to-digital conversion	[s]

Table 1 gives an overview of different resolutions, capacitors, and conversion times. The sensor resistance is 1 kΩ, f = 1.048 MHz:

Table 1. ADC Conversion With the Timer/Port Module

Resolution Bits	Capacitor Cm	Conversion Time tconv	Complete Conversion Time tcompl
8	232 nF	256 µs	2.8 ms
10	1 µF	1 ms	12.0 ms
12	3.7 µF	4.1 ms	45.2 ms
14	15 µF	16.4 ms	182.8 ms
16	60 µF	65 ms	730 ms

EXAMPLE: Use of the Universal Timer Port as an ADC without an interrupt. The measured time values of the two sensors (Rsens1 and Rsens2) and the reference resistors (Rref1 and Rref2) are stored in RAM starting at label MSTACK (Rref1 location). If an error occurs, 0FFFFh is written to the RAM location.
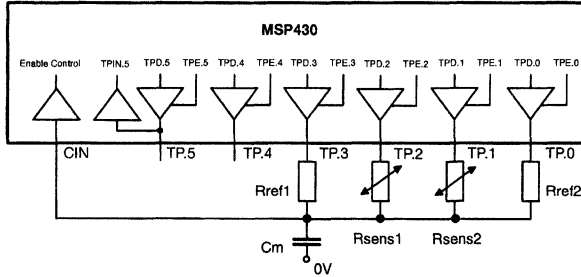
Figure 4. Schematic of Example

```
; DEFINITION PART FOR THE UT/PM ADC
;
TPCTL      .EQU      04Bh              ; TIMER PORT CONTROL REGISTER
TPSSEL0    .EQU      040h              ; TPSSEL.0
ENB        .EQU      020h              ; CONTROLS EN1 OF TPCNT1
ENA        .EQU      010h              ; AS ENB
EN1        .EQU      008h              ; ENABLE INPUT FOR TPCNT1
RC2FG      .EQU      004h              ; RIPPLE CARRY TPCNT2
EN1FG      .EQU      001h              ; EN1 FLAG BIT
;
TPCNT1     .EQU      04Ch              ; LO 8-BIT COUNTER/TIMER
TPCNT2     .EQU      04Dh              ; HI 8-BIT COUNTER/TIMER
;
TPD        .EQU      04Eh              ; DATA REGISTER
B16        .EQU      080h              ; 0: SEPARATE TIMERS  1: 16-BIT TIMER
CPON       .EQU      040h              ; 0: COMP OFF    1: COMP ON
TPDMAX     .EQU      008h              ; BIT POSITION OUTPUT TPD.MAX
;
TPE        .EQU      04Fh              ; DATA ENABLE REGISTER
;
MSTACK     .EQU      0240h             ; Result stack 1st word
NN         .EQU      011h              ; TPCNT2 VALUE FOR CHARGING OF C
;
; MEASUREMENT SUBROUTINE WITHOUT INTERRUPT. TPD.4 AND TPD.5
```

```
; ARE NOT USED AND THEREFORE OVERWRITTEN
; INITIALIZATION: STACK INDEX <- 0, START WITH TPD.3
; 16-BIT TIMER, MCLK, CIN ENABLES COUNTING
;
; Call:  CALL     #MEASURE
;
; Return:         Results for TP.3 to TP.0 in MSTACK to MSTACK+6
;        Result 0FFFFh if error
;
MEASURE  PUSH.B   #TPDMAX           ; START WITH SENSOR AT TPD.MAX
         CLR      R5                ; INDEX FOR RESULT STACK
MEASLOP  MOV.B    #(TPSSEL0*3)+ENA,&TPCTL    ; Reset flags
;
; CAPACITOR C IS CHARGED UP FOR > 5 TAU. N-1 OUTPUTS ARE USED
;
         MOV.B    #B16+TPDMAX-1,&TPD         ; SELECT CHARGE OUTPUTS
         MOV.B    #TPDMAX-1,&TPE    ; ENABLE CHARGE OUTPUTS
         MOV.B    #NN,&TPCNT2       ; LOAD NEG. CHARGE TIME
;
MLP0     BIT.B    #RC2FG,&TPCTL     ; CHARGE TIME ELAPSED?
         JZ       MLP0              ; NO CONTINUE WAITING
;
         MOV.B    @SP,&TPE          ; ENABLE ONLY ACTUAL SENSOR
         CLR.B    &TPCNT2           ; CLEAR HI BYTE TIMER
;
; SWITCH ALL INTERRUPTS OFF, TO ALLOW NON-INTERRUPTED START
; OF TIMER AND CAPACITY DISCHARGE
;
         DINT                       ; ALLOW NEXT 2 INSTRUCTIONS
         CLR.B    &TPCNT1           ; CLEAR LO BYTE TIMER
         BIC.B    @SP,&TPD          ; SWITCH ACTUAL SENSOR TO LO
         MOV.B    #(TPSSEL0*3)+ENA+ENB,&TPCTL      ; Reset flags
         EINT                       ; COMMON START TOOK PLACE
;
; Wait until EOC (EN1 = 1) or overflow error (RC2FG = 1)
;
MLP1     BIT.B    #RC2FG,&TPCTL     ; Overflow (broken sensor)?
         JNZ      MERR              ; Yes, go to error handling
         BIT.B    #EN1,&TPCTL       ; CIN < Ucomp?
         JNZ      MLP1              ; NO, WAIT
```

```
;
; EN1 = 0: End of Conversion: Store 2 x 8 bit result on MSTACK
; Address next sensor, if no one addressed: End reached
;
        MOV.B    &TPCNT1,MSTACK(R5)        ; STORE RESULT ON STACK
        MOV.B    &TPCNT2,MSTACK+1(R5)      ; HI BYTE
L$301   INCD     R5              ; ADDRESS NEXT WORD
        RRA.B    @SP             ; NEXT OUTPUT TPD.x
        JNC      MEASLOP         ; IF C=1: FINISHED
        INCD     SP              ; HOUSEKEEPING: TPDMAX
        RET
;
; ERROR HANDLING: ONLY OVERFLOW POSSIBLE (BROKEN SENSOR ?)
; 0FFFFh IS WRITTEN FOR RESULT AND SUBROUTINE CONTINUED
;
MERR    MOV      #0FFFFh,MSTACK(R5)        ; Overflow
        JMP      L$301
```

1.1 Interrupt Handling

If the Universal Timer/Port Module is used as an ADC for applications that need an accuracy greater than 10 bits, the digital noise generated by the running CPU has a strong influence on the result. If the flag (EN1) in the hardware register TPCTL is polled by software for the signal of a completed conversion then the results are normally different. They show a wide distribution that reflects the length of the polling loop (i.e. the results are concentrated on evenly spaced numbers with nothing in between). To avoid this effect the CPU is switched off during the conversion and woken-up at the completion of the conversion by the ADC interrupt. With this method and adequate hardware, results with much better accuracy are possible.

The influence of the digital noise is shown in Figure 5. The exponential discharge curve is relatively flat near the comparator threshold Vth. Therefore noise coming from the CPU (or other sources of non-wanted noise) can be under the threshold voltage and terminate the conversion. The result is a timer value tdcw that is too low. The correct value would be tdcc. The resulting error Ecnv is:

$$Ecnv = \frac{tdcw - tdcc}{tdcc} \times 100$$

Where:
tdcw	Resulting measurement time caused by CPU noise	[s]
tdcc	Correct measurement time	[s]
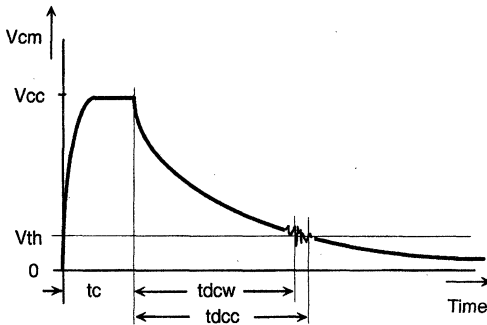
Figure 5. Noise Influence During Measurement

EXAMPLE: The hardware schematic is shown in Figure 6. Two resistive temperature sensors are used (Rmeas0 and Rmeas1) two reference resistors (Rref0 and Rref1) that have the resistance of the sensors at the lower (or upper) end of the measurement range and a resistor (Rcharge) that is used only for the charge-up of the capacitor (Cm). This charge resistor is only necessary if the sensors have low resistance (approximately 100 Ω.). Otherwise, the reference resistors can be used for charging.
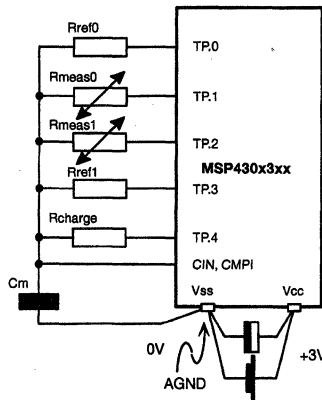
Figure 6. Hardware Schematic for Interrupt Example

The example software works with a status byte (MEASSTAT) that defines the current operation. Normally, this byte is zero, which indicates *no activity* or after a complete measurement sequence *conversions made*. The two reference resistors and two temperature sensors are measured one after the other; Rref0 first, then Rmeas0, then Rmeas1 and finally Rref1.

The measured discharge times (a direct measure for the relative resistance) are placed in successive RAM words starting at label ADCRESULT.

First these RAM words are set to zero (a value impossible as a measurement result). If an error occurs, the zero value indicates an erroneous result.

The Basic Timer is programmed to 0.5 s interrupt timing. The measurement sequence is shown in Figure 7. This sequence can be shortened to one reference resistor and one sensor as well as enlarged up to four sensors and two reference resistors. It is only necessary to add or delete charging and measurement states and the accompanying software parts.

The modulation mode of the FLL is switched off during the measurement to have the exactly same MCLK during all four measurements. Status 9 switches on the modulation mode again.

The software shown can be used for the MSP430C31x and MSP430C32x. The different interrupt enable bits and the different addresses of the interrupt vectors are used correctly by the definition of the software switch *Type*. If this switch is defined as *310*, the MSP430C31x is used; otherwise, the MSP430C32x is used.
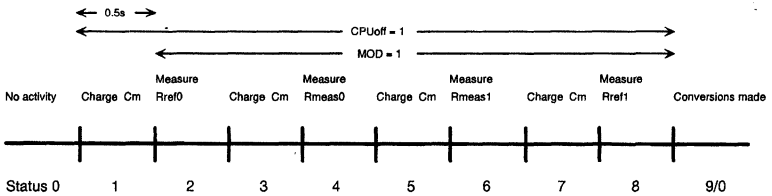
Figure 7. Measurement Sequence

```
; Definitions of the MSP430 hardware
;
Type       .equ     310          ; 310: MSP43C31x   0: others
BTCTL      .equ     040h         ; Basic Timer: Control Reg.
BTCNT1     .equ     046h         ;              Counter
BTCNT2     .equ     047h         ;              Counter
BTIE       .equ     080h         ;              : Intrpt Enable
DIV        .equ     020h         ; BTCTL: xCLK/256
IP2        .equ     004h         ; BTCTL: Clock Divider2
IP0        .equ     001h         ;        Clock Divider0
;
SCFQCTL    .equ     052h         ; FLL Control Register
MOD        .equ     080h         ; Modulation Bit: 1 = off
;
CPUoff     .equ     010h         ; SR: CPU off bit
GIE        .equ     008h         ; SR: General Intrpt enable
```

```
;
TPCTL      .equ     04Bh                ; Timer Port: Control Reg.
TPCNT1     .equ     04Ch                ;             Counter Reg.Lo
TPCNT2     .equ     04Dh                ;             Counter Reg.Hi
TPD        .equ     04Eh                ;             Data Reg.
TPE        .equ     04Fh                ;             Enable Reg.
;
           .if      Type=310            ; MSP430C31x?
TPIE       .equ     004h                ; ADC: Intrpt Enable Bit
           .else
TPIE       .equ     008h                ; MSP430C32x configuration
           .endif
IE2        .equ     001h                ;     Intrpt Enable Byte
TPSSEL1    .equ     080h                ; Selects clock input (TPCTL)
TPSSEL0    .equ     040h                ;
ENB        .equ     020h                ; Selects clock gate (TPCTL)
ENA        .equ     010h                ;
EN1        .equ     008h                ; Gate for TPCNTx    (TPCTL)
RC2FG      .equ     004h                ; Carry of HI counter (TPCTL)
RC1FG      .equ     002h                ; Carry of LO counter (TPCTL)
EN1FG      .equ     001h                ; End of Conversion Flag "
B16        .equ     080h                ; Use 16-bit counter (TPD)
;
Rref0      .equ     001h                ; TP.0: Reference Resistor
Rmeas0     .equ     002h                ; TP.1: Sensor0
Rmeas1     .equ     004h                ; TP.2: Sensor1
Rref1      .equ     008h                ; TP.3: Reference Resistor
Rcharge    .equ     010h                ; TP.4: Charge Resistor
;
; RAM Definitions
;
ADCRESULT          .equ         0200h    ; ADC results (4 words)
MEASSTAT .equ     ADCRESULT+8     ; Measurement Status Byte
;
;============================================================
;
         .sect    "INIT",0F000h      ; Initialization Section
;
INIT     MOV      #0300h,SP          ; Initialize Stack Pointer
         MOV.B    #DIV+IP2+IP0,&BTCTL        ; Basic Timer: 2Hz
```

```
        BIS.B    #BTIE,&IE2       ; Basic Timer Intrpt Enable
        CLR.B    &BTCNT1          ; Clear Basic Timer Regs.
        CLR.B    &BTCNT2
        CALL     #CLRRAM          ; Clear RAM
        ...                       ; Initialize other Modules
;
MAINLOOP ...                      ; Main loop of program
;
; It's time to measure the sensors
;
        MOV.B    #1,MEASSTAT      ; Activate Measurement
        JMP      MEASURE          ; Go to Measurement Part
        ...
; Measurement Part: The CPU is switched off to avoid noise
; that would falsify the measurements. Interrupt is used
; to indicate the end of conversion (and wake-up the CPU).
; The program remains on the NOP until MSTAT9 clears the
; CPUoff-bit of the stored SR on  the  stack.
;
MEASURE CLR      ADCRESULT        ; Clear result buffers
        CLR      ADCRESULT+2      ; 0 indicates error
        CLR      ADCRESULT+4      ;
        CLR      ADCRESULT+6      ;
        MOV      #CPUoff+GIE,SR   ; CPU off, but MCLK on
        NOP                       ; Wait for end of measurement
        ...                       ; Process measured data
;
; Interrupt Handler for the Basic Timer Interrupt: 2Hz
;
BT_INT  PUSH     R5               ; Save Help Register
        CALL     #INCRWTCH        ; Incr. Watch
        MOV.B    MEASSTAT,R5      ; Calculate Handler
        MOV.B    TABLE(R5),R5     ; Offset for PC
        ADD      R5,PC            ; Add Offset to PC
TABLE   .BYTE    MSTAT0-TABLE     ; 0: No activity
        .BYTE    MSTAT1-TABLE     ; 1: Charge for Rref0
        .BYTE    MSTAT2-TABLE     ; 2: Measure Rref0
        .BYTE    MSTAT1-TABLE     ; 3: Charge for Rmeas0
        .BYTE    MSTAT4-TABLE     ; 4: Measure Rmeas0
        .BYTE    MSTAT1-TABLE     ; 5: Charge for Rmeas1
```

```
          .BYTE    MSTAT6-TABLE      ; 6: Measure Rmeas1
          .BYTE    MSTAT1-TABLE      ; 7: Charge for Rref1
          .BYTE    MSTAT8-TABLE      ; 8: Measure Rref1
          .BYTE    MSTAT9-TABLE      ; 9: Finished, go on
;
MSTAT1    MOV.B    #B16+Rcharge,&TPD ; Charge Cm for 0.5s
          MOV.B    #Rcharge,&TPE     ; Use Rcharge
          JMP      BT_RET
;
MSTAT2    MOV      #Rref0,R5         ; Measure Rref0
          JMP      MEASCOM           ; To common Part
;
MSTAT4    MOV      #Rmeas0,R5        ; Measure Rmeas0
          JMP      MEASCOM           ; To common Part
;
MSTAT6    MOV      #Rmeas1,R5        ; Measure Rmeas1
          JMP      MEASCOM           ; To common Part
;
MSTAT8    MOV      #Rref1,R5         ; Measure Rref1
MEASCOM   BIS.B    #MOD,&SCFQCTL     ; Switch off FLL Modulation
          BIS      #SCG0,SR          ; Loop control off
          CLR.B    &TPE              ; TP.x to  HI-Z
          MOV.B    #B16,&TPD         ; TP.x LO (disabled!)
;
; No MCLK for ADC, Clear Flags RC2FG, RC1FG, EN1FG
;
          MOV.B    #TPSSEL1+ENB+ENA,&TPCTL
          CLR.B    &TPCNT1           ; Reset Counter LO
          CLR.B    &TPCNT2           ; Reset Counter HI
          BIS.B    R5,&TPE           ; Enable selected TP.x
;
; MCLK on, Comparator on: Intrpt for Ucm < Vth
;
          MOV.B    #TPSSEL1+TPSSEL0+ENB+ENA+EN1,&TPCTL
          BIS.B    #TPIE,&IE2        ; Enable ADC Intrpt
;
BT_RET    INC.B    MEASSTAT          ; To next Status
MSTAT0    POP      R5                ; If no activity necessary
          RETI
;
```

```
; MSTAT9: Measurements are completed, CPU is switched on,
; MSTAT is set to zero: no activity. FLL loop control on
;
MSTAT9  BIC     #CPUoff+SCG0,2(SP)  ; Stored SR on stack
        CLR.B   MEASSTAT           ; No activity
        BIC.B   #MOD,&SCFQCTL      ; Switch on FLL Modulation
        JMP     MSTAT0             ; Return
;
; End of Basic Timer Handler
;------------------------------------------------------------
; Interrupt Handler for the Analog-to-Digital Converter
; The results in TPCNT1 and TPCNT2 are stored starting at
; label ADCRESULT (result for Rref0)
;
ADC_INT PUSH    R5                 ; Save Help Register
        MOV.B   MEASSTAT,R5        ; Build offset for results
        SUB     #3,R5              ; Status for Rref0
        JN      ADC_F              ; MEASSTAT < 3: error
;
; Check  for correct result:
; If RC2FG  = 1: Overflow of the counter (Rx too high)
; If EN1    = 1: False interrupt, conversion not finished
;
        BIT.B   #RC2FG+EN1,&TPCTL ; Error?
        JNZ     ADC_RET            ; Yes, let 0h for error
        MOV.B   &TPCNT1,ADCRESULT(R5)      ; Store result
        MOV.B   &TPCNT2,ADCRESULT+1(R5)
;
ADC_RET BIC.B   #TPIE,&IE2         ; Disable ADC Intrpt
        BIC.B   #RC2FG+RC1FG+EN1FG,&TPCTL  ; Flags = 0
ADC_F   POP     R5                 ; Restore R5
        RETI
;
; End of Universal Timer/Port Module Handler
;------------------------------------------------------------
; Interrupt vectors
;
        .sect   "INT_VECT",0FFE2h
        .WORD   BT_INT             ; Basic Timer Vector
        .if     Type=310
```

```
.sect    "INT_VEC1",0FFEAh ; MSP430C31x
.else
.sect    "INT_VEC1",0FFE8h ; Others
.endif
.WORD    ADC_INT          ; Timer Port Vector (31x)
.sect    "INT_VEC2",0FFFEh
.WORD    INIT             ; Reset Vector
```

1.2 Connection of Long Sensor Lines

If it is a long distance from the MSP430C31x to the sensor (>30cm), a shielded lead between the microcomputer and the sensor is recommended. This gives protection to the ADC input. Figure 8 shows the schematic. The protection resistors (Rv/2) need to be included in the calculation and are connected in series with the sensor.

To protect the measurement against spikes, hum, and other unwanted noise (see Section 5.3, *Signal Averaging*. Here are some possibilities for the minimization of these influences.

Depending on the actual application, the omission of the two resistors (Rv/2) can give the best results. The relatively low internal resistance of the TP.2 output and the capacitor alone may get this.

If a shielded cable is not possible, a twisted cable or a three-core cable should be used. The unused wire is connected to Vss as shown in Figure 8 with Rsens2.
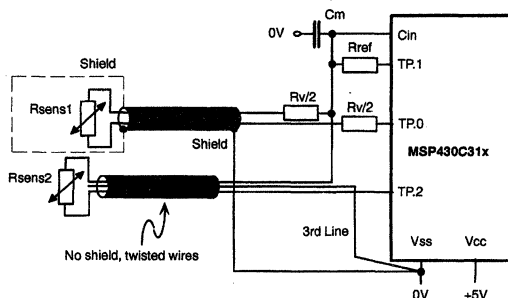
Figure 8. Connection of Long Sensor Lines

1.3 Grounding

The correct grounding is very important if ADCs with high resolution are used. There are some basic rules that need to be observed.

With the MSP430C31x and the MSP430C33x only the Vss pin exists as a common reference point.

1. Use of separate analog and digital ground planes wherever possible. No thin connections from the battery to VSS pin.

2. The Vss pin is a star point for all 0-V connections

3. Battery and capacitor are connected together at this star point. See Figure 9.

4. No common path for analog and digital signals
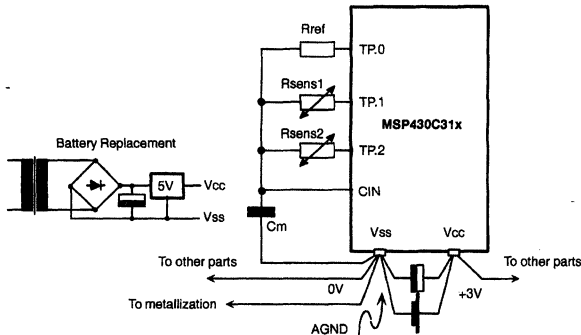
Figure 9. Grounding for the Universal Timer/Port ADC

Figure 9 also shows the use of an ac driven power supply. Its Vcc and Vss terminals are connected where the battery is normally connected. The capacitor across the MSP430 pins can be smaller when a power supply is used.

If a metallized case is used around the printed-circuit board containing the MSP430, then it is very important to connect the metallization to the ground potential (0V) of the board. Otherwise, the performance is worse than without the metallization.

1.4 Voltage Measurement With the Universal Timer Port/Module

The measurement of a restricted voltage range is also possible with the Universal Timer/Port Module. Normally a second circuit is used for this purpose.

This solution needs the least hardware effort. This measurement method delivers a very precise result, if a two-point calibration—with two voltages at the limits of the input voltage range—is used. A realized application delivers the following results:

- Accuracy for an input voltage between +8 V and +16 V better than $\pm 10^{-3}$ ($\pm 0.1\%$). A two-point calibration was used.
- Temperature deviation between $-20°C$ and $+30°C$ better than 45 ppm/°C (worst case)

1.4.1 Measurement Principle

The Universal Timer/Port Module of the MSP430 family allows the measurement of a restricted voltage range. Normally a second circuit (analog-to-digital converter) is necessary for this task. The measurement principle is explained with the circuitry shown in Figure 10.

For the voltage measurement with the MSP430x3xx family, the comparator input CMPI—with its well defined threshold voltage 0.25 × Vcc—is used and not the analog input CIN with its Schmitt-Trigger characteristic. The comparator input CMPI has different names with the different MSP430 family members. This is due to the fact that it normally uses the same pin as the highest numbered LCD select line.

The LCD pin is switched from the select function to the comparator function by a control bit located in the Universal Timer/Port Module (CPON, TPD.6, address 04Eh).

Figure 10 shows a voltage measurement circuit with two different input stages for the input voltage Vmeas:

- Input voltages with a relatively low impedance are connected directly to the input Vmeas0. The input impedance of the circuitry is approximately 10^6 Ohm (see example Figure 12).
- Input voltages with a very high impedance are connected to the non-inverting input of the operational amplifier (Vmeas1) with its input impedance of approximately 10^9 Ohm.

Only one of the two input stages described in Figure 10 can be used. If more than one input voltage is to be measured, than one of the circuits shown later is to be used.
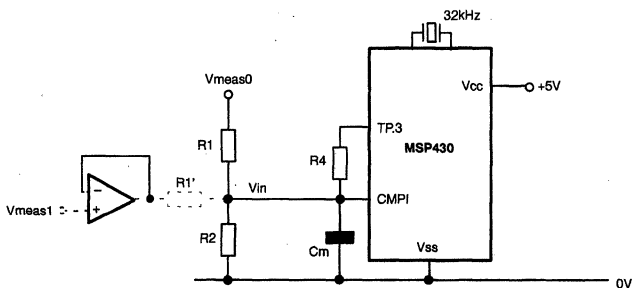
Figure 10. Voltage Measurement With the Universal Timer/Port Module

The voltage range for the input voltage Vin (seen at the input CMPI), can be measured with the circuitry shown previously is restricted to

$$V_{ref(com)\,max} < Vin \leq Vcc \qquad (1)$$

This means for a supply voltage, Vcc = +5V, voltages between 0,26 × 5 V = 1.3 V and +5 V can be measured.

With the resistor divider consisting of the two resistors R1 and R2, a nominal input voltage range for Vmeas0 results in

$$Vref \times \frac{R1 + R2}{R2} \ < \ Vmeas0 \ \leq \ Vcc \times \frac{R1 + R2}{R2} \qquad\qquad (2)$$

The sequence for the measurement of the voltage Vmeas is given in the following. The numbers used for the sequence correspond to the numbers of the Conversion States shown in Figure 11. The software is contained in this chapter.

Vin = Vmeas x R2/(R1+R2)
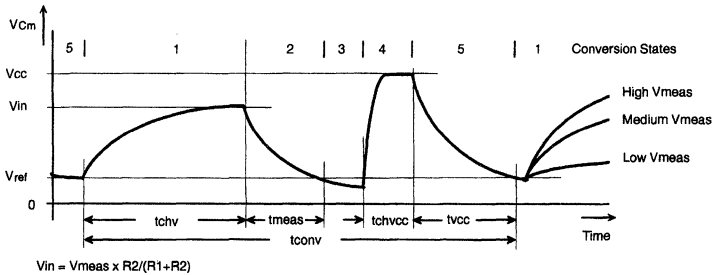
Figure 11. Voltage Measurement

1. The output TP.3 is switched to Hi-Z. The measurement capacitor Cm charges to the divided input voltage Vmeas during the time tchv between two voltage measurements.

2. The voltage measurement starts: TP.3 is switched to 0 V and discharges Cm. At the same time, the measurement of the time tmeas starts with the 16-bit counter of the Universal Timer/Port Module. When the threshold voltage Vref is reached, the time measurement is stopped automatically.

3. The measured time tmeas is stored.

4. TP.3 is switched to Vcc and charges the capacitor Cm to the supply voltage Vcc. The needed time tchvcc ranges from 5τ to 7τ dependent on the desired accuracy. $(\tau \approx R4 \times Cm)$

5. The reference measurement starts: TP.3 is switched to 0 V and discharges Cm. At the same time, the measurement of the time tvcc starts with the 16-bit counter. When the threshold voltage Vref is reached, the time measurement is stopped automatically.

6. The measured time tvcc is stored.

> **NOTE:** All formulas only show measured time intervals. The conversion of these time intervals tx into the measured counts nx can be made with the formula:

$$tx \ = \ \frac{nx}{fMCLK}$$

Where fMCLK represents the CPU frequency MCLK of the MSP430.

The voltage Vmeas can be calculated with the two measured time intervals tmeas and tvcc using the following formula:

$$Vmeas = Vcc \times \frac{R1 + R2}{R2} \times e^{\frac{tmeas - tvcc}{\tau}} \tag{3}$$

Where:

Vmeas	Input voltage to be measured	[V]
Vcc	Supply voltage of the MSP430 (used for reference)	[V]
R1,R2	Input resistor divider at input CMPI	[Ω]
tmeas	Discharge time of the divided Vmeas until Vref is reached	[s]
tvcc	Discharge time from Vcc to Vref	[s]
τ	Time constant of the discharge circuit ($\tau \approx$ R4 $\times$ Cm)	[s]
Vref	Threshold voltage of the comparator input CMPI	[V]
tconv	Time between two complete voltage measurements	[s]

To get a constant value for the value τ, an expensive, highly stable capacitor Cm is necessary. To avoid this capacitor, the value τ of the equation (3) is substituted. From the equation (4) for the discharge of the capacitor Cm

$$Vref = Vcc \times e^{-\frac{tvcc}{\tau}} \tag{4}$$

τ is calculated:

$$\tau = \frac{tvcc}{ln\dfrac{Vcc}{Vref}} \quad where \quad \frac{Vcc}{Vref} = 4 \tag{5}$$

Inserted into equation (3) this leads to:

$$Vmeas = Vcc \times \frac{R1 + R2}{R2} \times e^{\frac{tmeas - tvcc}{tvcc} \times ln\frac{Vcc}{Vref}} \tag{6}$$

With equation 6, Vmeas is calculated. Equation 6 is also used with the software example shown in Section 1.4.4.1.

For the capacitor Cm used for the voltage measurement, it is only important, that it owns a constant or a very high isolation resistance. The isolation resistor of the capacitor Cm is connected in parallel with the resistor R2 and changes the resistor ratio (e.g. due to temperature).

Equation 6 shows the dependence of the voltage measurement to the supply voltage Vcc (which is the reference), the threshold voltage Vref, the accuracy of the resistors R1 and R2 and the temperature drift of these values. To get a measurement accuracy of ±1% for Vmeas without calibration, the following basics are necessary:

- Stable supply voltage Vcc: Vcc needs to be within ±25 mV for the defined temperature range. The actual value of Vcc does not matter, if a two-point calibration is used.

- Input CMPI is used for the comparator input: the relatively good defined threshold voltage Vref (0.25 × Vcc) allows better results than the normal Schmitt-Trigger input CIN with its large tolerances for the threshold voltages.
- Temperature drift of the resistor divider maximum ±50 ppm/°C
- Sufficient charge-up times for the measurement capacitor Cm:
  - For an accuracy of one per cent approximately 5τ are necessary (e^5 = 148,41)
  - For an accuracy of 0.1% approximately 7τ are necessary (e^7 = 1096.63)

If a two-point calibration is used, the calculated values for slope and offset are stored in an external EEPROM, or if the battery is connected continuously to the MSP430 system, they are stored in the RAM.

1.4.2 *Resolution of the Measurement*

The resolution for one counter step nmeas of the voltage measurement is:

$$\frac{dVmeas}{dnmeas} = \frac{Vmeas}{\tau \times fMCLK} \approx \frac{Vmeas}{R4 \times Cm \times fMCLK} \tag{7}$$

This means for the circuit shown in Figure 12 (worst case) (Vmeas = Vmeas$_{max}$):

$$\frac{dVmeas}{dnmeas} \approx \frac{18}{47 \times 10^3 \times 47 \times 10^{-9} \times 3 \times 10^6} = 2.7 \times 10^{-3}$$

The resolution is for the worst case 2.7 mV for Vaccu = 18 V, Cm = 47 nF, R4 = 47 kΩ, f$_{MCLK}$ = 3 MHz. This equals an analog-to-digital converter with a bit length a of:

$$a = ld\frac{18V}{2,7mV} = 12,703 \tag{8}$$

(ld = log$_2$) The previous result means, the resolution of this circuit ranges between a 12-bit and a 13-bit analog-to-digital converter.

For the interesting voltage range at the input CMPI (Vref to Vcc) the non-linear characteristic of the exponential function can be substituted by a hyperbola. This method has the advantage of no time-consuming exponential function, only one division:

$$Vmeas = \frac{A}{(tmeas - tvcc) + B} + C \tag{9}$$

The values for A, B, and C can be determined by the solution of three equations or with a PC-software program like MATHCAD.

For the calculation of all of the previous formulas, the MSP430 floating point package FPP4 is ideally suited. The package contains all necessary functions like the exponential and the logarithm function. An example of its use is given in Section 2.2.4.4.1.

1.4.3 Measurement Timing

With the formulas shown previously, the worst case time interval tconv for a complete voltage measurement can be calculated.

This is the time interval that determines the highest repetition rate for a complete voltage measurement. The time interval tconv is the sum of all time intervals that are shown in Figure 11.

$$tconv = tchv + tmeas + tchvcc + tvcc \tag{10}$$

With the values that determine the time intervals of equation 10, the worst case value for the complete measurement time tconv can be calculated. The accuracy is assumed to be 1%. If the accuracy needs to be higher, then the ln100 in equation 11 must be replaced by the logarithm of the desired accuracy (e.g. by ln1000 for 0.1%). The time tmeas is assumed to be the maximum one, this means for Vin = Vcc

$$tconv = ln100 \times Cm \times R1 \| R2 + \tau \times ln\frac{Vcc}{Vref} + \tau \times ln100 + \tau \times ln\frac{Vcc}{Vref} \tag{11}$$

With the components of Figure 12 (right circuit), the time interval tconv between two complete voltage measurements is:

$$tconv = Cm \times \left(ln100 \times R1 \| R2 + 2 \times R4 \times ln4 + ln100 \times R4\right) \tag{12}$$

$$tconv = 47 \times 10^{-9} \times \left(4,6 \times 3 \times 10^{6} \| 820 \times 10^{3} + 2 \times 47 \times 10^{3} \times 1,386 + 4,6 \times 47 \times 10^{3}\right)$$

$$tconv = 0,155s$$

If an accuracy of 0.1% is used (10^{-3}), the time interval tconv gets 233 ms. With a modification of the values for R1, R2, R4, and Cm, the time interval between two complete measurements can be greatly changed. The component calculation of Figure 12 was made for a high-precision voltage measurement. The values of the components can be changed if the accuracy needs are less important.

1.4.4 Applications

This section shows how to connect different voltage sources to the Universal Timer/Port Module. Dependent on the structure of the external voltage source different hardware configurations are necessary.

1.4.4.1 Voltage Measurement

Figure 12 shows two circuits for the voltage measurement of a 12-V voltage source. The voltage reference is—like with all other circuits too—the supply voltage Vcc. The supply voltage can be between +3 V and +5 V. If the supply voltage is changed from +5V, only resistor R1 needs to be modified.

Two different circuits are shown. The input voltage Vmeas has different influence during the conversion.

- For the right hand circuit in Figure 12, the input voltage Vmeas also shows during the reference measurement with Vcc a small influence (approximately R4/R1 here)

- For the left hand circuit, the output TP.2 isolates the accumulator voltage from the reference measurement. TP.2 is always switched the same way, similar to TP.3 (0 V, +5 V, Hi-Z). After the charge of Cm the input voltage does not have an influence on the conversion.
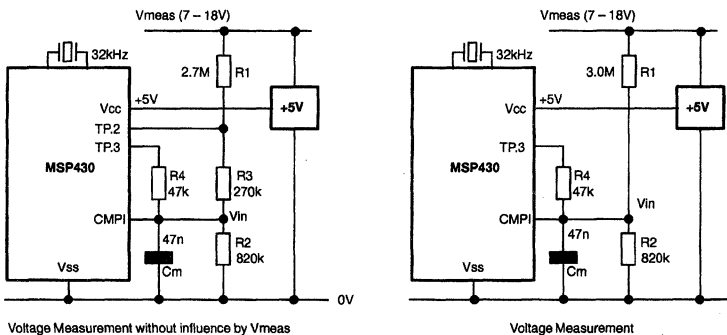
Figure 12. Voltage Measurement of a Voltage Source

Software Example: the voltage calculation is made for the right-hand circuit shown in Figure 12. The measurements for tmeas and tvcc are made as described previously. Equation 6 is implemented in software.

For the calculations the MSP430 floating point package FPP4 is used (32-bit format). All subroutine calls call FPP4 functions.

RAM word ADCref contains the 16-bit result of the Vcc measurement tvcc

RAM word ADCbatt contains the 16-bit result of the Vmeas measurement tmeas

Both time intervals are measured with MCLK cycles.

```
; Voltage measurement of Vmeas:
; Vmeas = factor * exp(((tmeas/tvcc) -1)* ln(Vcc/Vref))
; Where  factor = Vcc x (R1+R2)/R2
;
; Input:   ADCref:                 Measured reference value Vcc: tvcc
```

```
;          ADCbatt:                              Measured voltage value: tmeas
; Output:  Act. Stack                           Calculated voltage Vmeas:  @SP
;
Calc_VoltSUB     #4,SP                           ; Reserve stack
        MOV      #ADCbatt,RPARG                  ; ADC value of voltage tmeas
        CALL     #CNV_BIN16U                     ; Convert to unsigned number
        MOV      @RPRES+,x                       ; Store result to x. MSBs
        MOV      @RPRES+,x+2                      ; LSBs
        MOV      #ADCref,RPARG                   ; ADC value of Vcc tvcc
        CALL     #CNV_BIN16U                     ; Convert to unsigned number
        MOV      #x,RPRES                        ; Address tmeas
        CALL     #FLT_DIV                        ; tmeas/tvcc
        JN       Calc_Error                      ; Error
        MOV      #FLT1,RPARG                     ; Address 1.0
        CALL     #FLT_SUB                        ; (tmeas/tvcc) - 1.0
        MOV      #FLTLN4,RPARG                   ; Address Ln(Vcc/Vref)
        CALL     #FLT_MUL                        ; [(tmeas/tvcc)-1] * ln(Vcc/Vref)
        CALL     #FLT_EXP                        ; exp[(tmeas/tvcc) - 1)*ln4]
        JN       Calc_Error
        MOV      #factor,RPARG                   ; Address Vcc x (R1+R2)/R2
        CALL     #FLT_MUL                        ; Vmeas = factor * exp[...]
;
; Correction of Vmeas with calculated slope and offset
; Vmeas' = factor*exp[(tmeas/tvcc)-1]*ln4]*slope + offset
;
        MOV      #Slope,RPARG                    ; Address slope
        CALL     #FLT_MUL                        ; Vmeas * slope
        MOV      #Offset,RPARG                   ; Address offset
        CALL     #FLT_ADD                        ; Vmeas' = Vmeas * slope + offset
;
        MOV      @RPRES+,6(SP)                   ; Corrected Vmeas on Stack
        MOV      @RPRES+,8(SP)                   ; LSBs
        ADD      #4,SP                           ; Release stack
        RET                                      ; Return
;
; Calculation error (N = 1 after return): FFFF,FFFF result
;
Calc_Error MOV   #0FFFFh,6(SP)
        MOV      #0FFFFh,8(SP)
        ADD      #4,SP                           ; Correct stack
```

```
            SETN                              ; Set N-bit for error indication
            RET                               ; Return with N = 1
;
; factor describes supply voltage and resistor divider
;
factor    .float 23.292683                    ; 5V * (3.0M+820k)/820k
FLTLN4    .float 1.38629436                    ; ln Vcc/Vref.  (nom. ln 4.0)
FLT1      .float 1.0                           ; Constant 1.0
```

1.4.4.2 Current Measurement

Current that flows through a shunt resistor can also be measured with the Universal Timer/Port Module. The generated voltages are small, due to the normally low resistance of the shunt (this is because of the generated power $I^2 \times R_{shunt}$). The voltage across the shunt is not divided by a resistor divider to have the full resolution.

Figure 13 shows the circuit for the current measurement. The voltage across the shunt resistor ranges from –0.3 V to Vref (Vref is 0,25 × Vcc for the MSP430). The value –0.3 V is the most negative voltage that is allowed for an MSP430 input. To be able to also measure currents or voltages around the zero point (0 V), an inversion of the measurement method shown previously is necessary: The capacitor Cm is discharged to the voltage to be measured respective to the potential 0 V. Afterwards Cm is charged. During the charge-up, the time interval is measured until the comparator threshold Vref. is again reached. This measurement method shows a smaller resolution than the method shown previously—due to the smaller available voltage range for the charge-up—but is able to also measure voltages around the zero point.
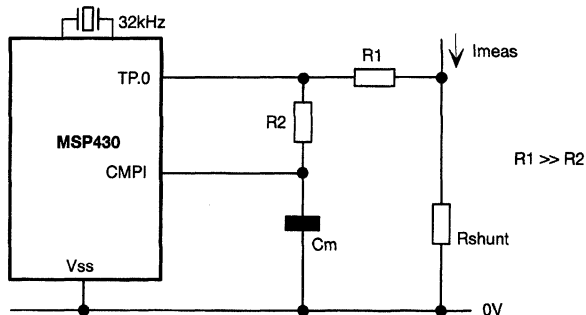
Figure 13. Circuit for the Current Measurement

Figure 14 shows the voltage at the capacitor Cm during the measurement of two currents: Vin0 is a positive one, Vin1 is a negative current. As described before, the measured time interval tvcc is used for reference purposes. The supply voltage Vcc is measured. In the previous circuitry, the voltage curve show the influence of the state of TP.0 (Vss, Vcc, Hi-Z). The equation for the calculation of the current Imeas is:

$$Imeas = \frac{1}{Rshunt} \times \left(Vcc + (Vref - Vcc) \times e^{-\frac{tmeas}{tvcc} \times \ln\left(1 - \frac{Vref}{Vcc}\right)}\right) \tag{13}$$

Equation 13 looks complicated but it can be substituted by the form

$$Imeas = a + b \times e^{\frac{tmeas}{tvcc} \times 0,2876821}$$

where a and b are constants, given by the values of the supply voltage and the shunt resistor.

Vin = Imeas x Rshunt   tchvx = tchargex   tmx = tmeasx
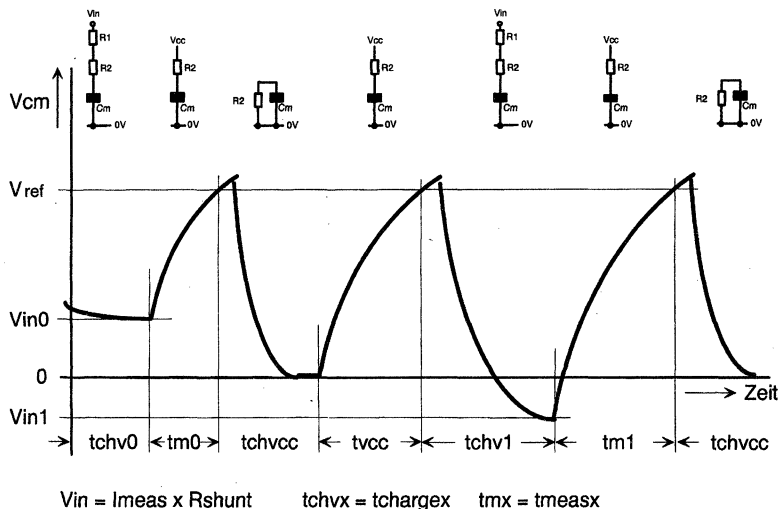
Figure 14. Current Measurement

The circuit shown in Figure 13 owns the advantage that the measurement value that represents the voltage 0 V (Vss) is known exactly. It is the value tvcc. This means no additional measurements are necessary to know the zero point (Imeas = 0) of the circuit.

The resolution of the current measurement can be calculated with equation 14. For the current Imeas, the difference for the counter steps Δnin results in:

$$\Delta nin = \tau \times fMCLK \times \left(\ln\left(1 - \frac{Vref - Imeas \times Rshunt}{Vcc - Imeas \times Rshunt}\right) - \ln\left(1 - \frac{Vref}{Vcc}\right)\right) \tag{14}$$

The first logarithm function shows the counter steps for the current Imeas, the second one shows the counter steps for a zero current.

With R2 = 47 kΩ, Cm = 33 nF (τ = 1.55 ms) and fMCLK = 3.3 MHz, equation 14 results in 1036 counter steps per volt. This means, if 1A flows through a shunt having a resistance of 0.1 Ω, then the resolution is approximately 10 mA.

1.5 Temperature Calculation Example

The temperature of an NTC sensor is calculated out of two time measurements:

- The time for the sensor Rsens—in parallel with the reference resistor Rref for linearization—to reach the lower threshold voltage VT– of the input CIN
- The time for the reference resistor alone to reach VT– of the input CIN

The measurement software is contained in Section 2.2.1.

The reference resistor Rsens is used three ways:

- As a reference for the measurement
- For the charge-up of the capacitor Cm before the measurement (eventually in parallel with the sensor for fastening)
- For the linearization of the sensor Rsens (this function defines the resistor Rref)
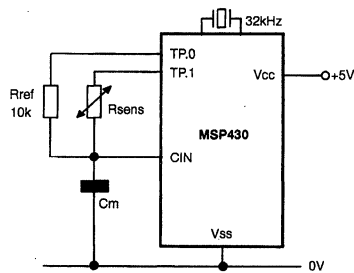
Figure 15. Temperature Measurement

EXAMPLE: the calculation of the sensor temperature for the hardware shown in Figure 15 is given in the following. The Floating Point Package is used for the calculations. The sensor characteristic is described in table NTC_TAB. For sensors with another characteristics only the sensor resistances at the necessary temperatures need to be changed.

```
; Temperature Calculation SW for Timer/Port ADC
; Input: ADCref   contains tref (MCLK cycles for Rref alone)
;        ADCsens  contains tsens (MCLK cycles for Rsens||Rref)
; Output:         Temperature on TOS (C)
;
CALC_TEMP SUB     #FPL,SP                  ; Free work space
          MOV     #ADCsens,RPARG           ; tsens  (Rref||Rsens)
          CALL    #CNV_BIN16U              ; Convert tsens to FP (NTC)
```

```
           MOV      @RPARG+,FPL+2(SP) ; To result area
           MOV      @RPARG+,FPL+4(SP)
           MOV      #ADCref,RPARG     ; tref  (Rref)
           CALL     #CNV_BIN16U       ; Convert tref to FP
           ADD      #FPL+2,RPARG      ; Point to tsens
           CALL     #FLT_DIV          ; tref/tsens (Rref/Rref||Rsens)
           MOV      @RPARG+,FPL+2(SP) ; Store to result area
           MOV      @RPARG+,FPL+4(SP)
           MOV      #NTC_TAB,R15      ; Store pointer to NTC_TAB
CTLOOP     MOV      R15,RPARG         ; Find lower margin
           CALL     #FLT_CMP          ; tref/tsens - tab-value
           JHS      CTCALC            ; Ratio > tab-value
           ADD      #FPL,R15          ; To next ratio in table
           CMP      #NTCTEND,R15      ; End of table reached?
           JLO      CTLOOP            ; No. If yes use last values
;
; Linear approximation is used between the two temperatures
;
CTCALC     PUSH     R15               ; Save pointer to lower ratio
           MOV      @SP,4(SP)         ; New work area below pointer
           MOV      R15,RPARG
           MOV      SP,RPRES
           ADD      #FPL+4,RPRES      ; Point to tref/tsens
           CALL     #FLT_SUB          ; tref/tsens - (lower ratio)
           MOV      #FLT5,RPARG       ; To 5.0
           CALL     #FLT_MUL          ; 5.0*(tref/tsens-lower ratio)
           SUB      #FPL,SP
           MOV      2*FPL(SP),RPRES   ; Address lower ratio
           MOV      RPRES,RPARG
           SUB      #FPL,RPRES        ; Address upper ratio
           CALL     #FLT_SUB          ; Delta ratio
           ADD      #FPL,RPRES        ; to 5 x ...
           CALL     #FLT_DIV          ; 5x()/delta ratio
           MOV      @RPARG+,FPL(SP)
           MOV      @RPARG+,FPL+2(SP)
           MOV      2*FPL(SP),RPARG   ; Pointer to lower ratio
           SUB      #NTC_TAB,RPARG    ; Delta start of table +90C
           RRA      RPARG             ; Divide by 4: .FLOAT length
           RRA      RPARG
           PUSH     RPARG
```

```
        MOV     SP,RPARG
        CALL    #CNV_BIN16U      ; Calculate offset (C)
        MOV     #FLT5,RPARG
        CALL    #FLT_MUL         ; x 5C
        MOV     #FLT90,RPRES     ; To +90C
        CALL    #FLT_SUB         ; 90C - lower temperature
        ADD     #FPL+2,RPARG     ; To delta within 5C ratios
        CALL    #FLT_ADD         ; minus offset = - 25 deg
        MOV     #FLT25,RPARG
        CALL    #FLT_SUB
        MOV     @SP+,2*FPL+4(SP) ; Sensor temperature to TOS
        MOV     @SP+,2*FPL+4(SP)
        ADD     #2*FPL,SP        ; Free stack
        RET                      ; Result on TOS
;
FLT5    .float  5.0              ; Delta T for table NTC_TAB
FLT90   .float  90.0             ; Temp. at table start NTC_TAB
FLT25   .float  25.0             ; offset -25 deg
;
; The NTC table contains the ratios for the temperature range
; -40C to +90C. Table values are for the ratio:
; Rref/(Rref||Rsens) = 1.0 + Rref/Rsens.    Rref = 10kOhm
; The sensor resistance Rsens is shown after the temperature
;                                           Temp    Rsens
        .float  1.0+1.0E4/0.9812E3  ; +95C: 0.9812Ω
NTC_TAB .float  1.0+1.0E4/0.1128E4  ; +90C: 1.128kΩ
        .float  1.0+1.0E4/0.1301E4  ; +85C: 1.301kΩ
        .float  1.0+1.0E4/0.1507E4  ; +80C: 1.507kΩ
        .float  1.0+1.0E4/0.1751E4  ; +75C: 1.751kΩ
        .float  1.0+1.0E4/0.2043E4  ; +70C: 2.043kΩ
        .float  1.0+1.0E4/0.2393E4  ; +65C: 2.393kΩ
        .float  1.0+1.0E4/0.2816E4  ; +60C: 2.816kΩ
        .float  1.0+1.0E4/0.3327E4  ; +55C: 3.327kΩ
        .float  1.0+1.0E4/0.3949E4  ; +50C: 3.949kΩ
        .float  1.0+1.0E4/0.4708E4  ; +45C: 4.708kΩ
        .float  1.0+1.0E4/0.5641E4  ; +40C: 5.641kΩ
        .float  1.0+1.0E4/0.6792E4  ; +35C: 6.792kΩ
        .float  1.0+1.0E4/0.8219E4  ; +30C: 8.219kΩ
        .float  1.0+1.0E4/1.0000E4  ; +25C: 10.00kΩ
        .float  1.0+1.0E4/1.223E4   ; +20C: 12.23kΩ
```

```
         .float   1.0+1.0E4/1.505E4          ; +15C: 15.05kΩ
         .float   1.0+1.0E4/1.862E4          ; +10C: 18.62kΩ
         .float   1.0+1.0E4/2.319E4          ; + 5C: 23.19kΩ
         .float   1.0+1.0E4/2.905E4          ;   0C: 29.05kΩ
         .float   1.0+1.0E4/3.663E4          ; - 5C: 36.63kΩ
         .float   1.0+1.0E4/4.650E4          ; -10C: 46.50kΩ
         .float   1.0+1.0E4/5.945E4          ; -15C: 59.45kΩ
         .float   1.0+1.0E4/7.654E4          ; -20C: 76.54kΩ
         .float   1.0+1.0E4/9.930E4          ; -25C: 99.30kΩ
         .float   1.0+1.0E4/12.98E4          ; -30C: 129.8kΩ
         .float   1.0+1.0E4/17.11E4          ; -35C: 171.1kΩ
         .float   1.0+1.0E4/22.73E4          ; -40C: 227.3kΩ
         .float   1.0+1.0E4/30.47E4          ; -45C: 304.7kΩ
NTCTEND  .float   1.0+1.0E4/41.21E4          ; -50C: 412.1kΩ
```

1.6 Measurement of the Position of a Potentiometer

The relative position of a potentiometer can be measured with the hardware shown in Figure 16. Independent of the accuracy of the potentiometer itself and the resistor Rv the relative position can be found with three measurements. The measurement of the two maximum positions allows a secure decision if these positions are reached or not. The measurements are:

1. Measurement of (Rpoti + Rv) with TP.3
2. Measurement of (Prel × Rpoti + Rv) with TP.4
3. Measurement of (Rv) with TP.5. Rv is necessary because a zero resistance cannot be measured with the Universal Timer/Port Module.
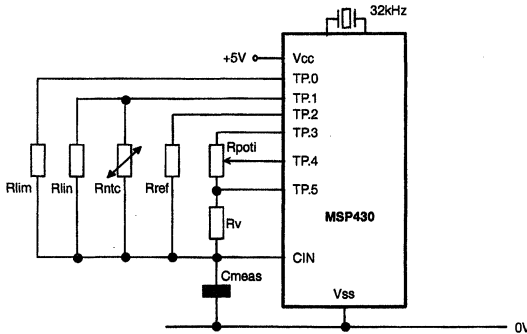
Figure 16. Measurement of a Potentiometer's Position

The formula to get the relative position Prel out of the three measurements is:

$$Prel = \frac{t4 - t5}{t3 - t5}$$

Where:

Prel Relative position of the moving arm (0 to 1)
t3   Result of the time measurement with TP.3 (Rpoti + Rv)   [s]
t4   Result of the time measurement with TP.4 (Prel × Rpoti + Rv) [s]
t5   Result of the time measurement with TP.5 (Rv)   [s]

1.7 Measurement of Sensors With Low Resistance

Figure 17 shows a hardware solution for low-resistive sensors (< 1 kΩ). With these sensors, the RDSon of the TP-ports (166 Ω to 333 Ω) plays a big role. To minimize this influence, NPN-transistors or FETs with low on-state resistance can be used for the switching of the sensors and reference resistors. The software is the same one as shown in the example in Section 1, only the switching of the TP-ports TP.0 to TP.3 needs to be changed:

• Sensor or reference resistor off: TP.x is switched to Vss

• Sensor or reference resistor on: TP.x is switched to Vcc
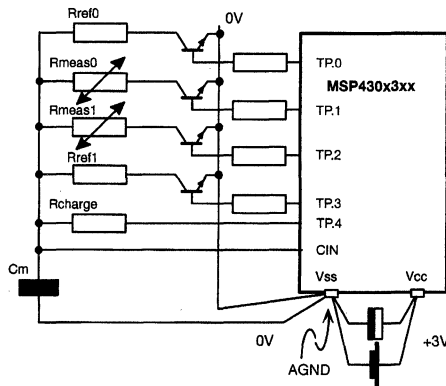
Figure 17. Hardware Schematic for Low-Resistive Sensors

Another way to eliminate the influence of the RDSon of the TP-ports is the use of a multiplexer with very low on-state resistance. The multiplexer shown in Figure 18 has a typical on-state resistance of only 5 Ω. This is small compared to the resistance of nearly all sensors.
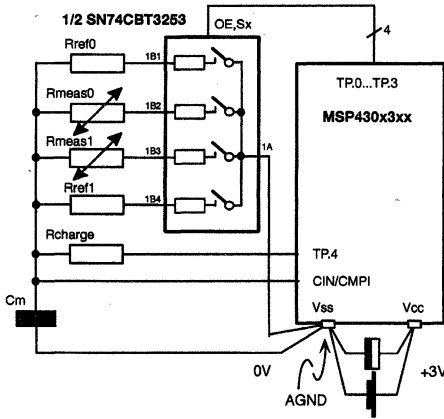
Figure 18. Solution With a Low-Resistive Multiplexer

1.8 Measurement of Capacitance

Figure 19. shows the hardware for the measurement of a capacitor Cx using a reference capacitor Cref. With the TP.1 output, the capacitor Cref is connected to Vss during the reference measurement, with TP.2 the unknown capacitor Cx is switched to Vss during the Cx measurement. The TP-ports are otherwise switched to Hi-Z.
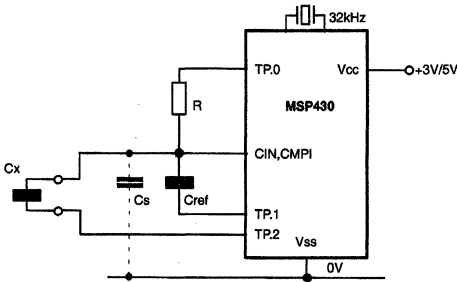
Figure 19. Measurement of a Capacitor Cx

The equation that describes the discharge curve is:

$$Vth = Vcc \times e^{-\dfrac{tref}{(Cref + Cs) \times R}} = Vcc \times e^{-\dfrac{tx}{(Cx + Cs) \times R}}$$

This leads to:

$$Cx \;=\; \frac{tx}{tref} \times \left(Cref + Cs\right) - Cs$$

Where:

Vth	Threshold voltage of the comparator	[V]
Vcc	Supply voltage of the MSP430	[V]
tref	Discharge time with the reference capacitor Cref	[s]
tx	Discharge time with the unknown capacitor Cx	[s]
tc	Charge time for the capacitors	[s]
Cx	Capacitor to be measured	[F]
Cref	Reference capacitor	[F]
Cs	Circuit capacity (may be omitted)	[F]

The voltage at the capacitors Cx and Cref during the measurement is shown in Figure 20.
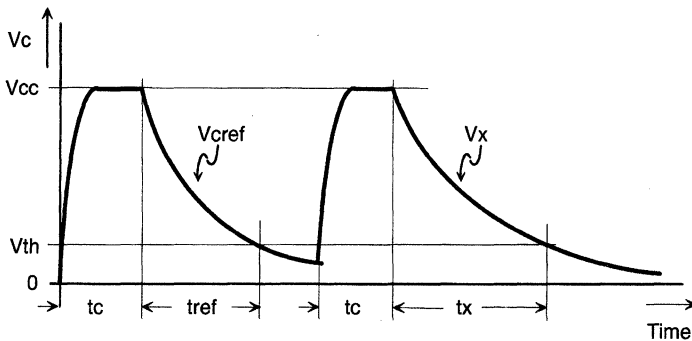
Figure 20. Timing for the Capacity Measurement

2 External Analog-To-Digital Converters

2.1 External Analog-To-Digital Converter ICs

The MSP430 can also use external ADCs. Figure 21 illustrates how to connect three different ADCs to the MSP430. This is especially important for MSP430s without an internal ADC.

Some low-cost possibilities are shown for the connection of 8-bit ADCs to the MSP430C31x and MSP430C33x.
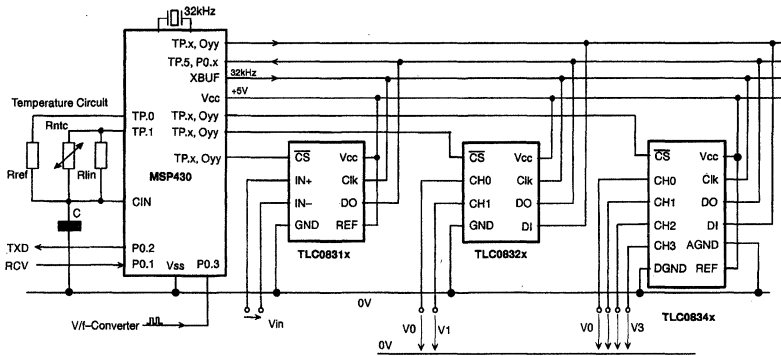
Figure 21. Analog-to-Digital Conversion With External ADCs

At the right-hand side three different 8-bit ADCs are connected to the MSP430. An 8-channel version TLC0838x with the same kind of control is also available.

Voltages higher than +5 V can be connected to the ADC inputs via resistor dividers or operational amplifiers.

Due to the interrupt capability of all Port0 inputs, voltage/frequency converters (V/f converters) can also be connected very easily. This is shown at input P0.3. For the time base one of the MSP430 timers is used.

The chapter "Electricity Meters" contains an application that uses an external 16-bit ADC.

2.2 R/2R Analog-To-Digital Converter

Due to its many I/Os the MSP430C33x can use the R/2R method, which allows strongly monotone and accurate analog-to-digital converters. Figure 22 shows an 8-bit ADC with four analog inputs. For the conversion, the successive approximation method is used. This means, that after n approximations—n equals the number of implemented bits—the conversion is complete.

If only one analog input is needed, the multiplexer can be omitted.

The MSP430C31x family can use the R/2R method also if enough outputs are available (e.g. the O-outputs if no LCD is used) (idea from F. Kirchmeier/TID).
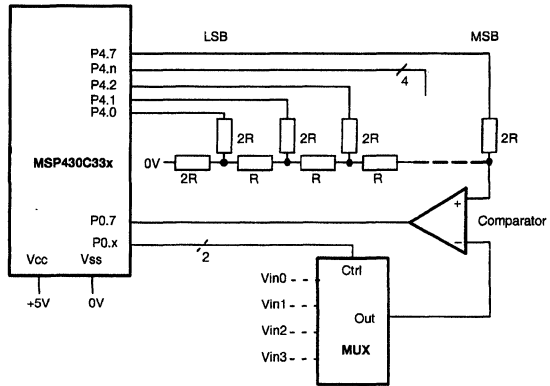
Figure 22. R/2R Method for Analog-to-Digital Conversion

Hardware Applications

3.1 I/O Port Usage

The each I/O of Port0, Port1, and Port2 has interrupt capability for the leading edge and trailing edge of an input signal. This has the following advantages:

❑ More than one interrupt input is available

❑ Eight resp. 24 external events can wake-up from Low Power Modes 3 or 4

❑ No glue logic is necessary for most applications: all inputs can be observed without the need of gates connecting them to a single interrupt input.

❑ Wake-up is possible out of any input state (high or low)

❑ Due to the edge-triggered characteristic of the interrupts, no external switch-off logic is necessary for long-lasting input signals, therefore no multiple interrupt is possible therefore.

3.1.1 General Usage

Six peripheral registers controling the activities of the I/O–Port0 are shown in Table 3–1.

Table 3–1. I/O–Port0 Registers

Register	Usage	State After Reset
Input register	Signals at I/O terminals	Signals at I/O terminals
Output register	Content of output buffer	Unchanged
Direction register	0: Input 1: Output	Reset to input direction
Interrupt flags	0: No interrupt pending 1: Interrupt pending	Set to 0
Interrupt edges	0: Low to high causes Interrupt 1: High to low causes Interrupt	Unchanged
Interrupt enable	0: Disabled 1: Enabled	Set to 0

The interrupt vectors, flags and peripheral addresses of I/O–port 0 are shown in Table 3–2.

Table 3–2. I/O–Port0 Hardware Addresses

Name	Mnemonic	Address	Contents	Vector
Input Register	P0IN	010h	P0IN.7 ... P0IN.0	—
Output Register	P0OUT	011h	P0OUT.7 ... P0OUT.0	
Direction Register	P0DIR	012h	P0DIR.7 ... P0DIR.0	—
Interrupt Flags	P0IFG	013h	P0IFG.7 ... P0IFG.2	0FFE0h
	IFG1.3	002h	P0IFG.1	0FFF8h
	IFG1.2	002h	P0IFG.0	0FFFAh
Interrupt Edges	P0IES	014h	P0IES.7 ... P0IES.0	
Interrupt Enable	P0IE	015h	P0IE.7 ... P0IE.2	
	IE1.3	000h	P0IE.1	
	IE1.2	000h	P0IE.0	

The other I/O–Ports are organized the same way except the following items:

❏ Port1 and Port2 contain eight equal I/Os, the special hardware for bits 0 and 1 is not implemented. Additionally, the ports have two function select registers, P1SEL and P2SEL .

❏ Port3 and Port4 do not have interrupt capability and registers P3IFG, P4IFG, P3IES, P4IES, P3IE and P4IE do not exist. Additionally, the ports have two function select registers P3SEL and P4SEL. These registers determine if the normal I/O–Port function is selected (PxSEL.y = 0) or if the terminal is used for a second function (PxSEL.y = 1) (see Table 3–3).

The MSP430C33x uses the two function select registers P3SEL and P4SEL for the following purposes:

❏ P3SEL.y = 1: The Timer_A I/O functions are selected (see Table 3–3)

❏ P4SEL.y = 1: The USART functions are selected (see the MSP430x33x Data Sheet, SLAS163)

Table 3–3. Timer_A I/O–Port Selection

P3SEL.y = 0	P3SEL.y = 1 Compare Mode	P3SEL.y = 1 Capture Mode
Port I/O P3.0	Port I/O P3.0	Port I/O P3.0
Port I/O P3.1	Port I/O P3.1	Port I/O P3.1
Port I/O P3.2	Timer Clock input TACLK	Timer Clock input TACLK
Port I/O P3.3	Output TA0	Capture input CCI0A
Port I/O P3.4	Output TA1	Capture input CCI1A
Port I/O P3.5	Output TA2	Capture input CCI2A
Port I/O P3.6	Output TA3	Capture input CCI3A
Port I/O P3.7	Output TA4	Capture input CCI4A

Example 3–1. Using Timer_A in the MSP430C33x System

An MSP430C33x system uses the Timer_A. The Capture/Compare Blocks are used as follows:

❑ An external clock frequency is used: input at terminal TACLK (P3.2)

❑ Capture/Compare Block 0: outputs a rectangular signal at terminal TA0 (P3.3)

❑ Capture/Compare Block 1: outputs a PWM signal at terminal TA1 (P3.4)

❑ Capture/Compare Block 2: captures the input signal at terminal TA2 (P3.5)

To initialize Port3 for the previous functions the following code line needs to be inserted into the software (for hardware definitions see Section 6.3, *Timer_A*):

```
MOV.B    #TA2+TA1+TA0+TACLK,&P3SEL ; Initialize Timer I/Os
```

Example 3–2. MSP430C33x System uses the USART Hardware for SCI (UART)

A MSP430C33x system uses the USART hardware for SCI (UART). To initialize terminal P4.7 as URXD and terminal P4.6 as UTXD the following code is used:

```
MOV.B    #URXD+UTXD,&P4SEL          ; Initialize SCI I/Os
```

Example 3–3. The I/O–ports P0.0 to P0.3 are used for input only.

The I/O–ports P0.0 to P0.3 are used for input only. Terminals P0.4 to P0.7 are outputs and initially set low. The conditions are:

```
P0.0      Every change is counted
P0.1      Any high-to-low change is counted
P0.2      Any low-to-high change is counted
P0.3      Every change is counted
;
; RAM definitions
;
          .BSS     P0_0CNT,2               ; Counter for P0.0
          .BSS     P0_1CNT,2               ; Counter for P0.1
          .BSS     P0_2CNT,2               ; Counter for P0.2
          .BSS     P0_3CNT,2               ; Counter for P0.3
;
; Initialization for Port0
;
          MOV.B    #000h,&P0OUT            ; Output register low
          MOV.B    #0F0h,&P0DIR            ; P0.4 to P0.7 outputs
          MOV.B    #00Bh,&P0IES            ; P0.0 to P0.3 Hi-Lo, P0.2 Lo-Hi
          MOV.B    #00Ch,&P0IE             ; P0.2 to P0.3 interrupt enable
          BIS.B    #00Ch,&IE1              ; P0.0 to P0.1 interrupt enable
;
; Interrupt handler for P0.0. Every change is counted
;
P0_0HAN   INC      P0_0CNT                 ; Flag is reset automatically
          XOR.B    #1,&P0IES               ; Change edge select
          RETI
;
; Interrupt handler for P0.1. Any Hi-Lo change is counted
;
P0_1HAN   INC      P0_1CNT                 ; Flag is reset automatically
          RETI
;
; Interrupt handler for P0.2 and P0.3
```

```
; The flags of all read transitions are reset. Transitions
; occurring during the interrupt routine cause interrupt after
; the RETI
;
PO_23HAN PUSH    R5                                      ; Save R5
         MOV.B   &P0FLG,R5                               ; Copy interrupt flags
         BIC.B   R5,&P0FLG                               ; Reset read flags
         BIT     #4,R5                                   ; P0.2 flag to carry
         ADC     P0_2CNT                                 ; Add carry to counter
         BIT     #8,R5                                   ; P0.3 flag to carry
         JNC     L$304
         INC     P0_3CNT                                 ; P0.3 changed
         XOR.B   #8,P0IES                                ; Change edge select
L$304    POP     R5                                      ; Restore R5
         RETI
;
         .SECT   "INT_VECT",0FFF8h
         .WORD   PO_1HAN                                 ; P0.1 INTERRUPT VECTOR;
         .WORD   PO_0HAN                                 ; P0.0 INTERRUPT VECTOR;
;
         .SECT   "INT_VECT1",0FFE0h
         .WORD   PO_23HAN                                ; P0.2/7 INTERRUPT VECTOR
```

3.1.2   Zero Crossing Detection

With the external components shown in Figure 3–1 it is possible to build a zero crossing input for the MSP430. The components shown are designed for an external voltage ac = 230 V. With a circuit capacitance (wiring, diodes) of C1 = 30 pF as shown in Figure 3–1, the following delays occur (all values for ac = 230 V, f = 50 Hz, V_{CC} = +5 V, timing is in μs):
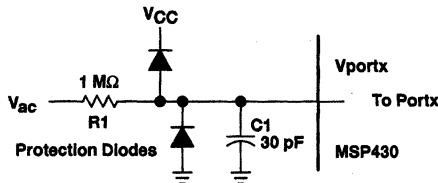
Figure 3–1. MSP430 Input for Zero-Crossing
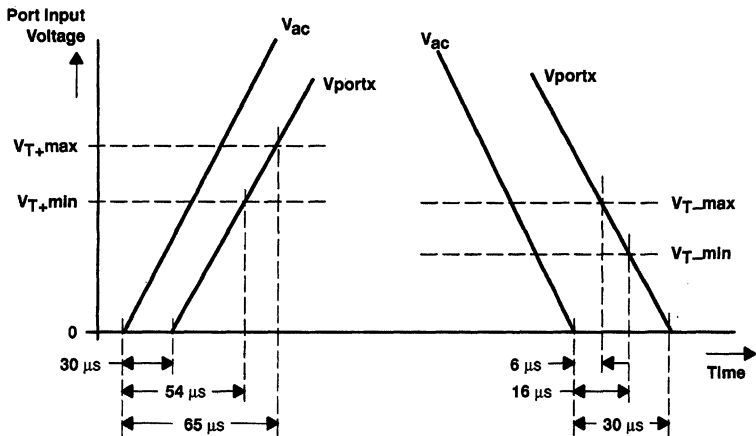
Figure 3–2. Timing for the Zero Crossing

Delay caused by RC (1 MΩ x 30 pF): 30 μs or 0.54° (same value for leading and trailing edges).

Delay caused by input thresholds:

Leading edge: 24 μs to 35 μs. (V_{T+} = 2.3 V to 3.4 V)
Trailing edge: 14 μs to 24 μs. (V_{T-} = 1.4 V to 2.3 V)

The resulting delays are:

Leading edge: 54 μs to 65 μs.
Trailing edge: 6 μs to 16 μs.

These small deviations do not play a role for 50 Hz or 60 Hz phase control applications with TRIACs. If other input conditions than 230 V and 50 Hz are used then the resulting delays can be calculated with the following formulas:

$$t_D = \frac{V_T}{S_V}; \quad S_V = \frac{d(U \times \sin\omega t)}{dt} = U \times \omega \times \cos\omega t$$

Where:

t_D	Delay time caused by the input threshold voltage	[s]
V_T	Input threshold voltage	[V]
S_V	Slope of the input voltage	[V/s]
ω	Angular frequency $2\pi f$	[1/s]
U	Peak value of the input voltage U_{ac}	[V]

For t = 0 (zero crossing time) the previous equation becomes:

$$t_D = \frac{V_T}{U \times \omega \times 1} = \frac{V_T}{U \times \omega}$$

3.1.3 Output Buffering

The outputs of the MSP430 (P0.x, P1.x, P2.x, P3.x, P4.x, Ox) have nominal internal resistances depending on the supply voltage, V_{CC}:

$V_{CC} = 3$ V: Max. 333 Ω ($\Delta V = 0.4$V max. @ 1.2mA)

$V_{CC} = 5$ V: Max. 266 Ω ($\Delta V = 0.4$V max. @ 1.5mA)

These internal resistances are non-linear and are valid only for small output currents (see the previous text). If larger currents are drawn, saturation effects will limit the output current.

These outputs are intended for driving digital inputs and gates and normally have too high an impedance level for other applications, such as the driving of relays, lines, etc. If output currents greater than the previously mentioned ones are needed then output buffering is necessary. Figure 3–3 shows some of the possibilities. The resistors shown in Figure 3–3 for the limitation of the MSP430 output current are minimum values. The application is designed for $V_{CC} = 5$ V. The values shown in brackets are for $V_{CC} = 3$ V.
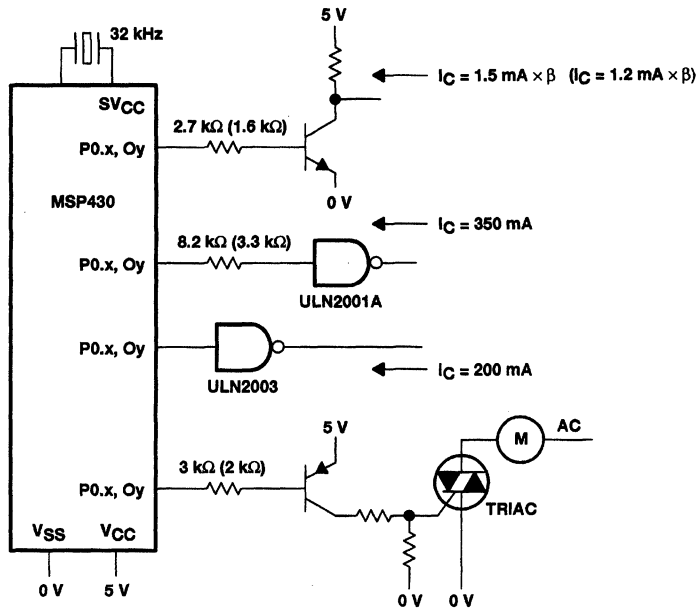
Figure 3–3. Output Buffering

3.1.4 Universal Timer/Port I/Os

If the Universal Timer/Port is not used for analog-to-digital conversion or is only partially used for this purpose, then the unused terminals are available as outputs that can be switched to high impedance. The Universal Timer/Port can be used in three different modes (see Figure 3–4):

☐ Two 8-bit timers, two inputs, one I/O, and 5 output terminals
☐ One 16-bit timer, two inputs, one I/O, and 5 output terminals
☐ An analog-to-digital converter with two to six output terminals

Ports TP0.0 to TP0.5 are completely independent of the analog-to-digital converter. Any of the ports can be used for the sensors and reference resistors.

After a power-up, the data register is set to zero and all TP0.x ports are switched to high impedance.
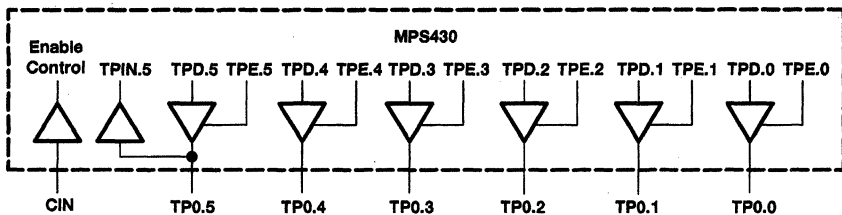
Figure 3–4. The I/O Section of the Universal Timer/Port Module

3.1.4.1 I/Os Used with the Analog-to-Digital Converter

The analog-to-digital conversion uses terminal CIN and at least two of the TP0.x terminals (one for the reference and one for the sensor to be measured); therefore up to 4 outputs are available. Bit instructions BIS.B, BIC.B, and XOR.B can only be used for the modification of the outputs. This is due to the location of the control bits in the data register TPD and data enable register TPE. The programming of the port is the same as described in the following section.

Note:

For precise ADC results, changes of the TP-ports during the measurement should be avoided. The board layout and the physical distance of the switched port determine the influence on the CIN terminal. Spikes coming from the switching of ports can change the result of a measurement. This is especially true if they occur near the crossing of the threshold voltage.

3.1.4.2 I/Os Used Without the ADC

This mode allows 5 outputs that can be switched to high impedance (TP0.0 to TP0.4) and one I/O terminal (TP0.5). Additionally, two 8-bit timers or one 16-bit timer are available. If one of the timers is used, only bit instructions BIT.B, BIS.B, BIC.B, or XOR.B can be used to operate the port. The four timer control bits are located in the data register TPD and data enable register TPE. If the MOV.B instruction is used, all the bits are affected.

Example 3–4. All Six Ports are Used as Outputs

All six ports are used as outputs. The possibilities of the port are shown in the following:

```
;
; Definitions for the Counter Port
;
TPD     .EQU    04Eh                            ; Data Register
TPE     .EQU    04Fh                            ; Data Enable Register. 1:
                                                  output enabled
TP0     .EQU    001h                            ; TP0.0 bit address
TP1     .EQU    002h                            ; TP0.1 bit address
TP2     .EQU    004h                            ; TP0.2 bit address
TP3     .EQU    008h                            ; TP0.3 bit address
TP4     .EQU    010h                            ; TP0.4 bit address
TP5     .EQU    020h                            ; TP0.5 bit address
;
; Reset all ports and switch all to output direction
;
        BIC.B   #TP0+TP1+TP2+TP3+TP4+TP5,&TPD   ; Data to low
        BIS.B   #TP0+TP1+TP2+TP3+TP4+TP5,&TPE   ; Enable outputs
;
; Toggle TP0.0 and TP0.4, set TP0.5 and TP0.2 afterwards
;
        XOR.B   #TP0+TP4,&TPD                   ; Toggle TP0.0 and TP0.4
        BIS.B   #TP5+TP2,&TPD                   ; Set TP0.5 and TP0.2
;
; Switch TP0.1 and TP0.3 to HI–Z state
;
        BIC.B   #TP1+TP3,&TPE                   ; HI–Z state for TP0.1
                                                  and TP0.3
```

3.1.5 I/O Used for Fast Serial Transfers

The combination of hardware and software, shown in the following, allows a fast serial transfer with the MSP430 family. The data line needs to be Px.0. Any other port can be used for the clock line. Any data length is possible. The LSB is transferred first. This can be easily changed by using RLC instead of RRC.

```
;
POOUT    .EQU    011h                    ; Port0 Output register
PODIR    .EQU    012h                    ; Port0 Direction register
P00      .EQU    01h                     ; Bit address of P0.0: Data
P01      .EQU    02h                     ; Bit address of P0.1: Clock
;
         MOV     DATA,R5                 ; 1st 16bit data to R5
         CALL    #SERIAL_FAST_INIT       ; 1st transfer, initialization
         MOV     DATA1,R5                ; 2nd 16bit data to R5
         CALL    #SERIAL_FAST            ; 2nd transfer, LSB to MSB
         ....                            ; aso.
;
; Initialization of the fast serial transfer: uses SERIAL_FAST too
;
SERIAL_FAST_INIT                         ; Initialization part
         BIC.B   #P00+P01,&POOUT         ; Reset P0.0 and P0.1
         BIS.B   #P00+P01,&PODIR         ; P0.0 and P0.1 to output dir.
;
; Part for 2nd and all following transfers
;
SERIAL_FAST                              ; Initialization is made
         RRC     R5                      ; LSB to carry            1 cycle
         ADDC.B  #P01,&POOUT             ; Data out, set clock      4 cycles
         BIC.B   #P00+P01,&POOUT         ; Reset data and clock     5 cycles
;
         RRC     R5                      ; LSB+1 to carry          1 cycle
         ADDC.B  #P01,&POOUT             ; Data out, set clock      4 cycle
         BIC.B   #P00+P01,&POOUT         ; Reset data and clock     5 cycles
;
         ......                          ; Output all bits the same way
;
         RRC     R5                      ; MSB to carry            1 cycle
         ADDC.B  #P01,&POOUT             ; Data out, set clock      4 cycles
         BIC.B   #P00+P01,&POOUT         ; Reset data and clock     5 cycles
         RET
;
```

Each bit needs 10 cycles for the transfer, this results in a maximum baud rate for the transfer:

$$\text{Baud rate}_{max} = \frac{\text{MCLK}}{10}$$

This means if MCLK = 1.024 MHz then the maximum baud rate is 102.4 kbaud.
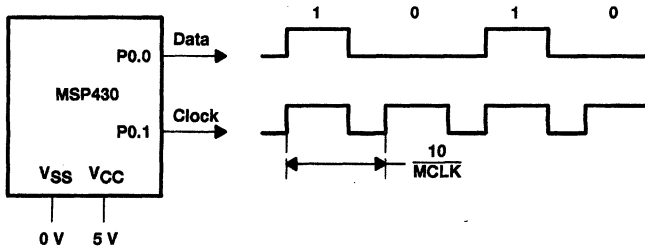
Figure 3–5. Connections for Fast Serial Transfer

3.2 Storage of Calibration Constants

Metering devices, such as electricity meters, gas meters etc., normally need to store calibration constants (offsets, slopes, limits, addresses, correction factors) for use during the measurements. Depending on the voltage supply (battery or ac), these calibration constants can be stored in the on-chip RAM or in an external EEPROM. Both methods are explained in the following sections.

3.2.1 External EPROM for Calibration Constants

The storage of calibration constants, energy values, meter numbers, and device versions in external EEPROMs may be necessary if the metering device is ac powered. This is because of the possibility of power failures.

The EEPROM is connected to the MSP430 by dedicated inputs and outputs. Three (or two) control lines are necessary for proper function:

❑ Data line SDA: an I/O port is needed for this bidirectional line. Data can be read from and written to the EEPROM on this line.

❑ Clock line SCL: any output line is sufficient for the clock line. This clock line can be used for other peripheral devices as long as no data is present on the data line during use.

❑ Supply line: if the current consumption of the idle EEPROM is too high, then switching of the EEPROM V_{CC} is needed. Three possible solutions are shown:

■ The EEPROM is connected to SV_{CC}. This is a very simple way to have the EEPROM powered off when not in use.

■ The EEPROM is switched on and off by an external PNP transistor driven by an output port.

■ The EEPROM is connected to 5 V permanently, when its power consumption is not a consideration.
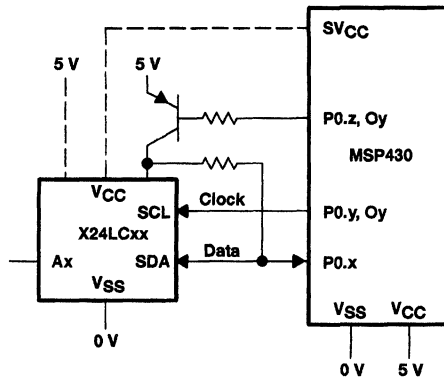
Figure 3–6. External EPROM Connections

An additional way to connect an EEPROM to the MSP430 is shown in Section 3.4, I^2C *Bus Connection*, describing the I^2C Bus.

Note:

The following example does not contain the necessary delay times between the setting and the resetting of the clock and the data bits. These delay times can be seen in the specifications of the EEPROM device. With a processor frequency of 1 MHz, each one of the control instructions needs 5 μs.

Example 3–5. External EEPROM Connections

The EEPROM, with the dedicated I/O lines, is controlled with normal I/O instructions. The SCL line is driven by O17, the SDA line is driven by P0.6. The line is driven high by a resistor and low by the output buffer.

```
P0OUT    .EQU    011h              ; Port0 Output register
P0DIR    .EQU    012h              ; Port0 Direction register
SCL      .EQU    0F0h              ; O17 controls SCL, 039h LCD Address
SDA      .EQU    040h              ; P0.6 CONTROLS SDA
LCDM     .EQU    030h              ; LCD control byte
;
; INITIALIZE I2C BUS PORTS:
; INPUT DIRECTION:   BUS LINE GETS HIGH
; OUTPUT BUFFER LOW: PREPARATION FOR LOW SIGNALS
;
```

```
        BIC.B    #SDA,&P0DIR              ; SDA TO INPUT DIRECTION
        BIS.B    #SCL,&LCDM+9            ; SET CLOCK HI
        BIC.B    #SDA,&P0OUT             ; SDA LOW IF OUTPUT

        . . .

;
; START CONDITION: SCL AND SDA ARE HIGH, SDA IS SET LOW,
; AFTERWARDS SCL GOES LO
;
        BIS.B    #SDA,&P0DIR             ; SET SDA LO (SDA GETS OUTPUT)
        BIC.B    #SCL,&LCDM+9           ; SET CLOCK LO
;
; DATA TRANSFER: OUTPUT OF A "1"
;
        BIC.B    #SDA,&P0DIR            ; SET SDA HI
        BIS.B    #SCL,&LCDM+9          ; SET CLOCK HI
        BIC.B    #SCL,&LCDM+9          ; SET CLOCK LO
;
; DATA TRANSFER: OUTPUT OF A "0"
;
        BIS.B    #SDA,&P0DIR            ; SET SDA LO
        BIS.B    #SCL,&LCDM+9          ; SET CLOCK HI
        BIC.B    #SCL,&LCDM+9          ; SET CLOCK LO
;
; STOP CONDITION: SDA IS LOW, SCL IS HI,    SDA IS SET HI
;
        BIC.B    #SDA,&P0DIR            ; SET SDA HI
        BIS.B    #SCL,&LCDM+9          ; Set SCL HI
;
```

The examples, shown in the previous text, for the different conditions can be implemented into a subroutine, which outputs the contents of a register. This shortens the necessary ROM code significantly. Instead of line Ox for the SCL line another I/O port P0.x can be used. See Section 3.4, *I²C Bus Connection*, for more details of such a subroutine.

3.2.2 Internal RAM for Calibration Constants

The internal RAM can be used for storage of the calibration constants, if a permanently connected battery is used for the power supply. The use of low power mode 3 or 4 is necessary for these kinds of applications and can get battery life times reaching 8 to 12 years.

3.3 M-Bus Connection

The MSP430 connection to the M-Bus (metering bus) is shown in Figure 3–7. Three supply modes are possible when used with the TSS721:

❑ Remote supply: The MSP430 is fully powered from the TSS721

❑ Remote supply/battery support: The MSP430 power is supplied normally from the TSS721. If this power source fails, a battery is used for backup power to the MSP430

❑ Battery Supply: The MSP430 is always supplied from a battery.

All these operating modes are described in detail in the *TSS721 M-Bus Transceiver Applications Book.*
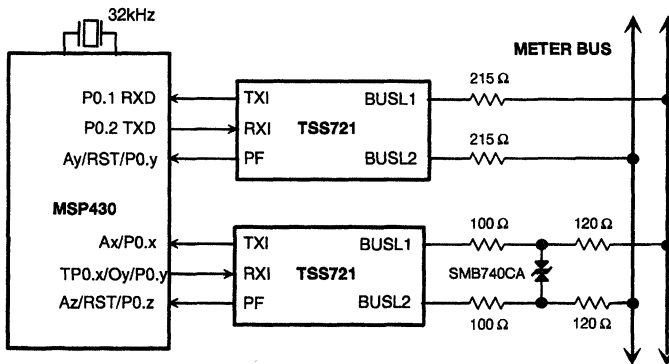
Figure 3–7. TSS721 Connections to the MSP430

Two different TSS721 connections are shown in Figure 3–7:

❑ If the 8-bit interval timer with its UART is used then the upper connection is necessary. TXI or TX are connected to RXD (P0.1) and RXI or RX is connected to TXD (P0.2).

❑ If a strictly software UART or an individual protocol is used, then any input and output combination can be used

The second connection uses a proven hardware for environments with strong EMV conditions. The 40-V suppressor diode gives the best results with this configuration.

For more details, see Section 3.8, *Power Supplies for MSP430 Systems.*

3.4 I²C Bus Connection

If more than one device is to be connected to the I²C-Bus, then two I/O ports are needed for the control of the I²C peripherals. This is needed to switch SDA and SCL to a high-impedance state.

Figure 3–8 shows the connection of three I²C peripherals to the MSP430:

☐ An EEPROM with 128x8-bit data
☐ An EEPROM with 2048x8-bit data
☐ An 8-bit DAC/ADC

The bus lines are driven high by the Rp resistors (P0.x is switched to input direction) and low by the output ports itself (P0.x is switched to output direction).
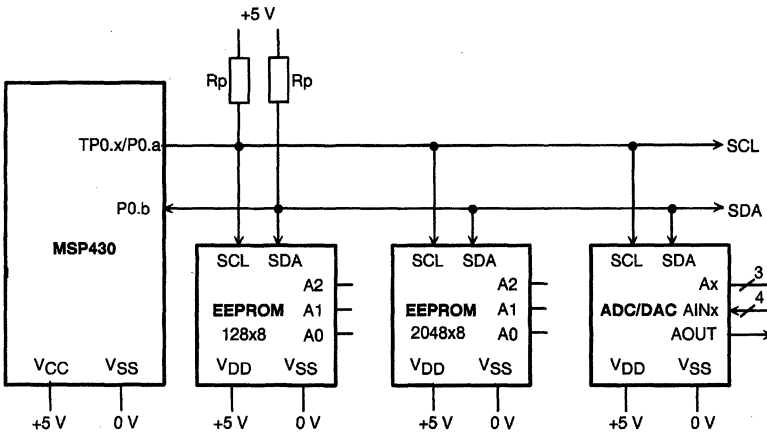
Figure 3–8. I²C Bus Connections

The following software example shows a complete I²C handler. It is designed for an EEPROM 24C65 with the following values:

☐ MCLK Frequency: 3.8 MHz
☐ Address Length: 13 bits
☐ Device Code: selectable by Code definition

```
; I2C-Handler: Transmission of 8-bit data via the I2C-bus
; Author: Christian Hernitscheck TID
;
; Definitions for I2C Bus
```

```
;
SCL       .EQU    040h             ; P0.6 controls SCL line (pull-up)
SDA       .EQU    080h             ; P0.7 controls SDA line (pull-up)
SCLIN     .EQU    010h             ; P0 input register P0IN
SDAIN     .EQU    010h             ; P0 input register P0IN
SCLDAT    .EQU    011h             ; P0OUT register address
SDADAT    .EQU    011h             ; P0 output direction register P0DIR
SCLEN     .EQU    012h             ; P0DIR register address
SDAEN     .EQU    012h             ; P0 direction register
Code      .equ    0A0h             ; Device Code 10 (24C65)
;
Address   .EQU    0200h            ; address pointer for EEPROM
I2CData   .EQU    0202h            ; used for I2C protocoll
;
; Register definitions
;
Data      .EQU    R5
Count     .EQU    R6
Mask      .EQU    R7
;
; Initialization in main program
;
          BIC.B   #SCL+SDA,&SDAEN  ; SCL and SDA to input direction
          BIC.B   #SCL+SDA,&SDADAT ; SCL and SDA output buffer lo
          ...                      ; Continue
;
; Subroutines of the I2C-Handler
;
; Write 8-bit data <data> into EEPROM address <address>:
;
; Call  MOV     <address>,Address  ; EEPROM data address
;       MOV.B   <data>,I2CData     ; 8-bit data
;       CALL    #I2C_Write         ; Call subroutine
;       JC      Error              ; Acknowledge error
;       JN      Error              ; Arbitration error
;       ...                        ; Continue program
```

```
;
; Read 8-bit data from EEPROM address <address>:
;
;           MOV       <address>,Address          ; EEPROM data address
;           CALL      #I2C_Read                  ; Call subroutine
;           JNC       Error                      ; Acknowledge error
;           JN        Error                      ; Arbitration error
;           ...                                  ; Data in I2CData (byte)
;
; Status Bits on return:
;
; C: Acknowledge Bit
; N: 1: Arbitration Error        0: no error
;
; Used Registers: R5 = Data   (pushed onto Stack)
;                 R6 = Count (pushed onto Stack)
;                 R7 = Mask   (pushed onto Stack)
;
; Used RAM:              Address         0200h
;                        I2CData         0202h        .
;                        I2CData+1       0203h
;                        I2CData+2       0204h
;                        I2CData+3       0205h
;                        I2CData+4       0206h
;
;
I2C_Write       MOV.B   I2CData,I2CData+3 ; Data to be written to EEPROM
        CALL    #ControlByte              ; Control byte
        MOV.B   Address+1,I2CData+1       ; Hi byte of EEPROM address
        AND.B   #01Fh,I2CData+1           ; Delete A2, A1 and A0 bits
        MOV.B   Address,I2CData+2 ·       ; Lo byte of EEPROM address
        JMP     I2C                       ; To common part
;
I2C_Read CALL   #ControlByte              ; I2CData   = Control byte
        MOV.B   Address+1,I2CData+1       ; Hi byte of EEPROM address
        AND.B   #01Fh,I2CData+1           ; Delete A2, A1 and A0 bits
```

```
              MOV.B    Address,I2CData+2       ; Lo byte of EEPROM address
              MOV.B    I2CData,I2CData+4       ; Control byte 2
              BIS.B    #01h,I2CData+4           , To common part
;
; Common I2C-Handler
;
I2C        PUSH     Count                   ; Save registers
              PUSH     Data
              PUSH     Mask
              CLR      Count
              BIS.B    #SDA,&SDAEN             ; Start Condition: set SDA Lo
              MOV.B    I2CData,Data           ; Send slave address and RW bit
              CALL     #I2C_Send
              JC       I2C_Stop
;                                              Write or Read?
              BIT.B    #01h,I2CData+4         ;   _
              JC       I2C_SubRead            ; R/W bit is 1: read
;            _
; Write data (R/W = 0)
;
I2C_Data INC       Count
              CMP      #4,Count
              JEQ      I2C_Stop
              CALL     #I2C_Send
              JNC      I2C_Data
                                             ; Stop Condition:
I2C_Stop BIS.B    #SCL,&SCLEN            ; SCL = 'L'
              BIS.B    #SDA,&SDAEN            ; SDA = 'L'
              NOP                             ; Delay 7 cycles
              NOP
              NOP
              NOP
              NOP
              NOP
              NOP
              BIC.B    #SCL,&SCLEN           ; SCL = 'H'
```

```
          CALL      #NOP9                    ; Delay 9 cycles
          BIC.B     #SDA,&SDAEN              ; SDA = 'H'
          CLRN                               ; Reset error flags
;
I2C_End   POP       Mask                     ; Restore registers
          POP       Data
          POP       Count
          RET                                ; Carry info valid
;
; Read data (R/W = 1)
;
I2C_SubRead INC     Count
          CMP       #3,Count
          JEQ       I2C_SubRead1
          CALL      #I2C_Send
          JC        I2C_Stop
          JMP       I2C_SubRead
I2C_SubRead1 BIS.B  #SCL,&SCLEN              ; SCL='L'
          CALL      #NOP9
          BIC.B     #SCL,&SCLEN              ; SCL='H'
          NOP
          NOP
          NOP
          NOP
          NOP                                ; Start condition:
          BIS.B     #SDA,&SDAEN              ; SCL='H', SDA='H' => 'L'
          MOV.B     I2CData+4,Data
          CALL      #I2C_Send                ; Send Control Byte
          JC        I2C_Stop
          BIS.B     #SCL,&SCLEN              ; SCL = 'L'
          BIC.B     #SDA,SDAEN               ; SDA = Input
          CLR       I2CData
          MOV       #8,Count                 ; Read 8 bits
;
I2C_Read1           BIC.B   #SCL,&SCLEN      ; SCL = 'H'
          BIT.B     #SDA,&SDAIN              ; Read data to carry
```

```
          RLC.B    I2CData               ; Store received Bit
          NOP
          NOP
          BIS.B    #SCL,&SCLEN           ; SCL = 'L'
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          DEC      Count
          JNZ      I2C_Read1
;
          CALL     #I2C_Ackn             ; Test acknowledge bit to C
          JMP      I2C_Stop
;
; Send byte
;
I2C_Send MOV.B    #80h,Mask              ; Bit mask: MSB first
;
I2C_Send1         BIT.B   Mask,I2CData(Count)   ; Info bit -> Carry
          JC       I2C_Send2
          BIS.B    #SCL,&SCLEN           ; Info is 0: SCL = 'L'
          BIS.B    #SDA,&SDAEN           ; SDA = 'L'
          CALL     #NOP9
          BIC.B    #SCL,&SCLEN           ; SCL = 'H'
          JMP      I2C_Send3
I2C_Send2         BIS.B   #SCL,&SCLEN    ; Info is 1: SCL = 'L'
          BIC.B    #SDA,&SDAEN           ; SDA = 'H'
          CALL     #NOP9
          BIC.B    #SCL,&SCLEN           ; SCL = 'H'
          BIT.B    #SDA,SDAIN            ; Arbitration
          JNC      Error_Arbit
;
I2C_Send3         CLRC
          RRC.B    Mask                  ; Next address bit
```

```
                NOP
                NOP
                NOP
                JNC      I2C_Send1                 ; No Carry: continue
;
I2C_Ackn NOP
                NOP
                BIS.B    #SCL,&SCLEN               ; SCL = 'L' Acknowledge Bit
                BIC.B    #SDA,&SDAEN               ; SDA = 'H'
                CALL     #NOP8
                BIC.B    #SCL,&SCLEN               ; SCL = 'H'
                BIT.B    #SDA,&SDAIN               ; Read data to carry
                RET                                ; (acknowledge bit)
;
; I2C-Bus Error
Error_Arbit ADD  #2,SP                             ; Remove return address
                SETN                               ; Set arbitration error
                JMP      I2C_End
;
; Build control byte
;
ControlByte CLR  I2CData
                MOV.B    Address+1,I2CData          ; Hi byte of EEPROM address
                RRC      I2CData                    ; Shift MSBs to bits 3..1
                RRC      I2CData
                RRC      I2CData
                RRC      I2CData
                AND.B    #0Eh,I2CData               ; A2, A1 and A0
                ADD.B    #Code,I2CData              ; Add device code (24C65)
                RET
;
; Delay subroutine. Slows down I2C Bus speed to spec
;
NOP9     NOP                                       ; 9 cycles delay
NOP8     RET                                       ; 8 cycles delay
```

3.5 Hardware Optimization

The MSP430 permits the use of unused analog inputs (A7 to A0) and segment lines (S29 to S2) for inputs and outputs, respectively. The following two sections explain in detail how to program and use these inputs and outputs.

3.5.1 Use of Unused Analog Inputs

Unused analog-to-digital converter (ADC) inputs can be used as digital inputs or, with some restrictions, as digital outputs.

3.5.1.1 Analog Inputs Used for Digital Inputs

Any ADC input A7 to A0 can be used as a digital input. It only needs to be programmed (for example, during the initialization) for this function. Three things are important if this feature is used:

❏ Any activity at these digital inputs has to be stopped during ongoing sensitive ADC measurements. This activity will cause noise, which invalidates the ADC results. Activity in this case means:

■ No change of the AEN register (switching between digital and analog mode)

■ No input change at the digital ADC inputs (this rarely allows changing signals at these inputs).

❏ All bits that are switched to ADC inputs will read zero when read. Therefore, it is not necessary to clear them with software after reading.

❏ Not all analog inputs are implemented in a given device

Example 3–6. A0 – A4 are used as ADC Inputs and A5 – A7 as Digital Inputs

```
AIN      .EQU     0110h            ; Address DIGITAL INPUT REGISTER
AEN      .EQU     0112h            ; Address DIGITAL INPUT ENABLE REG.
A7EN     .EQU     080h             ; Bits in Dig. Input Enable Reg.:
A6EN     .EQU     040h             ; 0: ADC    1: Digital Input
A5EN     .EQU     020h             ;
; INITIALIZATION: A7 TO A5 ARE SWITCHED TO DIGITAL INPUTS
; A4 TO A0 ARE USED AS ANALOG INPUTS
;
         MOV      #A7EN+A6EN+A5EN,&AEN    ; A7 TO A5 DIGITAL MODE
         . . .
;
```

```
; NORMAL PROGRAM EXECUTION:
; CHECK IF A7 OR A5 ARE HIGH. IF YES: JUMP TO LABEL L$100
;
        BIT     #A7EN+A5EN,&AIN         ; A7 .OR. A5 HI?
        JNZ     L$100                   ; YES
                ...                     ; NO, CONTINUE
;
; CHECK IF ALL DIG. INPUTS A7 TO A5 ARE LOW. IF YES: Go to L$200
;
        TST     &AIN                    ; A7 TO A5 LO?
        JZ      L$200                   ; YES, (ANALOG INPUTS READ ZERO)
```

3.5.1.2 Analog Inputs Used as Digital Outputs

If outputs are needed then the unused ADC inputs with the current source connection can be used with the following restrictions:

❏ Only one ADC input can be high at a given time (1 out of n principle)

❏ Only the ADC inputs A0 to A3 are usable (only they are connected to the current source)

❏ The outputs can go high only while the ADC is not using the current source.

❏ The output current is directly related to the supply voltage, V_{CC}.

❏ The output voltage is only about 50% of the supply voltage, V_{CC}. Logic levels have to be carefully monitored. A transistor stage might be necessary (if not there already, e.g. for a relay).

❏ The output current is the current of the current source. Again, logic levels have to be carefully monitored. The pull-down resistor has to be big enough to allow the maximum output level.

The example in Figure 3–9 shows the ADC using inputs A0 and A1 as digital outputs driving two stages; a transistor stage (energy pulse, e.g. with an electricity meter) and a 3.3-V gate (3.3 V ensures that the input levels are sufficient).
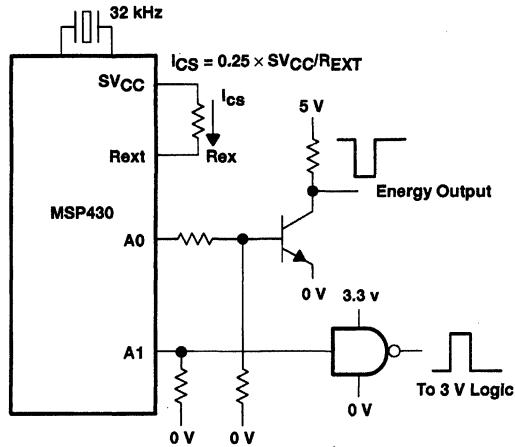
Figure 3–9. Unused ADC Inputs Used as Outputs

Example 3–7. Controlling Two Inputs as Outputs

To control the two outputs shown in Figure 3–9, the following software program is necessary:

```
ACTL    .EQU    0114h           ; ADC CONTROL REGISTER ACTL

VREF    .EQU    02h             ; 0: Ext. Reference    1: SVCC ON

A0      .EQU    0000h           ; AD INPUT SELECT A0

A1      .EQU    0004h           ;                              A1

CSA0    .EQU    0000h           ; CURRENT SOURCE TO A0

CSA1    .EQU    0040h           ;                              A1

CSOFF   .EQU    0100h           ; CURRENT SOURCE OFF BIT

;
; SET A0 HI FOR 3ms: SELECT A0 FOR CURRENT SOURCE AND INPUT
        MOV     #VREF+A0+CSA0,&ACTL     ; PD = 0, SVCC = on

        CALL    #WAIT3MS        ; WAIT 3ms

        BIS     #CSOFF,&ACTL    ; CURRENT SOURCE OFF;

;
; SET A1 HI FOR 3ms: SELECT A1 FOR CURRENT SOURCE AND INPUT
        MOV     #VREF+A1+CSA1,&ACTL     ; PD = 0, SVCC = on

        CALL    #WAIT3MS        ; WAIT 3ms
```

```
BIS      #CSOFF,&ACTL              ; CURRENT SOURCE OFF
```

3.5.2 Use of Unused Segment Lines for Digital Outputs

The LCD driver of the MSP430 provides additional digital outputs, if the segment lines are not used. Up to 28 digital outputs are possible by the hardware design, but not all of them will be implemented on a given chip. The addressing scheme for the digital outputs O2 to O29 is as illustrated in Table 3–4.

Table 3–4 shows the dependence of the segment/output lines on the 3-bit value LCDP. When LCDP = 7, all the lines are switched to LCD Mode (segment lines). Only groups of four segment lines can be switched to digital output mode. LCDP is set to zero by the PUC (O6 to O29 are in use).

Note:

Table 3–4 shows the digit environment for a 4-MUX LCD display. The outputs O0 and O1 are not available: S0 and S1 are always implemented as LCD outputs. (digit 1).

The digital outputs Ox have to be addressed with all four bits. This means that 0h and 0Fh are to be used for the control of one output.

Only byte addressing is allowed for the addressing of the LCD controller bytes. Except for S0 and S1, the PUC switches the LCD outputs to the digital output mode (LCDP = 0).

Table 3–4. LCD and Output Configuration

Address	7	6	5	4	3	2	1	0	Digit Nr.	LCDP
03Fh		O29				O28			Digit 15	6 to 0
03Eh		O27				O26			Digit 14	6 to 0
03Dh		O25				O24			Digit 13	5 to 0
03Ch		O23				O22			Digit 12	5 to 0
03Bh		O21				O20			Digit 11	4 to 0 S20/S21
03Ah		O19				O18			Digit 10	4 to 0
039h		O17				O16			Digit 9	3 to 0
038h		O15				O14			Digit 8	3 to 0
037h		O13				O12			Digit 7	2 to 0
036h		O11				O10			Digit 6	2 to 0 S10/S11
035h		O09				O08			Digit 5	1 to 0
034h		O07				O06			Digit 4	1 to 0
033h		O05				O04			Digit 3	0
032h		O03				O02			Digit 2	0
031h	h	g	f	e	d	c	b	a	Digit 1	S0/S1

Example 3–8. S0 to S13 Drive a 4-MUX LCD

S0 to S13 drive a 4-MUX LCD (7 digits). O14 to O17 are set as digital outputs.

```
; LCD Driver definitions:
;
LCDM    .EQU    030h            ; ADDRESS LCD CONTROL BYTE
LCDM0   .EQU    001h            ; 0: LCD off      1: LCD on
LCDM1   .EQU    002h            ; 0: high         1: low Impedance
MUX     .EQU    004h            ; MUX: static, 2MUX, 3MUX, 4MUX
LCDP    .EQU    020h            ; Segment/Output Definition LCDM7/6/5
O14     .EQU    00Fh            ; O14 Control Definition
O15     .EQU    0F0h            ; O15
O16     .EQU    00Fh            ; O16
O17     .EQU    0F0h            ; O17
;
; INITIALIZATION: DISPLAY ON:     LCDM0 = 1
;                 HI IMPEDANCE    LCDM1 = 0
;                 4MUX:                   LCDM4/3/2 = 7
;       O14 TO O17 ARE OUTPUTS:           LCDM7/6/5 = 3
;
        MOV.B   #(LCDP*3)+(MUX*7)+LCDM0,&LCDM      ; INIT LCD
        ...
;
; NORMAL PROGRAM EXECUTION:
; SOME EXAMPLES HOW TO MODIFY THE DIGITAL OUTPUTS O14 TO O17:
;
        BIS.B   #O14,&LCDM+8            ; SET O14, O15 UNCHANGED
        BIC.B   #O15+O14,&LCDM+8       ; RESET O14 AND O15
        MOV.B   #O15+O14,&LCDM+8       ; SET O14 AND O15
        MOV.B   #O17,&LCDM+9            ; RESET O16, SET O17
        XOR.B   #O17,&LCDM+9            ; TOGGLE O17, O16 STAYS UNCHANGED
```

3.6 Digital-to-Analog Converters

The MSP430 does not contain a digital-to-analog converter (DAC) on-chip, but it is relatively simple to implement the DAC function. Five different solutions with distinct hardware and software requirements are shown in the following:

❑ The R/2R method
❑ The weighted-resistors method
❑ Integrated DACs connected to the I²C Bus
❑ Pulse width modulation (PWM) with the universal timer/port module
❑ Pulse width modulation with Timer A

3.6.1 R/2R Method

With a CMOS shift register or digital outputs, a DAC can be built for any bit length. The outputs Qx of the shift register switch the 2R-resistors to 0 V or V_{CC} according to the digital input. The voltage at the non-inverting input and also at the output voltage V_{out} of the operational amplifier is:

$$_n V_{out} = \frac{k}{2} \times V_{CC}$$

Where:
  k   Value of the digital input word with n bits length
  n   Number of Q outputs, maximum length of input word
  V_{CC} Supply voltage

Signed output is possible by level shifting or by splitting of the power supply ($+V_{CC}/2$ and $-V_{CC}/2$). With split power supplies the voltage at the output of the operational amplifier is:

$$Vout = \frac{k}{2^n} \times V_{CC} - \frac{V_{CC}}{2} = V_{CC} \left(\frac{k}{2^n} - \frac{1}{2} \right)$$

Advantages of the R/2R Method

❑ Only two different resistors are necessary (R and 2R)
❑ Absolute monotony over the complete output range
❑ Internal impedance independent of the digital value: impedance is always R
❑ Expandable to any bit length by the adding of shift registers
❑ With only three digital outputs (Ox, TP0.x, Portx), an inexpensive solution is possible.

If enough digital outputs are available in an application, then the shift regis-ter(s) can be omitted. The outputs QA to QH of Figure 3–10 are substituted by O outputs, ports or TP outputs of the MSP430.
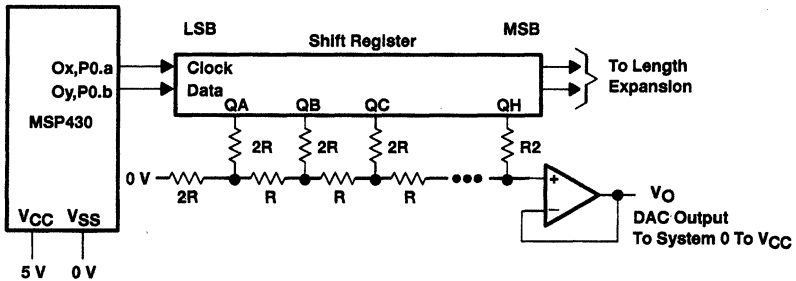
Figure 3–10. R/2R Method for Digital-to-Analog Conversion

3.6.2 Weighted Resistors Method

The simplest digital-to-analog conversion method, only (n+3) resistors and an operational amplifier are required for an n-bit DAC. This method is used when the DAC performance can be low.

The example shown in Figure 3–11 delivers 2^{n+1} different output voltage steps. They can be seen as signed if the voltage $V_{CC}/2$ is seen as a zero point. The output voltage Vout of this DAC is:

$$V_{out} = V_{ninv} - \sum I_n \times R = \frac{V_{CC}}{2} \times \left(1 + \left(a \times 2^{-1} + b \times 2^{-2} + c \times 2^{-3} \ldots + x \times 2^{-(n+1)}\right)\right)$$

Where:

V_{out} Output voltage of the DAC

V_{ninv} Voltage at the noninverting input of the operational amplifier ($V_{CC}/2$)

V_{CC} Supply voltage of the MSP430 and periphery

R   Normalized resistor used with the DAC

a...x   Multiplication factors for the weighted resistors R to $2^n \times R$:

+1 if port is switched to V_{SS}

0 if port is switched to input direction (high impedance)

−1 if port is switched to V_{CC}

Normally all of the ports are switched to the same potential (VSS or V_{CC}) or are disabled. This allows signed output voltages referenced to $V_{CC}/2$.

☐ Advantage of the Weighted Resistor-Method: Simplicity

☐ Disadvantage: Monotony not possible due to resistor tolerances
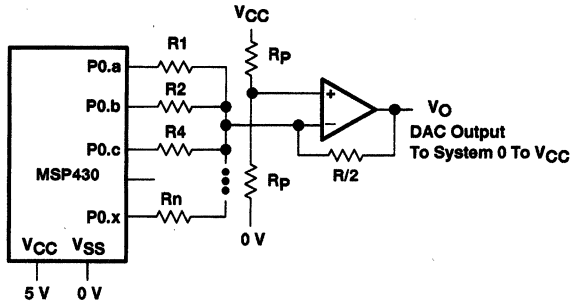
Figure 3–11. Weighted Resistors Method for Digital-to-Analog Conversion

3.6.3 Digital-to-Analog Converters Connected Via the I2C Bus

Figure 3–12 shows two different DACs that are connected to the MSP430 via the I2C Bus:

☐ A single output 8-bit DAC (with additional 4 ADC inputs); one analog output, AOUT, is provided.

☐ An octuple 6-bit DAC; eight analog outputs, DAC0 to DAC7, are available for the system

The generic software program to handle these devices is contained in the Section 3.4, *I²C Bus Connection*, explaining the I²C-Bus.
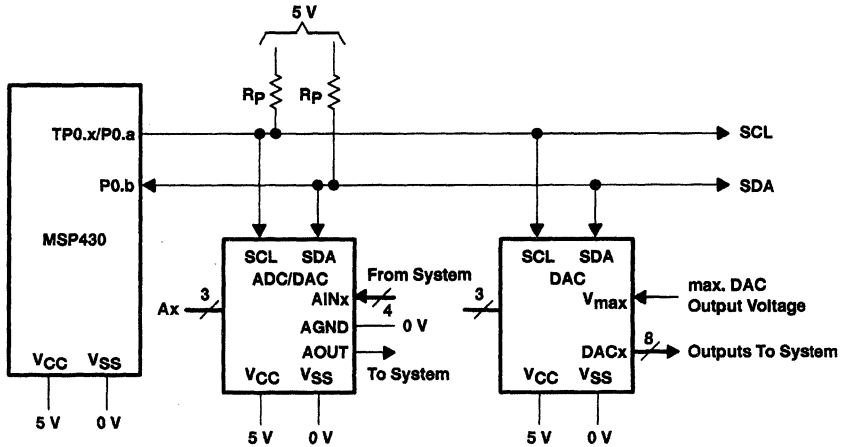
Figure 3–12. I²C-Bus for Digital-to-Analog Converter Connection

3.6.4 PWM DAC With the Universal Timer/Port Module

The two timers contained in the universal timer/port module can be used for one or two independent PWM generators. The ACLK frequency is used for the timing of these PWMs. The basic timer determines the period of the PWM signals. Its interrupt handler sets the programmed outputs and loads the two timer registers TPCNT2 and TPCNT1 with the negated pulse length values (see Table 3–5). The universal timer/port module terminates the pulses. Its interrupt handler resets the outputs when the counters, TPCNTx, overflow from 0FFh to 00h. The length of one step is always 1/ACLK, which is 30.51758 µs if a 32.768 kHz crystal is used.

Table 3–5 shows the necessary basic timer frequency , which is dependent on the PWM resolution used.

Table 3–5. Resolution of the PWM-DAC

Resolution Bits	Resolution Steps	Basic Timer Frequency
8	256	128
7	128	256
6	64	512
5	32	1024

Table 3–6 shows the values to be written into timer register TPCNT1 or TPCNT2 to get a certain PWM output value (related to V_{CC}); it is the desired value subtracted from the resolution value. The PWM switch (a byte in RAM) determines if the output is enabled (1) or disabled (0).

Table 3–6. Register Values for the PWM-DAC

PWM Output (Relative to V_{CC})	TPCNTx Value 256 Steps	TPCNTx Value 128 Steps	TPCNTx Value 64 Steps	TPCNTx Value 32 Steps	PWM Switch
0	x	x	x	x	0
0.25	C0h	E0h	F0h	F8h	1
0.50	80h	C0h	E0h	F0h	1
0.75	40h	A0h	D0h	E8h	1
1.00	00h	80h	C0h	E0h	1

Note:

The interrupt latency time plays an important role for this kind of PWM generation. Real time programming is necessary. Therefore, the first instruction of each interrupt handler must be the EINT instruction.

Example 3–9. PWM DAC With Timer/Port Module

Two PWM outputs with 8-bit resolution are realized. To get the highest speed, TP0.2 and TP0.1 are used as outputs (they have the same bit addresses as the flags RC2FG and RC1FG). The schematic is shown in Figure 3–13. The output ripple is shown in an exaggerated manner. If the PWM information is needed (as for DMC) then the signal at TP0.x is used directly.
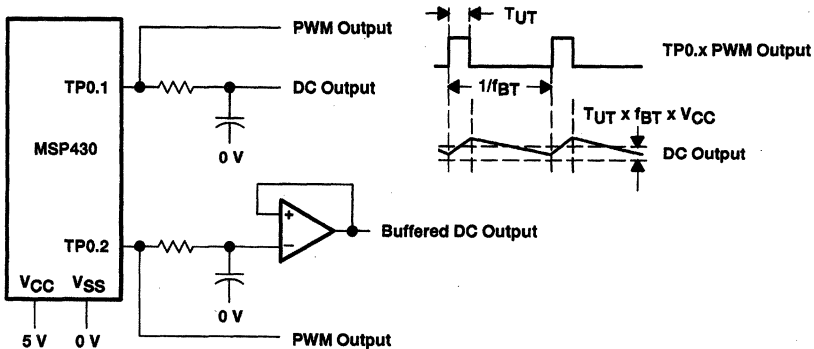
Figure 3–13. PWM for the DAC

Figure 3–14 illustrates the counting of the 8-bit counter during the PWM generation. The interrupt handler of the basic timer sets the 8-bit counter to a negative number of counts (–n1) and sets the output to high; the interrupt handler of the universal timer/port resets the output to zero when it overflows.
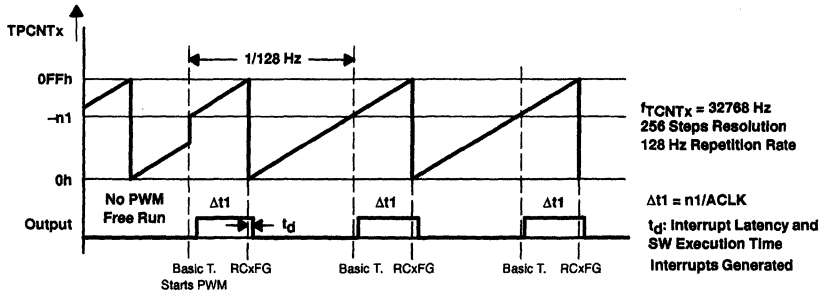
Figure 3–14. PWM Timing by the Universal Timer/Port Module and Basic Timer

```
; MSP430 Software for 8 bit PWM with Universal/Timer Port
; Definitions of the MSP430 hardware
;
Type      .equ    310           ; 310: MSP43C31x   0: others
BTCTL     .equ    040h          ; Basic Timer: Control Reg.
BTCNT1    .equ    046h          ;            Counter
BTCNT2    .equ    047h          ;            Counter
BTIE      .equ    080h          ;              : Intrpt Enable
SSEL      .equ    080h          ;
DIV       .equ    020h          ; BTCTL: xCLK/256
IP2       .equ    004h          ; BTCTL: Clock Divider2
IP1       .equ    002h          ;
IP0       .equ    001h          ;        Clock Divider0
;
SCFQCTL   .equ    052h          ; FLL Control Register
MOD       .equ    080h          ; Modulation Bit: 1 = off
;
CPUoff    .equ    010h          ; SR: CPU off bit
GIE       .equ    008h          ; SR: General Intrpt enable
;
```

```
TPCTL     .equ    04Bh          ; Timer Port:    Control Reg.
TPCNT1    .equ    04Ch          ;                Counter Reg.Lo
TPCNT2    .equ    04Dh          ;                Counter Reg.Hi
TPD       .equ    04Eh          ;                Data Reg.
TPE       .equ    04Fh          ;                Enable Reg.
TP1       .equ    002h          ; Bit address    TP0.1
TP2       .equ    004h          ;                TP0.2
TP3       .equ    008h          ;                TP0.3
TP4       .equ    010h          ;                TP0.4
TP5       .equ    020h          ;                TP0.5
;
          .if     Type=310      ; MSP430C31x?
TPIE      .equ    004h          ; ADC: Intrpt Enable Bit
          .else
TPIE      .equ    008h          ; MSP430C32x configuration
          .endif
IE2       .equ    001h          ; Intrpt Enable Byte
TPSSEL3 .equ      080h          ;
TPSSEL2 .equ      040h
TPSSEL1 .equ      080h          ; Selects clock input (TPCTL)
TPSSEL0 .equ      040h          ;
ENB       .equ    020h          ; Selects clock gate (TPCTL)
ENA       .equ    010h          ;
EN1       .equ    008h          ; Gate for TPCNTx   (TPCTL)
RC2FG     .equ    004h          ; Carry of HI counter (TPCTL)
RC1FG     .equ    002h          ; Carry of LO counter (TPCTL)
EN1FG     .equ    001h          ; End of Conversion Flag "
B16       .equ    080h          ; Use 16-bit counter (TPD)
; RAM Definitions
;
SW_PWM    .equ    0200h         ; Enable bits for TP0.2 and TP0.1
TIM_PWM1 .equ     0201h         ; Calc. PWM result PWM1
TIM_PWM2 .equ     0202h         ; Calc. PWM result PWM2
;
;============================================================
;
```

```
          .sect   "INIT",0F000h                ; Initialization Section
;
INIT      MOV     #0300h,SP                    ; Initialize Stack Pointer
          MOV.B   #IP2+IP1+IP0,&BTCTL          ; Basic Timer 128Hz
;
          MOV.B   #TPSSEL0+ENA,&TPCTL          ; ACLK, EN1=1, TPCNT1
          CLR.B   &TPCNT1                      ; Clear PWM regs
          CLR.B   &TPCNT2
          CLR.B   &TPD                         ; Output Data = Low
          MOV.B   #TPSSEL2+TP2+TP1,&TPE        ; TPCNT2: ACLK
          BIS.B   #TPIE+BTIE,&IE2              ; INTRPTS on
          CLR.B   SW_PWM                       ; No PWM output
          BIC.B   #RC2FG+RC1FG,&TPCTL          ; Reset flags
          EINT
          ...                                  ; Continue with SW
;
; Start both PWMs: calculation results in R6 and R5
;
          MOV.B   R6,TIM_PWM1                  ; (256 - result1)
          MOV.B   R5,TIM_PWM2                  ; (256 - result2)
          BIS.B   #TP2+TP1,SW_PWM              ; Enable PWM2 and PWM1
          ...                                  ; Continue
;
; Disable PWM2: Output zero
;
          BIC.B   #TP2,SW_PWM                  ; Disable PWM2
          ...
;
; Interrupt Handler for the Basic Timer Interrupt: 128Hz
;
BT_INT    BIC.B   #RC2FG+RC1FG,&TPCTL          ; Clear flags
          MOV.B   TIM_PWM2,&TPCNT2             ; (256 - time2)
          MOV.B   TIM_PWM1,&TPCNT1             ; (256 - time1)
          BIS.B   SW_PWM,&TPD                  ; Switch on enabled PWMs
          RETI
;
```

```
; End of Basic Timer Handler
;-----------------------------------------------------------
; Interrupt Handler for the Universal Timer/Port Module
; For max. speed TP0.2 and TP0.1 are used (same bit locations
; as RC2FG and RC1FG). If other locations are used, RLA
; instructions have to be inserted after the flag clearing
;
UT_HNDL PUSH.B   &TPCTL                    ; INTRPT from where?
        AND      #RC2FG+RC1FG,0(SP)        ; Isolate flags
        BIC.B    @SP,&TPCTL                ; Clear set flag(s)
        BIC.B    @SP+,&TPD                 ; Reset actual I/O(s)
        RETI
;
; End of Universal Timer/Port Module Handler
;-----------------------------------------------------------
;
        .sect    "INT_VECT",0FFE2h
        .WORD    BT_INT                    ; Basic Timer Vector
        .if      Type=310
        .sect    "INT_VEC1",0FFEAh         ; MSP430C31x
        .else
        .sect    "INT_VEC1",0FFE8h         ; Others
        .endif
        .WORD    UT_HNDL                   ; UTP Vector (31x)
        .sect    "INT_VEC2",0FFFEh
        .WORD    INIT                      ; Reset Vector
```

Example 3–10. PWM Outputs With 7-Bit Resolution

Two PWM outputs with 7-bit resolution are realized. TP0.4 and TP0.3 are used as PWM outputs (this makes shifting necessary). The schematic is shown in Figure 3–15. Due to the inverting filters at the PWM outputs, the outputs of the MSP430 are also inverted to compensate for this. The output ripple is shown

in an exaggerated manner. If the PWM information is needed (as for DMC) then the signal at TP0.x can be used directly.
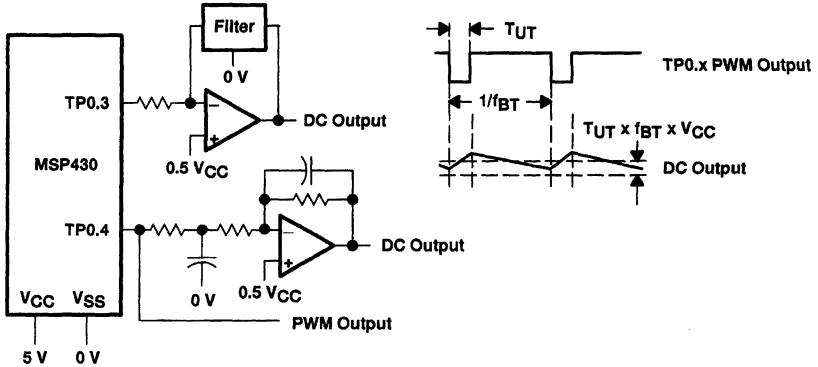
Figure 3–15. PWM for DAC

Figure 3–16 illustrates the operation of the 8-bit counter during the PWM generation. The interrupt handler of the basic timer sets the 8-bit counter to the negative number of counts (–n1) and resets the output to low; the interrupt handler of the universal timer/port sets the output to high when it overflows.
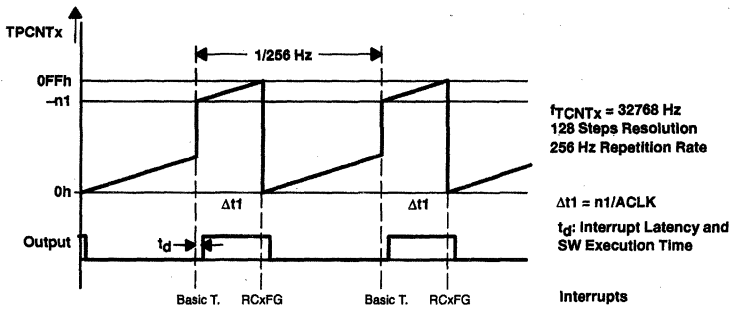
Figure 3–16. PWM Timing by Universal Timer/Port Module and Basic Timer

```
; MSP430 Software for 7 bit PWM with Universal/Timer Port
; Definitions of the MSP430 hardware like above
;
        .sect   "INIT",0F000h           ; Initialization Section
;
INIT    MOV     #0300h,SP               ; Initialize SP
        MOV.B   #IP2+IP1,&BTCTL ; Basic Timer 256Hz
;
        MOV.B   #TPSSEL0+ENA,&TPCTL     ; ACLK, EN1=1, TPCNT1
        CLR.B   &TPCNT1                 ; Clear PWM regs
        CLR.B   &TPCNT2
        BIS.B   #TP4+TP3,&TPD           ; Output Data = high
        MOV.B   #TPSSEL2+TP2+TP1,&TPE   ; TPCNT2: ACLK
        BIS.B   #TPIE+BTIE,&IE2         ; INTRPTS on
        CLR.B   SW_PWM                  ; No output
        BIC.B   #RC2FG+RC1FG,&TPCTL     ; Clear flags
        EINT
        ...
; Start both PWMs: Calculation results in R6 and R5
;
        MOV.B   R6,TIM_PWM1             ; (128 - result)
        BIS.B   #TP3,SW_PWM             ; Enable PWM1
        MOV.B   R5,TIM_PWM2             ; (128 - result)
        BIS.B   #TP4,SW_PWM             ; Enable PWM2
```

```
          . . .
; Disable PWMs: Output is zero
;

          BIC.B    #TP4+TP3,SW_PWM                ; No output
;
; Interrupt Handler for the Basic Timer Interrupt: 256Hz
; The enabled PWMs are switched on
;
BT_INT   BIC.B    #RC2FG+RC1FG,&TPCTL            ; Clear flags
         MOV.B    TIM_PWM2,&TPCNT2              ; (128 - time2)
         MOV.B    TIM_PWM1,&TPCNT1              ; (128 - time1)
         BIC.B    SW_PWM,&TPD                   ; Switch on enabled PWMs
         RETI
;
; End of Basic Timer Handler
;------------------------------------------------------------
; Interrupt Handler for the UT/PM. The PWM-channel that
; caused the interrupt is switched off.
;
UT_HNDL  PUSH     R6                            ; Save R6
         MOV.B    &TPCTL,R6                     ; INTRPT from where?
         AND      #RC2FG+RC1FG,R6              ; Isolate flags
         BIC.B    R6,&TPCTL                     ; Clear set flag(s)
         RLA      R6                            ; To TP0.4/TP0.3
         RLA      R6
         BIS.B    R6,&TPD                       ; Set actual I/O(s)
         POP      R6                            ; Restore R6
         RETI
;
; End of Universal Timer/Port Module Handler
;------------------------------------------------------------
; Vectors like with the example before
```

3.6.5 PWM DAC With the Timer_A

Timer_A of the MSP430 family is ideally suited for the generation of PWM signals. The output unit of each one of the (up to five) capture/compare registers is able to generate seven different output modes. The PWM generation depends mainly on which mode of the Timer_A was used.

❏ Continuous Mode: the timer register runs continuously upwards and rolls over to zero after the value 0FFFFh. The capture/compare register 0 is used like the other capture/compare registers. This mode allows up to five independent timings. The continuous mode is not intended for PWM applications. But, it can be used for relatively slow PWM applications, if other timings are also needed. Interrupt is used for the setting and the resetting of the PWM output. The output unit controls the PWM output and the interrupt handler adds the next time interval to the capture/compare register and modifies the mode of the output unit (set, toggle, or reset).

❏ Up Mode: The timer register counts up to the content of capture/compare register 0 (here the period register) and restarts at zero when it reaches this value The capture/compare register 0 contains the period information for all other capture/compare registers.

❏ Up-Down Mode: The timer register counts up to the content of capture/compare register 0 (here the period register) and counts down to zero when it reaches this value. When zero is reached again, the timer register counts up again. capture/compare register 0 contains the period information for all other capture/compare registers.

All three modes are explained in detail in the Section 6.3, *Timer_A*. Software program examples are also given. If dc output is needed, the same output filters can be used as shown in the previous section. The only difference is the possible speed of the Timer_A (input frequency can be up to the MCLK frequency).

3.6.5.1 PWM DAC With Timer_A Running In Continuous Mode

Up to five completely different PWM generations are possible. If the Timer Register equals one of the four capture/compare latches (programmed to compare mode), the hardware task programmed to the output unit is performed (set, reset, toggle etc.) and an interrupt is requested. Figure 3–17 illustrates the generation of a PWM signal with the capture/compare registers 0. The interrupt handler is responsible for the following tasks:

❏ The time difference (represented by the clock count nx) to the next interrupt is added to the used capture/compare register by software: once $\Delta t0$, once $\Delta t1$

❑ The output unit is programmed to the appropriate mode: set TA0 if Δt1 is added, reset TA0 if Δt0 is added.

❑ Other tasks if necessary

Note:

The continuous mode is not the normal mode for PWM generation due to the software overhead that is necessary. It is used for this purpose only if other independent timings are necessary that cannot be realized with the up mode or the up-down mode.
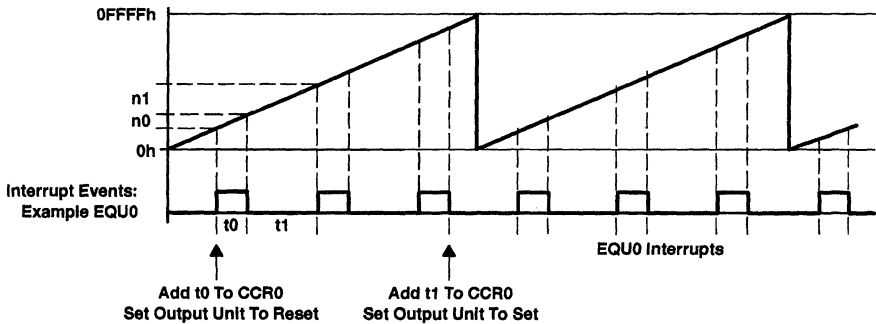
Figure 3–17. PWM Generation with Continuous Mode

3.6.5.2 PWM DAC With Timer_A Running In Up Mode

Up to four different PWM generations with an equal period (repetition rate) are possible. If the timer register equals one of the four capture/compare latches (programmed to compare mode), the hardware task programmed to the output unit is performed (set, reset, toggle etc.) and an interrupt is requested. During the execution of the interrupt handler, the necessary software task is completed. No reloading of the capture/compare register is necessary except if the pulse width changes. If the timer register reaches the programmed value of the capture/compare register 0, then it is reset to zero and restarts there. Figure 3–18 illustrates the generation of two independent PWM signals with the capture/compare registers 1 and 2.
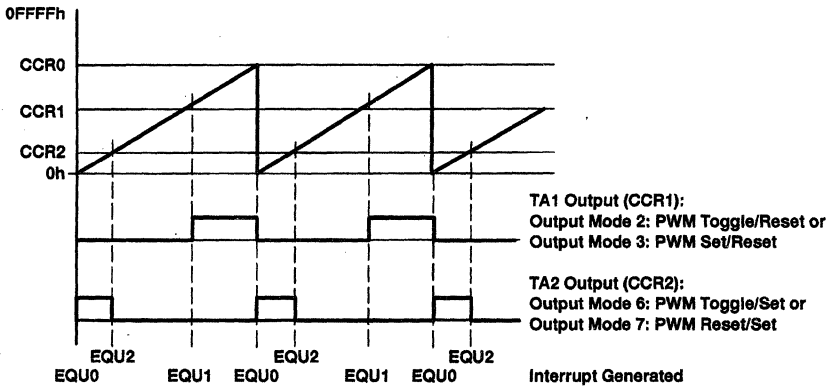
Figure 3–18. PWM Generation With Up Mode

3.6.5.3 PWM-DAC With Timer_A Running In Up-Down Mode

Up to four different PWM generations with an equal period are possible. If the timer register equals one of the four capture/compare latches (programmed to compare mode), the hardware task programmed to the output unit is performed (set, reset, toggle etc.) and an interrupt is requested. During the interrupt handler, the necessary software task is completed. No reloading of the capture/compare register is necessary except if the pulse width changes. The timer register continues to count upward until the value of capture/compare register 0 is reached. Then it counts downward to zero. When it reaches the value of a capture/compare register, the programmed task is made by the output unit and an interrupt is requested again. When zero is reached, the sequence restarts. This way, symmetric PWM generation is possible. The value of the capture/compare register is reached twice for each up-down cycle. Figure 3–19 illustrates the generation of two independent PWM signals with the capture/compare registers 1 and 3.
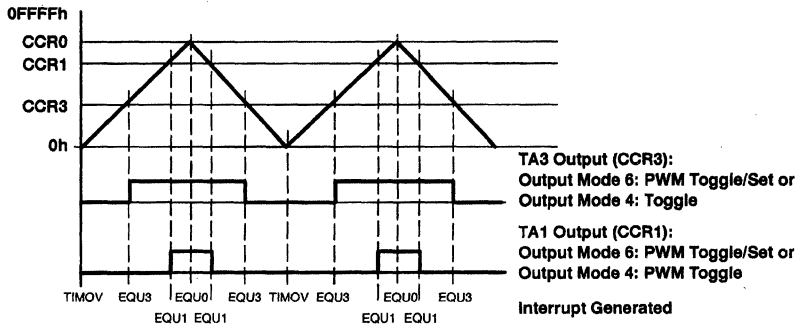
Figure 3–19. PWM Generation with Up-Down Mode

3.7 Connection of Large External Memories

For a lot of MSP430 applications, it is necessary to be able to store large amounts of measured data. For this purpose external memories can be used:

❑ Dynamic RAMs like the TMS44460 (1M × 4 bits)
❑ Synchronous Dynamic RAMs like the TMS626402 (2M × 4 bits)
❑ Flash memories like the TMS28F512A (512K × 8-bits)
❑ EEPROMs

DRAM versions with a self-refresh feature are recommended, otherwise the necessary refresh cycles would waste too much of the processing time.

Figure 3–20 shows the simplest way to control external memory. The unused LCD segment lines are used for addressing and control of the external memory. Four bidirectional I/O lines of port 0 (or another available port) are used for the bidirectional exchange of data. The necessary steps to read from or write to the example TMS44460 DRAM memory are:

1) Output row address to address lines A9 to A0
2) Set the RAS control line low
3) Output column address to address lines A9 to A0
4) Set CAS control lines low and reset them back to high
5) If a read is desired, set OE low and W control lines high. Then read data from DQ4 to DQ1.
6) If a write is desired, set OE high, set W low, and then write the data to DQ4 to DQ1.

The proposal shown in Figure 3–20 needs approximately 200 MCLK cycles for each block of 4-bit nibbles when the O-output lines are used.
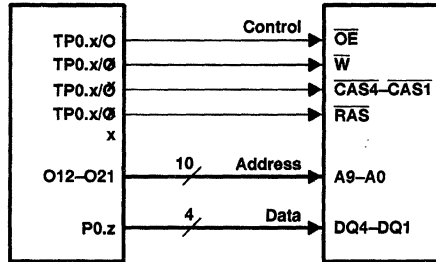
Figure 3–20. External Memory Control With MSP430 Ports

Example 3–11. External Memory Connected to the Outputs

For the circuit shown in Figure 3–20, the 10 address lines of an external memory are connected to the O-outputs, O12 (LSB) to O21 (MSB). The subroutine O_HNDLR is used for the row and column addressing. The driver software and the subroutine call follows:

```
N         .EQU   10/2            ; 10 O-outputs are controlled (O13 to O4)
O_STRT    .EQU   037h            ; Control byte for O12 and O13 (1st byte)
;
          MOV    #03FFh,R5        ; Start with row addressing
          CALL   #O_HNDLR
          ...                     ; Output @RAS signal
          MOV    R9,R5            ; Column address in R9
          CALL   #O_HNDLR         ; Output column address
          ...                     ; Output @CAS signals
;
; Subroutine outputs address info in R5 to O-outputs
; Bit 0 is written to the MSB of the O-outputs. R5 is destroyed
; Execution time:  69 cycles for  8 O-outputs (including CALL)
;                  129 cycles for 16 O-outputs (like above)
;
O_HNDLR   CLR    R6              ; Clear counter
O_HN      MOV    R5,R4           ; Copy actual info
```

```
        AND     #3,R4                   ; Isolate next two address bits
        MOV.B   TAB(R4),O_STRT(R6)      ; Write address bits
        RRA     R5                      ; Prepare next two address bits
        RRA     R5
        INC     R6                      ; Increment counter
        CMP     #N,R6                   ; Through?
        JNZ     O_HN                    ; No, next two bits
        RET
;
; Table contains bit pattern used for the O-outputs
;
TAB     .BYTE   0,0Fh,0F0h,0FFh         ; Patterns 00, 01, 10, 11
```

Figure 3–21 gives an example to use when the LCD segment lines are not available. Two 8-bit shift registers are used for addressing and control of the external memory. Four bidirectional I/O lines of port 0 (or another available port) are used for the exchange of data. Instead of outputting the address and control signals in parallel, this solution's signals are output in series. The output enable signals G2 and G1 are used to omit bad signals that are due to the shifting of the information. The example shown in Figure 3–21 needs approximately 500 cycles for each block of 4-bit nibbles.
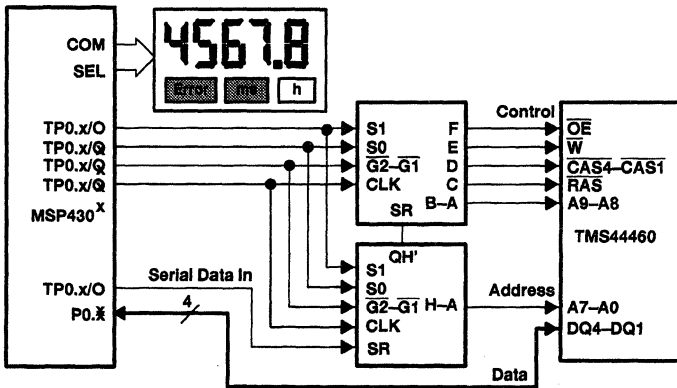
Figure 3–21. External Memory Control With Shift Registers

With nearly the same two hardware solutions, other external memories can be controlled also.

❑ Synchronous Dynamic RAM (TMS626402   2M × 4 bits) with 12 address lines and 6 control lines. Row and column addressing is used. It also uses 4 data bits.

❑ Flash memory (TMS28F512A   512K × 8-bit) with 16 address lines and 3 control lines. Direct addressing is used. It also uses 8 data bits.

Any combination of unused outputs (port, TP0.x, Oy) and shift registers can be used. If DRAMs without self-refresh are used, the low address bits should be controlled by a complete port (port 1 ,2, 3, or 4) to get minimum overhead for the refresh task.

The different versions of the MSP430C33x allow a much simpler and faster solution because of the five available I/O ports. Figure 3–22 illustrates the connection of an AT29LV010A EEPROM (128K × 8 bit) to the MSP430C33x. The example shown in Figure 3–22 needs approximately 30 to 50 MCLK cycles for each byte read or written. The control lines at the MSP430 are I/Os with no second function. All the peripheral functions are available and can be used freely. The MSP30C31x and 32x can address this type of memory by its TP0.x and Ox ports.
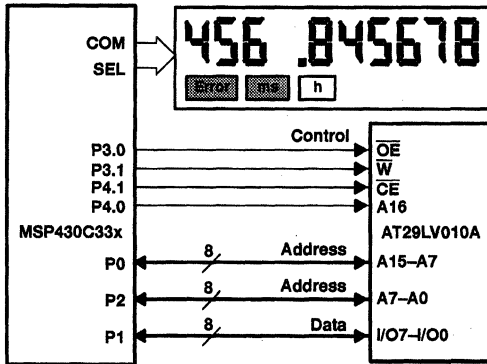
Figure 3–22. EEPROM Control With Direct Addressing by I/O Ports

Figure 3–23 shows the use of an MSP430C33x for the addressing of an external 1-MB RAM. The actual address of the external memory is stored in I/O Ports P0, P2, and P4. The special architecture of the MSP430 allows this method to be used. This method results in the fastest possible access time. The software used for addressing and reading of the next byte is in the following text (this assumes that the address ports are initialized).

```
;                                                               Cycles

        INC.B   &P2OUT          ; Address next Byte              4

        JNC     L$1             ; No carry to A15..A8            2

        ADC.B   &P0OUT          ; Carry to A15..A8              4

L$1     MOV.B   &P1IN,R15       ; Read data at Port1             3
```

The reading of a byte needs (4 + 2 + 3) = 9 cycles. An MCLK frequency of 3.8 MHz results in a read time of 2.37 µs. This access time can be compared with the internal access time of an 8-bit microcomputer. The initialization of a 64-KB memory block is shown in the following text (memory block 1).

```
;                                                               Cycles

        MOV.B   #0,&P2OUT       ; A7..A0 = 00                    4

        MOV.B   #0,&P0OUT       ; A15..A8 = 00                   4

        MOV.B   #CS1+WE,&P4OUT  ; Address memory block1          4
```
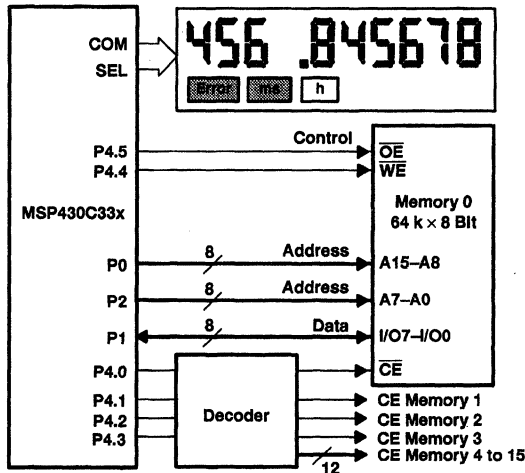

Figure 3–23. Addressing of 1-MB RAM With the MSP430C33x

Figure 3–24 shows how to address an external 1-MB RAM with an MSP430C31x. The actual address of the external memory (stored in the internal RAM) is output with the O-outputs (alternative use of the select lines) and the 6 TP ports. The software for addressing and reading of the next bytes is given in the following text (the address ports are initialized).

```
;                                                           Cycles
        INC.B   A7_0             ; Address next byte          4
        JNC     L$1              ; No carry to A15..A8        2
        ADC.B   A15_8            ; Carry to A15..A8           4
;
; Address A15 to A8 is output to O17 to O10.
; This part is necessary for only 0.4% of all accesses
;
        MOV.B   ·A15_8,R14       ; A15..8 -> R14              3
        MOV     R14,R15          ;                            1
        AND     #3,R15           ; Next 2 address bits        2
        MOV.B   TAB(R15),&036h   ; A9..8 to O11..10           6
        RRA     R14              ; Next 2 address bits        1
        RRA·    R14                                           1
        MOV     R14,R15                                       1
        AND     #3,R15           ; Address A7..6 aso.         2
        ...                      ; 4 x the same A15..6       36
;
; Address bits A7 to A0 output to TP-Port and O27/26
;
L$1     MOV.B   A7_0,&TPD        ; A7..A2 to TP-Port          6
        MOV.B   A7_0,R15         ; A1..A0 generated           3
        AND     #3,R15           ; A1..A0 in R15              2
        MOV.B   TAB(R15),&LCDx   ; A1..A0 to O27 and O26      6
        MOV.B   &P0IN,R15        ; Read data at Port0         3
        ...                      ; Process data
;
TAB     .BYTE   0, 0Fh, 0F0h, 0FFh   ; For O-outputs
```

The reading of one byte needs 26 cycles (addresses A15 – A8 are unchanged) or 83 cycles when A15 – A8 must be changed. An MCLK frequency of 3.8 MHz results in 6.9 µs or 21.8 µs for one byte, respectively.

The decoding of the 64-KB memory blocks is made with a normal 4-to-16 line decoder.
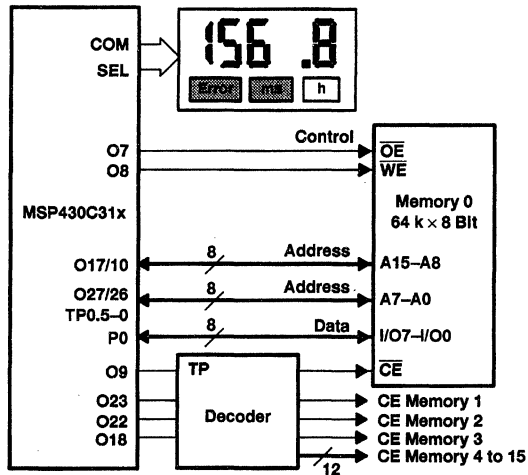
Figure 3–24. Addressing of 1-MB RAM With the MSP430C31x

3.8 Power Supplies for MSP430 Systems

There are various ways to generate the supply voltage(s) for the MSP430 systems. Due to the extremely low-power consumption of the MSP430 family, this is possible with batteries, accumulators, the M-Bus, fiber-optic lines, and ac. Every method uses completely different hardware and is explained in depth. Wherever possible, the formulas necessary for the hardware design are given too.

3.8.1 Battery-Power Systems

Due to the extremely low current consumption of the MSP430 family it is possible to run an MSP430 system with a 0.5-Ah battery more than 10 years. This makes possible applications that were impossible before. To reach such extended time spans, it is only necessary to observe some simple rules. The most important one is to always switch off the CPU when its not needed (e.g., after the calculations are completed). This reduces the current consumption from an operational low of 400 μA down to 1.6 μA.

The Figures 3–25 and 3–26 are drawn in a way that makes it easier to see how the battery needs to be connected to get the highest accuracy out of the ADC.

Figure 3–25 illustrates the MSP430C32x with its separated digital and analog supply terminals. This provides a separation of the noise-generating digital parts and the noise-sensitive analog parts.

Figure 3–26 shows how to best separate the two parts for the MSP430 family members with common supply terminals for the analog and digital parts of the chip.

If the battery used has a high internal resistance, RI, (like some long-life batteries) then the parallel capacitor Cch must have a minimum capacity. The supply current for the measurement part (which cannot be delivered by the battery) is delivered via Cch. The equation includes the small current coming from the battery.

$$C_{chmin} \geq t_{meas} \times \left(\frac{I_{AM}}{\Delta V_{ch}} - \frac{1}{R_I} \right)$$

Between two measurements, the capacitor Cch needs time, tch, to get charged-up to V_{CC} for the next measurement. During this charge-up time, the MSP430 system runs in low-power mode 3 to have the lowest possible power consumption. The charge-up time, tch, to charge Cch to 99% of V_{CC} is:

$$t_{chmin} \geq 5 \times C_{chmax} \times R_{imax}$$

Where:

I_{AM}	Medium system current (MSP430 and peripherals)	(A)
t_{meas}	Discharge time of Cch during measurement	(s)
ΔV_{ch}	Tolerable discharge of Cch during time tmeas	(V)
R_i	Internal resistance of the battery	(Ω)
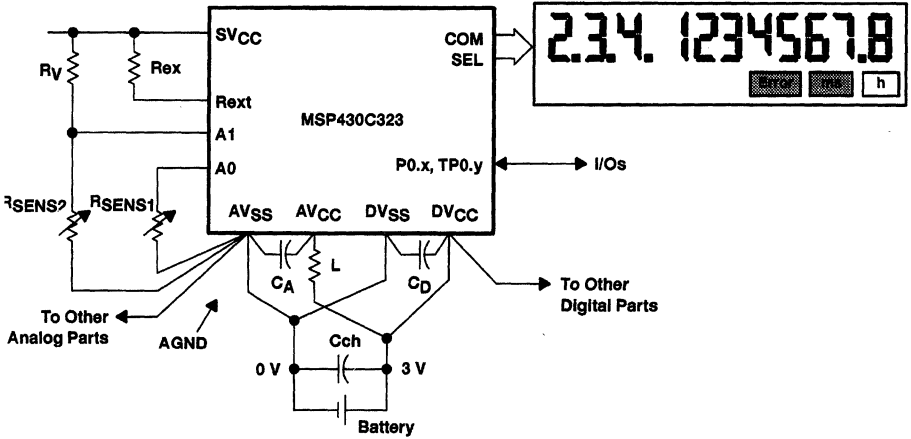
Figure 3–25. Battery-Power MSP430C32x System
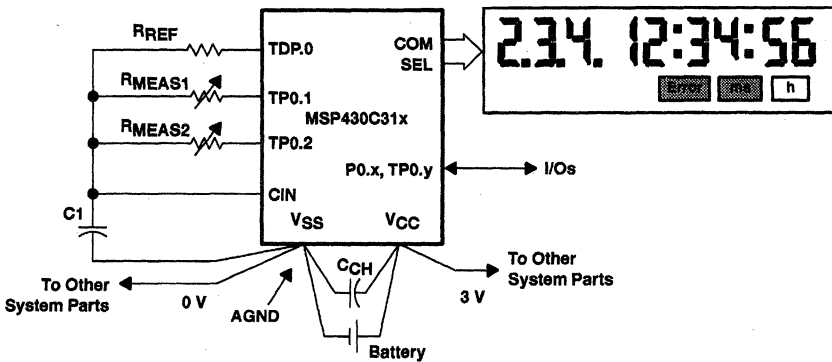
Figure 3-26. Battery-Power MSP430C31x System

Note:

The way the battery is connected to the MSP430 (shown in Figures 3-25 and 3-26) is not restricted to battery-driven MSP430 systems. The decoupling of the analog and the digital parts is necessary for all methods of supplied power. The following schematics are drawn in a simpler way to give better readability.

3.8.2 Accumulator-Driven Systems

The MSP430 can also be supplied from an accumulator. An advantage of this solution is that the MSP430 can also take over the battery management for the accumulator.

☐ Current Measurement: Summing up of the charge and discharge currents. If these currents (measured with sign) are multiplied with constants that are unique for the accumulator type used (e.g. NiCd, Pb) then it is possible to have a relatively accurate value for the actual charge. The current is measured with a shunt. The measured voltage drop is shifted into the middle of the ADC range by the current Ics (generated by the MSP430's internal current source) that flows through Rc. This method allows signed current measurements.

❏ Temperature Measurement: All of the internal processes of an accumulator (e.g., maximum charge, self discharge) are strongly dependent on the temperature of the pack. Therefore, the temperature of the pack is measured with a sensor and used afterwards with the calculations. When the MSP430's current source is used, the voltage drop of its current Ics across the sensor resistance is measured with the ADC input A2.

❏ Voltage Measurement: The voltage of an accumulator pack is an indication of the states full charge and complete discharge. Therefore, the voltage of the pack is measured with the voltage divider consisting of R1 and R2.

❏ Charge Control: Dependent on the result of the charge calculations, the MSP430 can decide if the charge transistor needs to be switched on or off. This decision can also be made in PWM (Pulse Width Modulation) mode. Figure 3–27 shows three possible charge modes. If replaceable accumulators are used, the charge control is not needed.

❏ Rest Mode Handling: During periods of non-use, the low power mode 3 of the MSP430 allows the control of the rest mode. The rest mode has nearly no current consumption. In fact, the supply current has the same magnitude as the self-discharge current of the accumulator. All system peripherals are switched off; the MSP430 wakes-up at regular intervals, which are controlled by its basic timer. It then calculates, every few hours, the amount of self discharge of the accumulator. This calculated value is subtracted from the actual charge level.

Figure 3–27 illustrates an MSP430 system driven by an accumulator. The battery management is done by the MSP430 also. The hardware needed is simple. As shown in the figure, just a few resistors and a temperature sensor. The actual charge of the accumulator is indicated in the LCD with a bar graph ranging from Empty to Full.

All necessary constants and a security copy of the actual charge are contained in an external EEPROM typically with 128 x 8 bits.

> **Note:**
>
> The hardware shown in Figure 3–27 can also be used for an intelligent accumulator controller. Only the hardware necessary for this task is shown. The measurement parts for voltage, current, and temperature are exactly the same as shown.
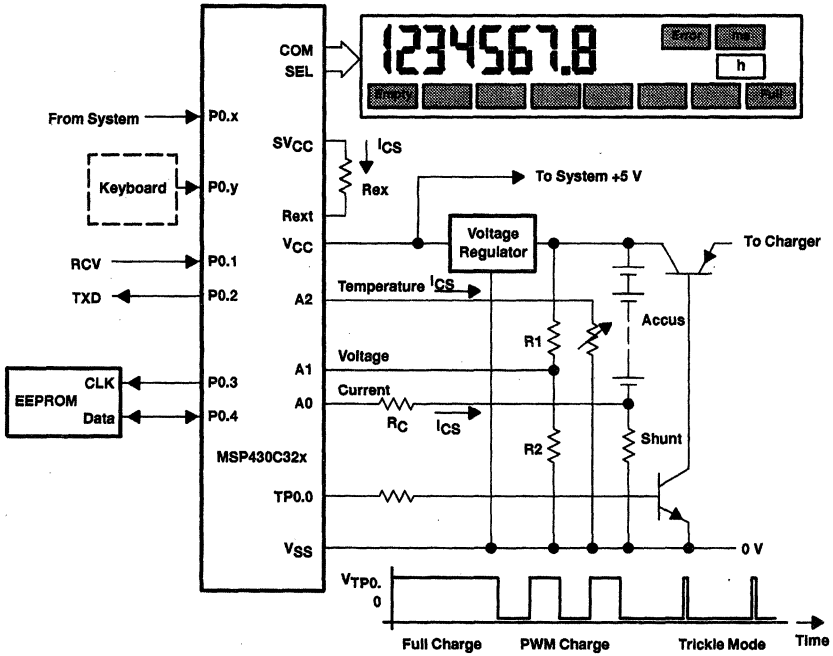
Figure 3–27. Accumulator-Driven MSP430 System With Battery Management

3.8.3 AC-Driven Systems

The current consumption of microcomputer systems gets more and more important for ac-driven systems. The lower the power consumption of a microcomputer system, the simpler and cheaper the power supply can be

3.8.3.1 Transformer Power Supplies

Transformers have two big advantages:

❏ Complete isolation from ac. This is an important security attribute for most systems.

❏ Very good adaptation to the needed supply voltage. This results in a good power efficiency.

Most ac-driven applications are only possible because of the isolation from the ac the transformer provides.

Half-Wave Rectification

Half-wave rectification uses only one half-wave of the transformer's secondary voltage, V_{SEC}, for the powering of an application. Figure 3–28 illustrates the voltages used with the equations.
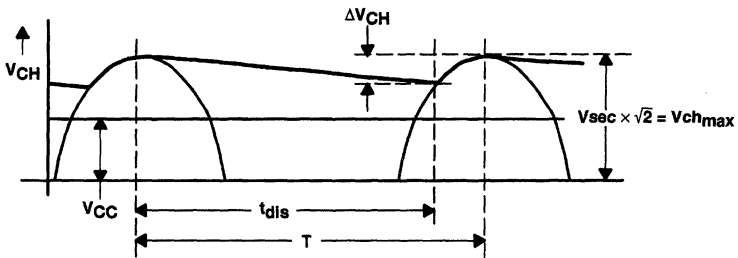
Figure 3–28. Voltages and Timing for the Half-Wave Rectification

❏ Advantages

■ Simplified hardware
■ Rectification with the voltage drop of only one diode

❏ Disadvantages

■ Charge capacitor, C_{CH}, must have doubled capacity compared to full-wave rectification
■ Higher ripple on the dc supply voltage
■ DC flows through the transformer's secondary winding

Figure 3–29 shows the most simple ac driven power supply. The positive half-wave of the transformer's secondary side charges the load capacitor, $C_{CH'}$.

The capacitor's voltage is stabilized with a Zener diode having a Zener voltage equal to the necessary supply voltage V_{CC} of the MSP430.

Two conditions must be met before a final calculation is possible:

$$V_{SECmin} \times \sqrt{2} - \Delta V_{ch} > V_Z$$

and:

$$\frac{T}{2 \times C_{chmin}} < R_V < \frac{V_{SECmin} \times \sqrt{2} - \Delta V_{ch} - V_Z}{I_{AMmax}}$$

The charge capacitor, C_{CH}, must have a minimum capacity:

$$C_{chmin} \geq \frac{T}{2} \times \left(\frac{1}{R_V} + \frac{I_{AM}}{V_{SECmin} \times \sqrt{2} - V_z} \right)$$

The peak-to-peak ripple voltage $V_{N(PP)}$, of the supply voltage, V_{CC}, is:

$$V_{npp} \approx \frac{\dfrac{V_{CC}}{I_{AM}} \parallel R_z}{R_V + \dfrac{V_{CC}}{I_{AM}} \parallel R_z}$$

The final necessary secondary voltage, V_{SEC}, of the ac transformer is (ΔV_{CH} = $0.1 \times V_{CHmax}$):

$$V_{SECmin} \geq \frac{1}{\sqrt{2}} \times \left[\frac{0.45 \times T \times I_{AM}}{C_{chmin} - \frac{0.45 \times T}{R_V}} + V_z \right]$$

Where:

I_{AM}	Medium system current (MSP430 and peripherals)	[A]
T	Period of the ac frequency	[s]
ΔV_{ch}	Discharge of Cch during time tdis	[V]
V_{CC}	Supply voltage of the MSP430 system	[V]
V_z	Voltage of the Zener diode	[V]
R_z	Differential resistance of the Zener diode	[$\Delta V/\Delta A$]
R_v	Resistance of the series resistor	[Ω]
V_{sec}	Secondary (effective) voltage of the transformer (full load conditions)	[V]
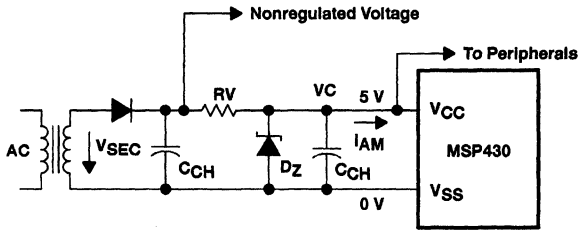
Figure 3–29. Half-Wave Rectification With 1 Voltage and a Zener Diode

Figure 3–30 shows a simplified power supply that uses a voltage regulator like the µA78L05. The charge capacitor, Cch, must have a minimum capacity:

$$C_{chmin} \geq \frac{I_{AM} \times t_{dis}}{\Delta V_{ch}}$$

The peak-to-peak ripple VN(PP) on the output voltage Vreg depends on the used voltage regulator. The regulators ripple rejection value can be seen in its specification. The necessary secondary voltage Vsec of the ac transformer under full load conditions is:

$$V_{SECmin} \geq \frac{1}{\sqrt{2}} \times \left(V_{reg} + V_r + V_d + t_{dis} \times \frac{I_{AM}}{C_{chmin}} \right)$$

The discharge time tdis used with the previous equations is:

$$t_{dis} = T \times \left[1 - \frac{\arccos\left(1 - \frac{\Delta V_{ch}}{V_{SEC} \times \sqrt{2}} \right)}{2\pi} \right]$$

Where:

t_{dis}	Discharge time of Cch	[s]
V_d	Voltage drop of one rectifier diode	[V]
V_r	Dropout voltage (voltage difference between output and input) of the voltage regulator for function	[V]
V_{reg}	Nominal output voltage of the voltage regulator	[V]

For first estimations the value of tdis is calculated for two different discharge values:

❑ 10% discharge of Cch during tdis   tdis = 0.93T
❑ 30% discharge of Cch during tdis   tdis = 0.88T
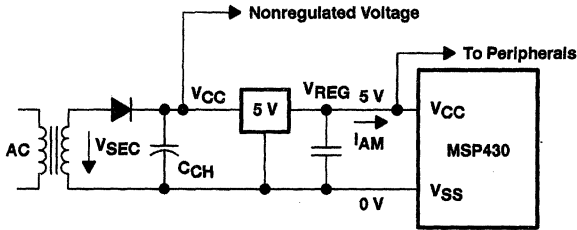
Figure 3–30. Half-Wave Rectification With One Voltage and a Voltage Regulator

Figure 3–31 shows an MSP430 system that uses two supply voltages: +5 V and –5 V. The negative supply voltage is used for analog interfaces. Simple resistor dividers interface the 10-V analog part into the 5 V range of the MSP430. The formulas for the calculation of the charge capacitor, Cch, and the necessary secondary voltage, Vsec, are the same as shown for the circuitry in Figure 3–30. The same circuitry can be used for a system with +2.5 V and –2.5 V (see Figure 3–37 for more details).
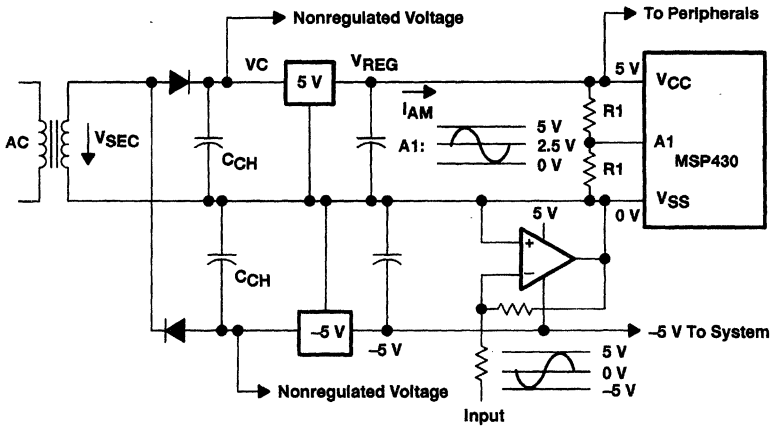
Figure 3–31. Half-Wave Rectification With Two Voltages and Two Voltage Regulators

Full-Wave Rectification

Full-wave rectification uses both half-waves of the secondary voltage, Vsec, for the powering of the application.
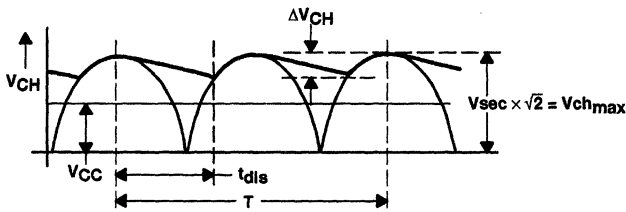
Figure 3–32. Voltages and Timing for Full-Wave Rectification

❑ Advantages

  ■ Smaller charge capacitor Cch
  ■ Lower ripple voltage
  ■ No dc current through transformer's secondary winding

❑ Disadvantages

  ■ Four diodes or a transformer with center tap is necessary
  ■ Voltage drop of two diodes in series (except with a transformer having a center tap)

Figure 3–33 shows a simple power supply that uses a μA78L05 voltage regulator. The charge capacitor, Cch, must have a minimum capacity:

$$C_{chmin} \geq \frac{I_{AM} \times t_{dis}}{\Delta V_{ch}}$$

The peak-to-peak ripple, Vn_{pp}, on the voltage, V_{CC}, depends on the voltage regulator used. The ripple rejection value can be seen in the voltage regulator specification. The necessary secondary voltage, Vsec, of the ac transformer is for the upper rectifier with four diodes (full load conditions):

$$V_{SECmin} \geq \frac{1}{\sqrt{2}} \times \left(V_{reg} + V_r + 2 \times V_d + t_{dis} \times \frac{I_{AM}}{C_{chmin}} \right)$$

For the center tap transformer, Vd, in the previous equation is multiplied by one ($1 \times$ Vd). The discharge time tdis used with the previous equations is:

$$t_{dis} = T \times \left[0.5 - \frac{arcos\left(1 - \dfrac{\Delta V_{ch}}{V_{SEC} \times \sqrt{2}} \right)}{2\pi} \right]$$

For first estimations the value of tdis is calculated for two different discharge values:

- ☐ 10% discharge of Cch during tdis   tdis = 0.43T
- ☐ 30% discharge of Cch during tdis   tdis = 0.38T
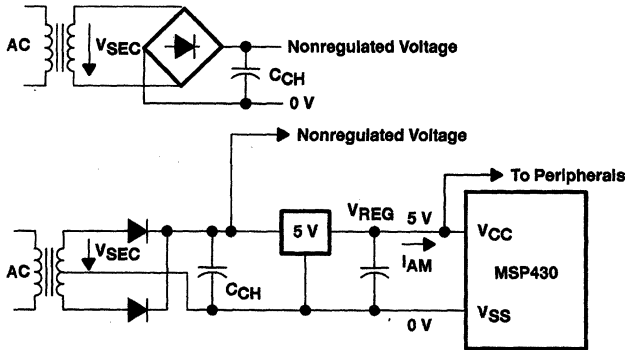
Figure 3–33. Full Wave Rectification for one Voltage with a Voltage Regulator

Figure 3–34 shows an MSP430 system that uses two supply voltages: +2.5 V and –2.5 V. The formulas for the calculation of the charge capacitor, Cch, and the necessary secondary voltage, Vsec, are the same as given for the circuitry in Figure 3–33. The circuitry of Figure 3–34 can also be used for a system with +5-V and –5-V supply (see Figure 3–31 for more details).

Also shown, is how to connect a TRIAC used for ac motor control. The relatively high gate current needed is taken from the non-regulated positive voltage. This reduces the noise within the regulated MSP430 supply. The current flowing through the motor is measured with the ADC for control purposes. The ADC result for 0 V (measured at A0) is subtracted from the current ADC value and results in a signed, offset-corrected value. If a single supply voltage is used (+5V only), the current source can be used to shift the signed current information into the range of the ADC. See Figure 3–27 for the current measurement circuit.
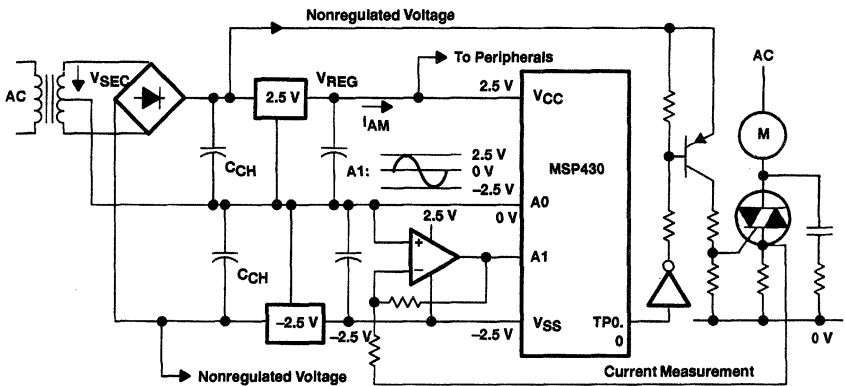
Figure 3–34. Full-Wave Rectification for Two Voltages With Voltage Regulators

3.8.3.2 Capacitor Power Supplies

Applications that do not need isolation from the ac supply or that have a defined connection to the ac supply (like electricity meters) can use capacitor power supplies. The transformer is not needed and only the series capacitor, Cm, must have a high voltage rating due to the voltage spikes possible on ac source.

The ac resistance of the series capacitor, Cm, is used in a voltage divider. This means relatively low power losses. The active power losses are restricted to the protection resistor, Rm, connected in series with Cm. This protection resistor is necessary to limit the current spikes due to voltage spikes and high frequency parts overlaid to the ac voltage. The current I_{ac} through the circuitry is:

$$I_{ac} = \frac{V_{ac}}{\frac{1}{j\omega \times C_m} + R_m} \approx \frac{V_{ac}}{\sqrt{\frac{1}{\omega^2 \times C_m^2} + R_m^2}}$$
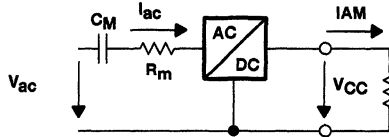
Where:

V_{ac}	ac voltage	[V]
f_{ac}	Nominal frequency of the ac	[Hz]
ω	Circle frequency of the ac: $\omega = 2\pi f$	[1/s]
C_m	Series capacitor	[F]
R_m	Series resistor	[Ω]

The previous formula for I_{ac} is valid for all shown capacitor power supplies. The formula assumes low voltages will be generated (< 5% of the ac voltage). For a dc current, I_{AM}, the necessary ac current Iac is:

$$I_{ac} \geq I_{AM} \times \frac{\pi}{\sqrt{2}} = I_{AM} \times 2.221$$

The capacitor, Cm, is:

$$C_{mmin} \geq \frac{1}{2\pi \times f_{acmin}} \times \frac{1}{\sqrt{\left(\frac{V_{acmin} \times \sqrt{2}}{\pi \times I_{AM}}\right)^2 - R_{mmax}^2}}$$

This formula for Cm is valid for all shown capacitor supplies. The calculated value for Cm includes the tolerances for the ac voltage and the ac frequency; the minimum values used for Vac and fac ensure this.

The protection resistor, Rm, for a maximum spike current Imax generated by a voltage spike Vspike is:

$$R_m \geq \frac{V_{spike}}{I_{max}}$$

The charge capacitor, Cch, must have a minimum capacity:

$$C_{chmin} \geq \frac{I_{AM} \times T}{2 \times \Delta V_{ch}}$$

❏ Advantages

■ No transformer necessary
■ Very simple hardware

❏ Disadvantages

■ No isolation from ac

Capacitor Supplies for a Single Voltage

Figure 3–35 shows the simplest capacitor power supply. The Zener diode used for limiting the voltage of the charge capacitor, Cch, is used for the voltage regulation too. The peak-to-peak ripple voltage, Vn_{pp}, on voltage, V_{CC}, is:

$$V_{npp} \approx I_{AM} \times \frac{T}{C_{ch} \times 2}$$

The voltage of the Zener diode, Dz, is:

$$V_z \approx V_{CC} + V_d$$

Cch is calculated as shown in Section 3.8.3.2, *Capacitor Power Supplies*.
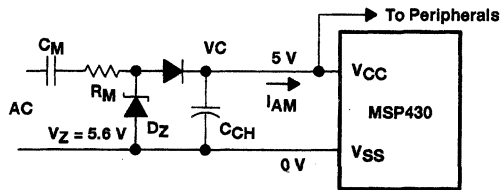
Figure 3–35. Simple Capacitor Power Supply for a Single Voltage

Figure 3–36 shows a hardware proposal for a regulated output voltage, V_{CC}. The voltage, Vz, of the Zener diode, Dz, must be:

$$V_z \geq V_d + V_{reg} + V_r + T \times \frac{I_{AM}}{2 \times C_{chmin}}$$

Cch is calculated as shown in Section 3.8.3.2, *Capacitor Power Supplies*.
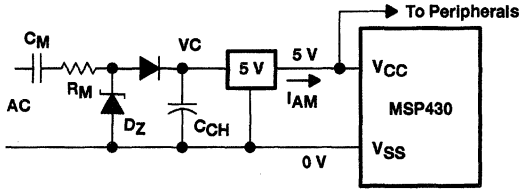
Figure 3–36. Capacitor Power Supply for a Single Voltage

Capacitor Supplies for Two Voltages

Applications that need two voltages (e.g., +2.5 V and –2.5 V) can also use a capacitor supply.

Figure 3–37 shows a split power supply with two regulated output voltages. Together, they deliver the supply voltage, V_{CC}. The split power supply allows the measurement of the voltage of the 0-V line at A0. This value can be subtracted from all other measured analog inputs. This results in offset corrected, signed values. The voltage, Vz, of each Zener diode, Dz, must be:

$$V_z \geq V_r + V_{reg} + T \times \frac{I_{AM}}{2 \times C_{chmin}}$$

The two charge capacitors, Cch, must have the values:

$$C_{chmin} \geq \frac{I_{AM} \times T}{\Delta V_{ch} \times 2}$$
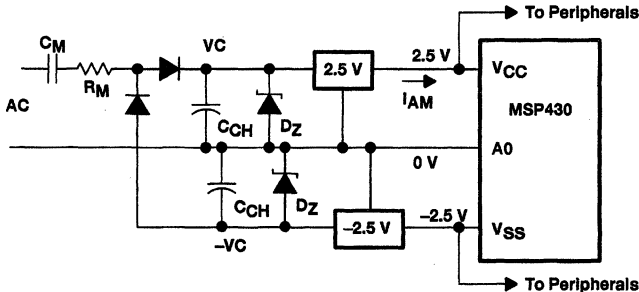
Figure 3–37. Split Capacitor Power Supply for Two Voltages

Figure 3–38 shows a split power supply for +2.5 V and –2.5 V made in a completely different way. It is capable of delivering relatively large output currents due to the buffer transistors. If the high current capability is not needed, the transistors can be omitted and the loads connected to the outputs of the two operational amplifiers directly. The reference for all voltages is a reference diode, LMx85. The highly stable 1.25 V output of this diode is multiplied by two (for +2.5V) or multiplied by –3 and added to the reference value, which delivers –2.5 V. The voltage drop of each one of the two diodes, D, is compensated by the series connection of the two Zener diodes, Dz. The required Zener voltage, Vz, of the two diodes Dz is:

$$V_z \geq \frac{V_{CC}}{2} + V_{BE} + \left(V_{CC} - V_{om}\right) + T \times \frac{I_{AM}}{2 \times C_{chmin}}$$

Where:

V_{BE}	Basis-Emitter voltage of a transistor	[V]
V_{om}	Maximum peak output voltage swing of the operational amplifier with VC	[V]
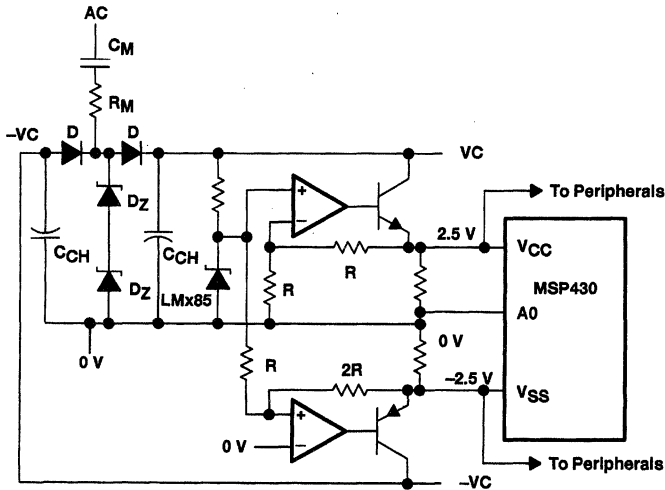
Figure 3–38. Split Capacitor Power Supply for Two Voltages With Discrete Components

3.8.4 Supply From Other System DC Voltages

Existing dc voltages of the controlled system, like +12 V or +24 V, can be used for the supply to the MSP430 system. This is possible due to the low current consumption of the MSP430. So there is nearly no power wasted in the voltage regulator for V_{CC}. If relays and other power consuming peripherals need to be used, the system dc voltage, Vsys, can be used (see Figure 3–40). This solution has two advantages:

❏ The switching noise is generated outside of the MSP430 supply

❏ The power for the switched parts does not increase the power of the MSP430 supply

Figure 3–39 and 3–40 show four different possible supplies for an MSP430 system from an existing +12 V (or +24 V) power supply.

3.8.4.1 Zener Diode

A simple configuration of a series resistor R_V with a Zener diode D_Z delivers an output voltage of +3 V or +5 V. The resistor (Rv) is:

$$R_{vmax} > \frac{V_{sysmin} - V_z}{I_{zmin} + I_{AM}}$$

Where:

V_z	Zener voltage of the Zener diode	[V]
I_z	Current through the Zener diode	[A]
V_{sys}	Nominal system voltage	[V]

3.8.4.2 Zener Diode and Operational Amplifier

If larger currents or a higher degree of decoupling is necessary, then an operational amplifier can be used additionally. This way the series resistor Rv can have a much higher resistance than without the operational amplifier. The NPN buffer transistor is only necessary if the operational amplifier cannot output the needed system current. The series resistor Rv is calculated with:

$$R_{vmax} > \frac{V_{sysmin} - V_z}{I_{zmin}}$$

3.8.4.3 Reference Diode With Operational Amplifier

The low voltage of a reference diode (e.g., LMx85) is amplified with an operational amplifier and also buffered. The series resistor, Rv, feeds only the reference diode and has a relatively high resistance. Therefore, it is calculated the same way as shown in Section 3.8.4.2, *Zener Diode*. The output voltage, Vout, is calculated with:

$$V_{out} = V_z \times \frac{R1 + R2}{R2}$$
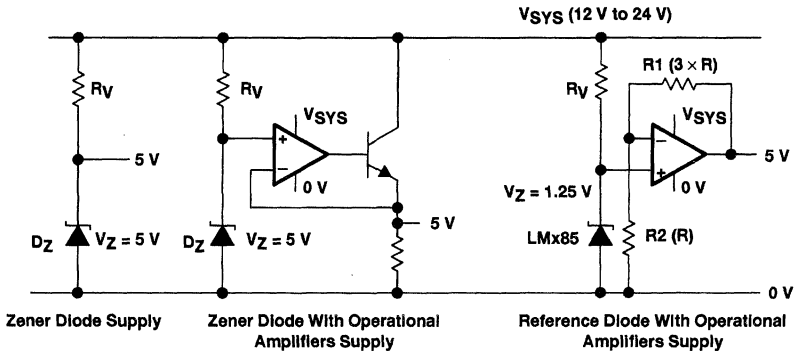
Figure 3–39. Simple Power Supply From Other DC Voltages

3.8.4.4 Integrated Voltage Regulator

Figure 3–40 illustrates the use of an integrated voltage regulator. Here a TPS7350 (regulator plus voltage supervisor) is used, so a highly-reliable system initialization is possible. The TPS7350 also allows the use of the RST/NMI terminal of the MSP430 as described in Section 5.7, *Battery Check and Power Fail Detection*. The RST/NMI terminal is used while running a normal program as an NMI (Non-Maskable Interrupt). This makes possible the saving of important data in an external EEPROM in case of power failure. This is because PG

outputs a negative signal starting at $V_{CC} = 4.75$ V, which allows a lot of activities until V_{CCmin} of the MSP430 (2.5 V) is reached.
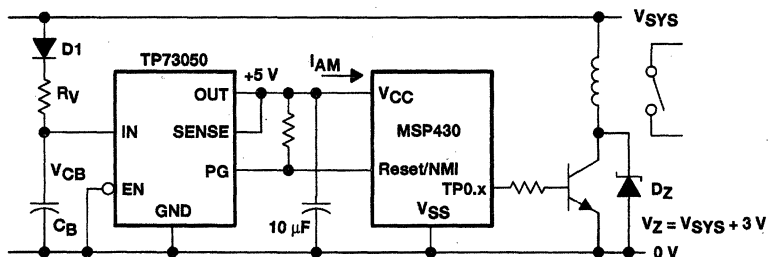
Figure 3–40. Power Supply From Other DC Voltages With a Voltage Regulator

With two additional components (an RC combination) the MSP430 system can be protected against spikes and bridging of supply voltage dropouts is possible. The diode, D1, protects the capacitor, Cb, against discharge during dropouts in voltage Vsys. The series resistor, Rv, is:

$$R_{vmax} < \frac{\left(V_{sysmin} - V_d - V_{CC} - V_r\right)}{I_{AM} + I_{regmax}}$$

The minimum capacity of Cb is:

$$C_{bmin} > \frac{\Delta t \times \left(I_{AM} + I_{reg}\right)}{V_{sysmin} - \left(I_{AM} + I_{regmax}\right) \times R_{vmax} - V_{rmax} - V_{CCmin}}$$

Where:

I_{AM}	System current (medium value MSP430 and peripherals)	[A]
I_{reg}	Supply current of the voltage regulator	[A]
V_{sys}	System voltage (e.g., +12 V)	[V]
V_d	Diode forward voltage (< 0.7 V)	[V]
Vr	Dropout voltage of the voltage regulator for function	[V]
V_{CC}	Supply voltage of the complete MSP430 system	[V]
Δt	Dropout time of V_{sys} to be bridged	[s]

3.8.5 Supply From the M Bus

If the MSP430 system is connected to the M–Bus, three possibilities exist for the supply of the MSP430:

❑ Battery Supply: The supply of the MSP430 is completely independent of the M-Bus. This method is not shown in Figure 3–41, because it is the normal way the MSP430 is powered (see Section 3.8.1, *Battery Driven Systems*).

❑ M-Bus Supply: The MSP430 system is always supplied by the M-Bus. During off phases of the M-Bus, the MSP430 is not powered.

❑ Mixed Supply: Normally the M-Bus supplies the MSP430 and only during off phases of the M-Bus does the battery of the MSP430 provide power.

3.8.5.1 M-Bus Supply

The MSP430 is always powered from the M-Bus. The TSS721 power fail signal, PF, indicates to the MSP430 failure of the bus voltage. This early warning enables the MSP430 to save important data in an external EEPROM. The capacitor, Cch, must have a capacity that allows this storage:

$$C_{chmin} \geq \frac{I_{AM} \times t_{store}}{V_{DD} - V_{CCmin}}$$

Where:

I_{AM}	System current (MSP430 and EEPROM)	[A]
t_{store}	Processing time to store important data into the EEPROM	[s]
V_{DD}	Supply voltage delivered from the TSS721	[V]
V_{CCmin}	Minimum supply voltage of the complete MSP430 system	[V]

3.8.5.2 Mixed Supply

The MSP430 is powered from the M-Bus while bus voltage is available. During times without bus voltage, the battery powers the MSP430. Therefore, a smaller battery can be used when normal bus power is available. The MOS transistor switches to the battery when there is a dropout of the M-Bus voltage. Details are described in the *TSS721 M-Bus Transceiver Application Report*.
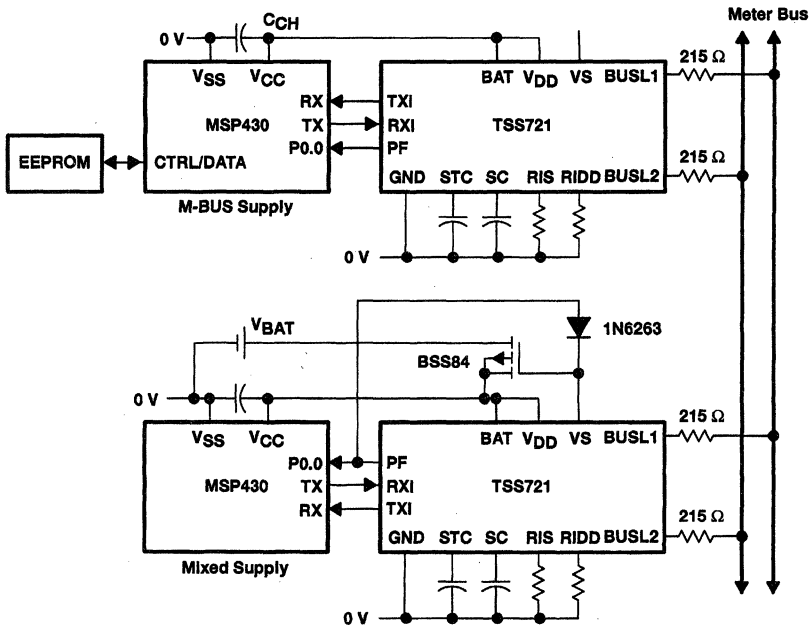
Figure 3–41. Supply From the M Bus

3.8.6 Supply Via a Fiber-Optic Cable

The MSP430 needs a supply current of only 400μA, if supplied with a voltage of 3 V and operating with an MCLK of 1 MHz. This low power can be transmitted via a glass-fiber (fiber-optic) cable. This allows completely isolated measurement systems, which are not possible with other microcomputers. This transmission mode is an advantage for applications in strong electric or magnetic fields.

Because the data transmitted from the host to the MSP430 is also used for the supply of the MSP430 system, a certain amount of light is required continuously and is independent of the transmitted data. Possible ways to reach this are:

❏ Use of extended charge periods between host-to-MSP430 data transfers. The MARK level of the RS232 protocol is used for this purpose. This method is shown in Figure 3–42 with every logical one, stop bit, and MARK level used for the supply.

❏ Use of a transmission code that always transmits the same number of ones and zeroes, independent of the transmitted data. (e.g., the Bi-Phase Code)

To achieve a positive current balance, a few conditions must be met:

❏ The complete hardware design uses ultra-low-power devices (operational amplifiers, reference diode, measurement parts etc.).

❏ The MSP430 is in Low Power Mode 3 anytime processing power is not needed.

❏ The measurement unit is switched on only during the actual measurement cycle.

❏ All applicable hints given in Section 4.9, *Ultra-Low-Power Design with the MSP430 Family* are used.

3.8.6.1 Description of the Hardware

The host sends approximately. 15 mW of optical power into the fiber-optic cable. This optical power is made with a laser diode consuming 30 mW of electrical power. At the other end of the fiber-optic cable, the optical power is converted into 6 mW of electrical power with a power-converter diode. The open-circuit voltage of the power converter (approximately 6 V) decreases to 5 V with the load represented by the MSP430. The received electrical energy is used to charge the capacitor, Cch. The charge-up time required is approximately 300 ms for a capacitor with 30 μF.

The uppermost operational amplifier is used for the voltage regulation of the system supply voltage (3 V to 4 V).

If a stabilized supply voltage is unnecessary, then this operational amplifier can be omitted, as well as the diodes and pull-up resistors at the RST/NMI and P0.1 inputs.

The reference for the complete system is an LMx85 reference diode. This reference voltage (1.25 V) is used for several purposes: trigger threshold for the Schmitt-triggers, reference for the calculation of V_{CC}, and reference for the voltage regulator.

The operational amplifier in the middle works as a reset controller. The Schmitt-trigger switches the RST/NMI input of the MSP430 to a high level when VC reaches approximately 4 V. The RST/NMI input is set low when VC falls below 2.5 V.

The third operational amplifier decodes the information out of the charge voltage and data of the power converter output. This decoder also shows a Schmitt-trigger characteristic.

The measured data is sent back to the host by an IR LED controlled by an NPN transistor. The data format used here is an inverted RS232 protocol and has no current flow for the MARK information (e.g. stop bits).

3.8.6.2 Working Sequence

The normal sequence for a measurement cycle is as follows:

1) The host starts a measurement sequence with the transmission of steady light. This time period is used for the initial charge-up of the charge capacitor, Cch.

2) When this capacitor has enough charge, which means a capacitor voltage (VC) of approximately 4 V is reached, then the reset-Schmitt-trigger switches the RST/NMI input of the MSP430 from low to high. The MSP430 program starts with execution

3) The MSP430 program initializes the system and signals its readiness to the host by the transmission of a defined code via the back channel (a second fiber-optic cable).

4) After the receive of the acknowledge, the host sends the first control instruction (data) to the MSP430.

5) The MSP430 executes the received control instruction and sends back the measured result to the host via the back channel.

6) Items 4 and 5 are repeated as often as needed by the host.
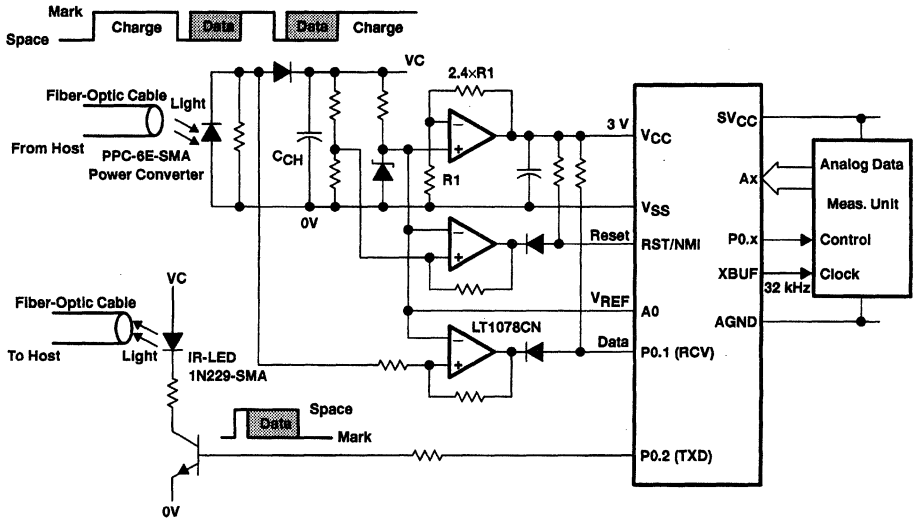
Figure 3–42. Supply via Fiber-Optic Cable

3.8.6.3 Conclusion

The illustrated concepts for supplying the MSP430 family with power demonstrate the numerous ways this can be done. Due to the extreme low power consumption of the MSP430 family, it is possible to supply them with all known power sources. Even fiber-optic cables can be used.

Application Examples

Several MSP430 application examples are given in the following sections. Common to nearly all of them is the storage of calibration data, tables, constants, etc. in the external EEPROMs. External EEPROMs are used for safety reasons. If the microcomputer fails completely, it is still relatively easy to read out the accumulated consumption values. This is usually impossible if these values reside in internal EEPROMs.

These EEPROMs can also store tables that describe the principal errors of a given measurement principle that is dependent on the input value (current, flow, heat etc.). The MSP430, with its excellent table processing capabilities, can determine the right starting value out of these tables and calculate the linear, quadratic or cubic approximation value. The following figure shows the principal error of a meter. The complete range starting at 1% up to 200% is divided into sub ranges of different length. A stored table would contain the starting point, the different distances and the inherent error at the beginning of each range. With this information, the MSP430 can calculate the error at any point of the measurement range.

4.1 Electricity Meters

4.1.1 Overview

The MSP430 can be used in two completely different kinds of electronic electricity meters. The difference between the two methods is mainly where the electrical energy

$$W = \int U \times I \times dt$$

is measured:

❑ The electrical energy is measured in a front-end separated from the MSP430. Several methods exist for doing that: Hall effect sensors, Ferraris wheel pick-ups, analog multipliers, etc. The interface to the MSP430 is normally a train of pulses, where every pulse represents a defined amount of energy (Ws, kWs, Wh). All family members can be used for this purpose.

❑ The electrical energy is calculated by the MSP430 itself, using its 14-bit analog-to-digital converter (ADC) for the measurement of current and voltage. Only the MSP430C32x can be used for this purpose.

The two different methods are shown in Figure 4–1
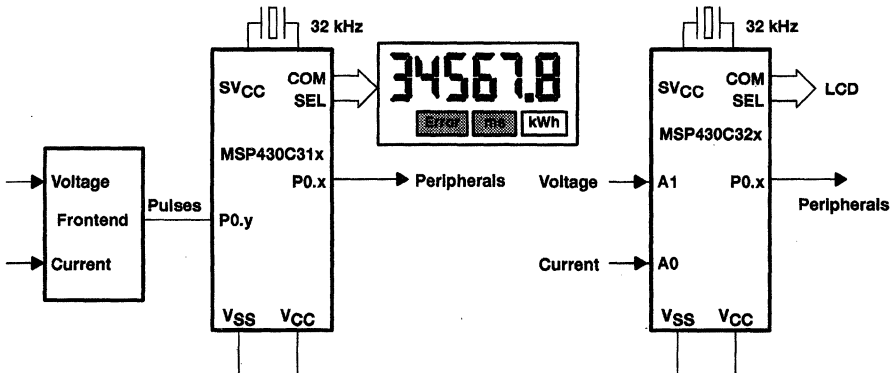
Figure 4–1. Two Measurement Methods for Electronic Electricity Meters

The second method is mainly used with the electricity meters described in this chapter. The unnecessary front end gives a cost advantage when compared

to the two-chip solution. An example for the 1st method that uses a front end is shown at the end of this chapter.

4.1.2 The Measurement Principle

The principle used (Reduced Scan Principle) measures current and voltage in regular time intervals and multiplies the current and voltage samples. The multiplication results are summed up, with the sum representing the consumed energy (Ws, kWh). While the method normally used measures voltage and current at exactly the same time, the Reduced Scan Principle (a protected TI method) alternately measures voltage and current samples. Every sample is used twice; once it is multiplied with the value measured before and once with the value measured afterwards. To further reduce the required multiplications, these two multiplications are reduced to one by using the sum of the two voltage samples. This measurement principle is shown in Figure 4–3.

The following shows the measurement sequence for a single-phase measurement. Current and voltage are measured alternately. The time, α, represents the angle between related voltage and current samples.
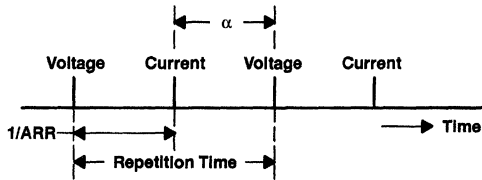
Figure 4–2. Timing for the Reduced Scan Principle (Single Phase)

Where:

α	Inherent Phase Shift of the Measurement Method	[rad]
Repetition Time	Length of a complete measurement cycle	[s]
1/ARR	Time Distance between two ADC Conversions	[s]

Note:

The Reduced Scan Principle is intellectual property of Texas Instruments. This measurement principle may be used only with the microcomputers produced by Texas Instruments.
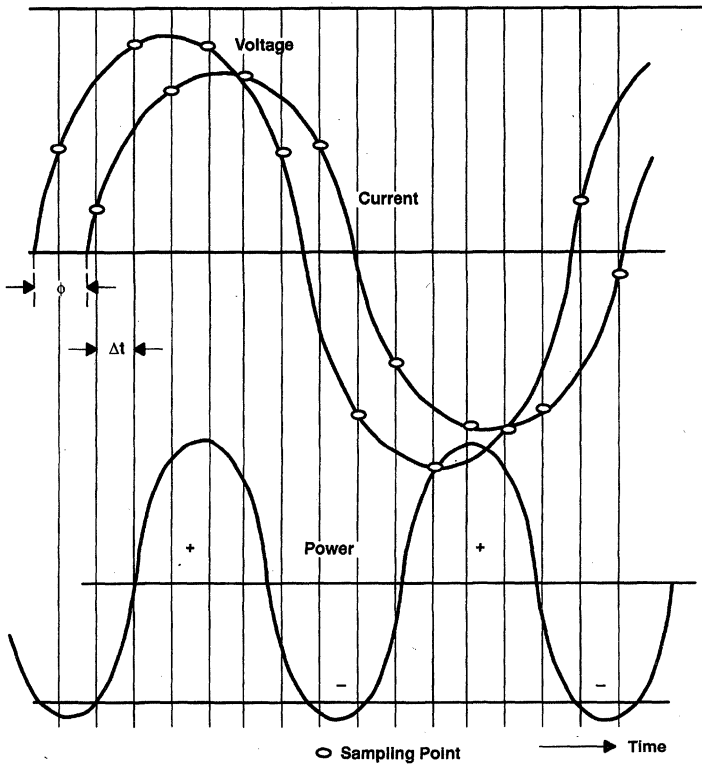
Figure 4–3. Reduced Scan Measurement Principle

The measured energy W (for a single phase) is:

$$W = \sum_{t=0}^{t=\infty} i_n \times \left(u_{n-1} + u_{n+1}\right) \times \Delta t$$

Where:

W	Accumulated energy	[Ws]
i_n	Current sample at time t_n	[A]
u_{n-1}	Voltage sample at time t_{n-1}	[V]
u_{n+1}	Voltage sample at time t_{n+1}	[V]
Δt	Sampling interval between appertaining voltage and current measurements	[s]

4-4

4.1.2.1 The Inherent Error of the Reduced Scan Principle

The Reduced Scan Principle has a small inherent error caused by the phase shift Δt, once inductive and once capacitive, due to the time interval between voltage and current measurements. Any calculated energy sample shows this error, it is independent of the phase angle φ between voltage and current. The value, e, of this error is:

$$e = (\cos (\Delta t \times f \times 2\pi) - 1) \times 100$$

where:

e	Error	[%]
Δt	Sampling interval between voltage and current measurements	[s]
f	AC frequency	[Hz]

For example, with the values ($f = 60$ Hz, $\Delta t = 300$ µs) the inherent error is −0.639%. This error can be eliminated during runtime by a multiplication of the accumulated energy with the correction factor c:

$$c = \frac{1}{\cos(\Delta t \times f \times 2\pi)}$$

The correction factor, c, is normally included in the calibration constants (slope and offset) and not used explicitly.

For a multiple-phase electricity meter, the Reduced Scan Principle is used for all phases one after the other. This is described in the following chapters.

Derivation of the inherent error

The flawless equation (except the quantization error) for the electric energy W is:

$$W = \sum_{t=0}^{t=\infty} i_n \times u_n \times \Delta t$$

The equation used for the Reduced Scan Principle is:

$$W = \sum_{t=0}^{t=\infty} i_n \times \left(u_{n-1} + u_{n+1}\right) \times \Delta t$$

Where:

$u_n = U \times \sin \omega t$ Voltage sample at time t

$i_n = I \times \sin(\omega t + \varphi)$ Current sample at time t

$u_{n-1} = U \times \sin(\omega t - \alpha)$ Voltage sample at time $t - \Delta t$

$u_{n+1} = U \times \sin(\omega t + \alpha)$ Voltage sample at time $t + \Delta t$

α Angle in radians between current and voltage samples ($\alpha = \omega \Delta t = 2\pi \times f \times \Delta t$)

Δt Time between appertaining current and voltage samples

φ Phase angle in radians between voltage and current

The error e of an energy sample due to the Reduced Scan Principle is:

$$e = \frac{erroneous}{correct} - 1$$

$$e = \frac{0.5 \times I \times \sin(\omega t + \phi) \times (U \times \sin(\omega t - \alpha) + U \times \sin(\omega t + \alpha))}{U \times \sin \omega t \times I \times \sin(\omega t + \phi)} - 1$$

$$e = \frac{0.5 \times (\sin(\omega t - \alpha) \times \sin(\omega t + \alpha))}{\sin \omega t} - 1$$

$$e = \frac{0.5 \times (\sin \omega t \times \cos \alpha - \sin \alpha \times \cos \omega t + \sin \omega t \times \cos \alpha + \sin \alpha \times \cos \omega t)}{\sin \omega t} - 1$$

$$e = \frac{0.5 \times (2 \times \sin \omega t \times \cos \alpha)}{\sin \omega t} - 1 = \cos \alpha - 1$$

or in percent

$$e = (\cos \alpha - 1) \times 100 = (\cos(2\pi \times f \times \Delta t) - 1) \times 100$$

This result means that the error of each energy sample calculated with the Reduced Scan Principle shows a constant value e. This inherent error depends only on the angle α between the current and the voltage samples; it is independent of the phase angle φ and of the sample point of the measurement inside the sine wave. So for all samples, the same correction can be used.

4.1.2.2 The Advantages of the Reduced Scan Principle

1) Only 50% of the measurements are necessary because every measured current or voltage sample is used twice

2) Only 50% of the multiplications are necessary because two voltage samples are added before the multiplication

3) Only one ADC is needed compared to up to six with the usual method.

4)   The computing power gained by reducing the number of multiplications can be used by the microcomputer for other system tasks. The MSP430 is able to do the task of the front-end and of the host computer.

5)   The Reduced Scan Principle is nearly independent of frequency deviations of the ac. See Section 4.1.2.4 for results.

6)   The Reduced Scan Principle is also nearly independent of the interrupt latency time of the microcomputer. See Section 4.1.2.5 for results.

The Reduced Scan Measurement Principle is implemented in an evaluation board for a 3-phase meter, which shows a typical error of 0.2%.

4.1.2.3   Measurement Errors for Some Sampling Frequencies

Table 4–1 gives an overview for the measurement errors dependent on the sampling frequency. The inherent error shows the error for the ac frequency (50 Hz or 60 Hz). The 3rd harmonics error shows the corrected measurement error for the 3rd harmonic of the ac frequency (150 Hz or 180 Hz). The 5th harmonics error shows the corrected measurement error for the 5th harmonic of the ac frequency (250 Hz or 300 Hz). For any number of measurements (current and voltage samples together) for a full period, a rough error estimation can be made with this table.

Table 4–1. Errors Dependent on the Sampling Frequency

Measurements per Full Period	Sample Frequencies			Errors		
	Single Phase (50Hz)	Two Phase (60Hz)	Three Phase (50Hz)	Inherent Error	3rd Harmonic†	5th Harmonic†
20	1000	2400	3000	–4.89%	–36.4%	–95.2%
30	1500	3600	4500	–2.19%	–16.9%	–47.8%
40	2000	4800	6000	–1.23%	–9.7%	–28.0%
50	2500	6000	7500	–0.78%	–6.2%	–18.3%
60	3000	7200	9000	–0.55%	–4.3%	–13.4%
70	3500	8400	‡	–0.40%	–3.2%	–9.5%
80	4000	9600		–0.30%	–2.4%	–7.3%
90	4500	‡		–0.24%	–1.9%	–6.0%
100	5000			–0.20%	–1.6%	–4.7%
110	5500			–0.16%	–1.3%	–3.9%
120	6000			–0.13%	–1.1%	–3.2%
130	6500			–0.11%	–0.9%	–2.7%
140	7000			–0.10%	–0.8%	–2.4%
160	8000			–0.08%	–0.6%	–1.9%
180	9000			–0.06%	–0.5%	–1.5%
200	10000			–0.05%	–0.4%	–1.2%

† The errors of the harmonics are corrected by the value of the inherent error
‡ Sampling frequencies above 10000Hz are not possible due to the speed of the ADC
  (132 ADCLKs/conversion @ ADCLK = 1.5MHz)

4.1.2.4 Measurement Error for Deviations of the AC Frequency

If the ac frequency deviates from the nominal value used during the calibration, then a small error is generated. Table 4–2 shows this error dependent on the sample frequency and the ac frequency deviation. The introduced error, Fmd, is:

$$F_{md} = \left(\frac{\cos(\Delta t \times (f + \Delta f) \times 2\pi)}{\cos(\Delta t \times f \times 2\pi)} - 1 \right) \times 100$$

Where:

F_{md}	Error due to the ac frequency deviation from the nominal frequency	[%]
Δt	Time between related current and voltage samples	[s]
f	Nominal ac frequency (used during calibration)	[Hz]
Δf	Frequency deviation of the ac frequency during runtime	[Hz]

Table 4–2. Errors dependent on the AC Frequency Deviation

Measurement per full Period	Sample Frequencies			Errors		
	Single Phase (50Hz)	Two Phase (60Hz)	Three Phase (50Hz)	$\Delta f/f = +0.5\%$	$\Delta f/f = +1.0\%$	$\Delta f/f = +5.0\%$
20	1000	2400	3000	−0.051%	−0.103%	−0.523%
40	2000	4800	6000	−0.012%	−0.025%	−0.127%
80	4000			−0.003%	−0.006%	−0.030%
130	6500			−0.001%	−0.002%	−0.010%

The errors for negative frequency deviations are the same as shown in Table 4–2 but with positive signs. The ADC is assumed to be error-free, this way only the influence of the frequency deviation is shown.

The additional error due to the deviation of the ac frequency can be reduced to nearly zero by the measurement of the actual ac frequency and an appropriate correction of the calculated energy.

4.1.2.5 Measurement Error Dependent on the Interrupt Latency Time

The calibration of an electricity meter is made normally in an environment without interrupt activity. This can be completely different to the real time environment where the meter has to measure the electric energy later. Therefore the interrupt latency time (here the time the interrupt request of the sampling time base is delayed by other interrupts) can have an influence on the accuracy of the measurement. Table 4–3 shows the errors introduced by different interrupt latency times. The calibration is made with a maximum interrupt latency time of 5µs (due to missing interrupt activities): this is the maximum delay caused by the completion of the current instruction (indexed,indexed mode) with MCLK = 1MHz. The conditions used for the simulations of Table 4–3 are:

❑ The simulation conditions are the same ones as described in section 4.1.3 except where noted otherwise.

❑ The given interrupt latency times are the maximum values; each voltage and current sample is delayed by a random time interval ranging between zero and this maximum value.

❑ The ADC is assumed to be error-free (except the range transition error), this way only the influence of the interrupt latency time is shown.

□ For other values of MCLK than 1MHz , the shown latency times are not given in microseconds but CPU cycles.

□ The used current is 100% except for the last line (1%)

□ The measurement time is 20 seconds

Table 4–3. Errors dependent on the Interrupt Latency Time

Measurement per Full Period	Single Phase (50 Hz)	Maximum Interrupt Latency Time				
		5 µs (Calibr.)	20 µs	40 µs	80 µs	160 µs
20	1000	–0.0013%	–0.0010%	+0.0023%	+0.0052%	+0.0103%
40	2000	–0.0010%	+0.0010%	–0.0005%	–0.0053%	–0.0113%
80	4000	+0.0007%	+0.0002%	–0.0035%	–0.0053%	–0.0292%
130	6500	–0.0011%	+0.0002%	–0.0006%	–0.0025%	–†
cos φ=0.5	6500	–0.0011%	0%	+0.0001%	–0.0055%	–†
1% I_n	6500	–0.0098%	–0.0175%	+0.0170%	–0.0786%	–†

† Interrupt latency time is greater than sampling interval

Table 4–3 shows the extreme low influence of the interrupt latency time: even non-realistic high latency times like 160 µs result in negligible influence. This means that the Reduced Scan Principle is not sensitive to the interrupt latency time of the system.

Note:

The errors shown in Table 4–3 are won by the use of random values for the interrupt latency time. Despite the relatively long simulation time (20 seconds) every simulation made under exactly the same conditions returned therefore a slightly different error.

4.1.2.6 Measurement Error Due to Overvoltage and Overcurrent

With the simulation conditions described in Section 4.1.3, *The Analog-to-Digital Converter of the MSP430C32x*, the ADC measures up to 111% of the maximum current or voltage without additional error. It is important to know how the electricity meter behaves if the input values are above these limits: there must be a smooth transition and no oscillations or sudden changes. Due to the saturation the ADC shows for overflow and underflow, the errors shown in Table 4–4 result. The ADC is assumed to be error-free (with the exception of the range transition error), so only the effect of the overflow is shown.

Table 4–4. Errors dependent on Overvoltage and Overcurrent

Load Current	100% Vnom	110% Vnom	120% Vnom	130% Vnom
100%	0%	0%	–2.4%	–6.5%
110%	0%	0%	–2.4%	–6.5%
120%	–2.4%	–2.4%	–4.7%	–8.6%
130%	–6.5%	–6.5%	–8.6%	–12.3%

4.1.3 The Analog-to-Digital Converter of the MSP430C32x

The analog-to-digital converter (ADC) of the MSP430 measures the voltage between its AVss and SVcc connections with a resolution of 14 bits. The signed voltages coming from the current and voltage interfaces are shifted into the unsigned range of the ADC by simple interfaces described below. The MSP430 subtracts the measured or calculated offset value from every measured current or voltage sample: this enables signed, offset corrected measurements.
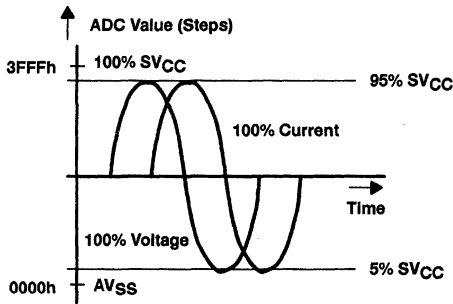
Figure 4–4. Allocation of the ADC Range

Figure 4–4 shows the placement of the current and voltage coming from the voltage dividers and the current interfaces into the analog-to-digital converter's range. All calculations and proposals base on a use of 90% of the ADC range for nominal (100%) values of current and voltage. This means up to 111% of the nominal values are still measured correctly. This allocation may be changed if necessary.

Table 4–5 shows the influence of the analog-to-digital converter's performance to the accuracy of the measurement of the electric energy. Two influences are involved:

1) The deviation of the ADC from the linearity. Each one of the four ranges A, B, C and D has calculated deviations up to 20 ADC steps compared to the two ranges bordering on it.

2) The saturation effect at the range limits: if the sample for the definition of the range is taken in another range than the sample for the 12-bit conversion (36 ADCLKs later) than the result is xFFFh for increasing input signals and x000h for decreasing input signals (x denotes the number of the range where the range sample was taken). As the results show, the two saturation effects compensate nearly to zero.

Note:

The deviations of the analog-to-digital converter used with the examples below (±20 steps) are greater than the specified ones. These large deviations are used only to show the relative independence of the overall accuracy from the ADC error. The actual, specified deviations are ±10 steps.

It is recommended not to use the exact midpoint of the supply voltage Vcc (Vcc/2) for the common reference point. This is due to the possible slight slope deviation at the border of two ADC ranges (here B and C). This may influence the accuracy for the lowest currents.

Table 4–5 shows also the influence for some extreme deviations of the analog-to-digital converter characteristic. Figure 4–5 explains the meaning of the used graphics: it shows the second deviation curve of Table 4–5 in detail.
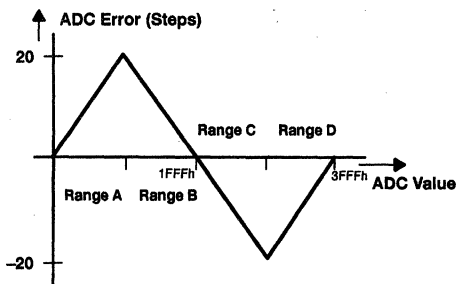
Figure 4–5. Explanation of ADC Deviation (2nd Column of Table 4–5)

The function shows the deviation at any point of the four ADC ranges. Due to the monotony of the ADC the errors at the range limits are always equal. The errors shown in Table 4–5 were calculated with a PASCAL program. The following steps were taken:

1) Measurement and calculation of the error at 5% of the nominal current.

2) Measurement and calculation of the error at 100% of the nominal current

3) Calculation of the slope and offset for the correction (calibration)

4) Simulation of voltage and current samples: any sample is modified with the ADC error (exactly like during calibration).

5) Correction of all measured values with the calculated slope and offset

6) Calculation of the resulting error

The saturation effect at the range limits is always included. The first column of Table 4–5 with an ideal ADC characteristic (zero deviation) shows only this effect and the finite ADC resolution. This column can be used as a reference for the errors of the other five columns.
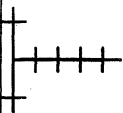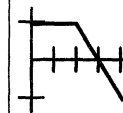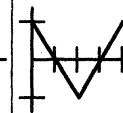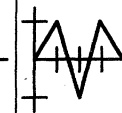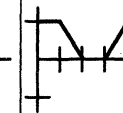
The calculations are made with the following conditions:

- Virtual Ground location in the ADC range: 8190 steps (1FFEh) 49.98% of full ADC range
- Measurement time for calibration points: 5s (calibration points are measured this time)
- Measurement time for different loads: 9s
- AC frequency: 50 Hz
- Cosine φ: 1 (0°)
- Sample frequency: 2048 Hz (488.3 µs sample distance)
- Voltage: 100% V_{pp} uses 90% of the ADC range
- Current: 100% I_{pp} uses 90% of the ADC range

Note:

The drawings on top of the columns of Table 4–5 indicate the ADC error in dependence of the ADC value. Figure 4–5 shows the drawing above the second column in a magnified form.

Table 4–5. Errors With One Current Range and Single Calibration Range

Load Current						
0.1%	+0.7771%	+7.79%	−2.93%	+0.45%	+0.57%	+3.94%
1%	−0.0114%	+0.83%	−0.24%	0%	+0.01%	+0.38%
2%	+0.0620%	+0.50%	+0.01%	+0.01%	0%	+0.24%
Calibr. P. 5%	−0.0001%	0%	0%	0%	0%	0%
10%	+0.0005%	−0.19%	−0.01%	0%	0%	−0.09%
25%	0%	−0.27%	−0.01%	%	0%	−0.13%
50%	−0.0001%	−0.31%	−0.01%	0%	0%	−0.15%
75%	+0.0001%	−0.17%	0%	0%	0%	−0.09%
Calibr. P. 100%	−0.0002%	0%	0%	0%	0%	0%

The large errors at 0.1% of the nominal current result from the relatively far distance from the 5% calibration point and from the missing resolution of the ADC at this small load. The peak-to-peak value of the ADC result is only 14.7 steps. These errors can be reduced drastically by using one of the following methods.

4.1.3.1 Methods to reduce the Error of the Energy Measurement

Three relatively simple methods are given to reduce the error of the energy measurement. In any case, the values used for the correction are stored in the EEPROM and are loaded into the RAM during the initialization.

Using a Second Hardware Range
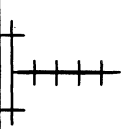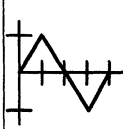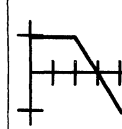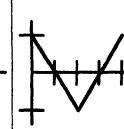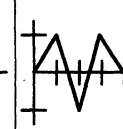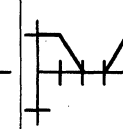
This method is shown with all hardware examples. An analog switch like the TLC4016 switches a second resistor in parallel to the one used for the low current range. Both ranges uses its own set of calibration constants (slope and offset) that are measured during two independent calibration runs for every phase. The advantage of this method is the real increase of resolution for the low current range.

Using a Second Calibration Range

This method only uses a second set of calibration constants (slope and offset) without additional hardware for the low current range (e.g., from 0.1% to 5% of the nominal current). This method needs two calibrations per phase, but uses only three measurements (one measurement is used for both ranges).

Table 4–6 shows the enhancement of the accuracy when a second calibration run is made for the low current range, 0.1% to 5% of the nominal value. The calculations are made with the same conditions used with Table 4–5. The enhancement can be seen with a comparison of the two tables. The errors, for the range 5% to 100% of the nominal current, are the same as shown in Table 4–5.

Table 4–6. Errors With One Current Range and Two Calibration Ranges

Load Current						
Calibr. P. 0.1%	+0.004%	+0.004%	+0.002%	+0.005%	+0.005%	+0.003%
0.5%	−0.236%	−0.002%	−0.251%	−0.163%	−0.161%	−0.119%
1%	−0.075%	+0.190%	−0.003%	−0.041%	−0.040%	+0.058%
2%	−0.018%	+0.262%	+0.098%	−0.005%	−0.012%	+0.122%
3%	−0.006%	+0.062%	+0.024%	−0.013%	−0.012%	+0.022%
4%	−0.010%	−0.035%	−0.025%	−0.009%	−0.007%	−0.023%
Calibr. P. 5%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%

Measurement of the ADCs Characteristic

This method uses the actual deviations of the ADC for a rough correction of the measurement results. During a first run, the ADC characteristic is measured and correction constants are calculated for any of 8 to 32 software subranges of the ADC. These correction constants are written into the EEPROM and loaded into the RAM for use. For every subrange, one byte is needed, which allows corrections up to ±127 steps. The correction for the samples needs only seven instructions per 14-bit value. The advantage of this method is the adaptation to the actual deviation of the individual ADC. Figure 4–6 shows the correction with the ADC characteristic using only 8 correction values. The deviations reduce to one quarter of the original ones. If the correction shows a step near the virtual zero point like shown in Figure 4–6, the subranges B1 and C0 can be corrected in a way that omits this step. Chapter 2, *The Analog-To-Digital Converters* gives more information.
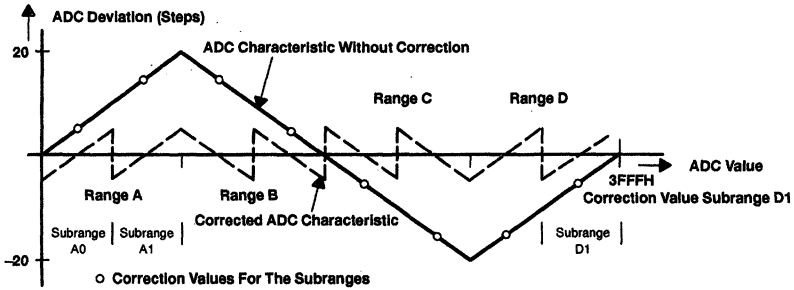
Figure 4–6. Use of the Actual ADC Characteristic for Corrections (8 Subranges Used)

4.1.3.2 Dependence on the Voltage and the Phase Angle φ

Table 4–7 shows the dependence of the MSP430 using the Reduced Scan Principle on the load current, the ac voltage and the phase angle, φ, between current and voltage. The ADC is assumed to be error-free; the saturation effect at the range limits is included. Single calibration with only one range is used. Nominal voltage is used for the load current dependence and nominal current (100%) is used with the voltage dependence. The calculations are made with the same conditions used for the calculations in Table 4–5.

Table 4–7. Errors in Dependence on Current, Voltage and Phase Angle

Angle φ	Load Current			AC Voltage		
	1%	10%	100%	80%	90%	110%
Ind. −80°	+4.119%	+0.447%	+0.046%	+0.048%	+0.047%	+0.045%
−60°	+0.857%	+0.099%	+0.010%	+0.009%	+0.010%	+0.010%
−40°	+0.257%	+0.032%	+0.003%	+0.003%	+0.003%	+0.004%
−20°	+0.047%	+0.009%	+0.001%	0.000%	+0.001%	+0.001%
0°	−0.011%	+0.001%	0.000%	0.000%	0.000%	0.000%
+20°	+0.043%	+0.004%	0.000%	+0.001%	+0.001%	0.000%
+40°	+0.248%	+0.021%	0.000%	+0.004%	+0.003%	+0.001%
+60°	+0.844%	+0.075%	+0.007%	+0.012%	+0.009%	+0.005%
Cap. +80°	+4.051%	+0.376%	+0.037%	+0.056%	+0.046%	+0.031%

4.1.3.3 Derivation of the Measurement Formulas

The electronic meter equivalent of the meter constant of a Ferraris wheel meter (revolutions per kWh) is the meter constant, C_Z, that defines (ADC steps)^2 per Ws. The corrected equation used for the electric energy W is:

$$W = \frac{1}{\cos(2\pi \times f \times \Delta t)} \times \sum_{t=0}^{t=\infty} i_n \times \left(u_{n-1} + u_{n+1}\right) \times \Delta t \qquad [\text{Ws}]$$

With the ADC results ADCi (current sample) ADCu (voltage sample) and ADC0u and ADC0i (zero volt samples) the previous equation gets:

$$W = \frac{1}{\cos(2\pi \times f \times \Delta t)} \times \sum_{t=0}^{t=\infty} ki \times \left(ADCi_n - ADC0i\right) \times ku \times \left(ADCu_{n-1} + ADCu_{n+1} - 2 \times ADC0u\right) \times \Delta t$$

Separation into variable and constant values results in:

$$W = \frac{1}{\cos(2\pi \times f \times \Delta t)} \times \Delta t \times ki \times ku \times \sum_{t=0}^{t=\infty} \left(ADCi_n - ADC0i\right) \times \left(ADCu_{n-1} + ADCu_{n+1} - 2 \times ADC0u\right)$$

Where:

f	AC frequency	[Hz]
Δt	Sampling interval between appertaining voltage and current samples	[s]
ki	Current multiplication factor See Section 4.1.4.5 for more details	[A/step].
ku	Voltage multiplication factor. See section 4.1.4.6 for more details	[V/step]
$ADCi_n$	ADC value of current sample taken at time t_n	
$ADCu_{n-1}$	ADC value of voltage sample taken at time t_{n-1} ($t_n - \Delta t$)	
$ADCu_{n+1}$	ADC value of voltage sample taken at time t_{n+1} ($t_n + \Delta t$)	
ADC0u	ADC value of voltage zero point (measured or calculated)	
ADC0i	ADC value of current zero point (measured or calculated)	

The first, constant part of the equation is the inverse value of the meter constant, C_Z:

$$C_Z = \frac{\cos(2\pi \times f \times \Delta t)}{\Delta t \times ki \times ku} \qquad [\text{Steps}^2/\text{Ws}]$$

The values for ki and ku for different interfaces are explained in detail in Section 4.1.4.

For a system using a current transformer and a resistor divider for the voltage, the previous equation gets:

$$W = \frac{1}{\cos(2\pi \times f \times \Delta t)} \times \Delta t \times \frac{SV_{CC} \times wsec}{2^{14} \times w_{prim} \times Rsec} \times \frac{SV_{CC} \times (Rm + Rc)}{2^{14} \times Rc} \times$$

$$\sum_{t=0}^{t=\infty} \left(ADCi_n - ADC0i\right) \times \left(ADCu_{n-1} + ADCu_{n+1} - 2 \times ADC0u\right)$$

Where:

Rsec	Load resistor (secondary) of the current transformer	[Ω]
w_{sec}	Secondary windings of the current transformer	
w_{prim}	Primary windings of the current transformer	
SV_{CC}	Voltage at terminal SV_{CC} (AV_{CC} or external reference voltage)	[V]
Rm	Voltage divider: resistor between ac connection and analog input	[Ω]
Rc	Voltage divider: resistor between analog input and zero volts	[Ω]

The first, constant part of the equation is the inverse value of the meter constant C_Z:

$$C_Z = \frac{\cos(2\pi \times f \times \Delta t) \times 2^{28} \times w_{prim} \times Rsec \times Rc}{\Delta t \times SV_{CC}^2 \times w_{sec} \times (Rm + Rc)} \qquad [\text{Steps}^2/\text{Ws}]$$

With the previous value of C_Z, the equation for the energy W is:

$$W = \frac{\sum_{t=0}^{t=\infty}(ADCi_n - ADC0i) \times (ADCu_{n-1} + ADCu_{n+1} - 2 \times ADC0u)}{C_Z} \qquad [\text{Ws}]$$

If the energy W is to be expressed in kWh:

$$W = \frac{\sum_{t=0}^{t=\infty}(ADCi_n - ADC0i) \times (ADCu_{n-1} + ADCu_{n+1} - 2 \times ADC0u)}{3.6 \times 10^9 \times C_Z} \qquad [\text{kWh}]$$

The value W needs to be corrected with the slope and offset calculated during the calibration process.

4.1.4 Analog Interfaces to the MSP430

This chapter describes some important topics that can affect the overall accuracy of the electricity meter.

4.1.4.1 Analog and Digital Grounding

The following schematics are drawn in a simplified manner to make them easier to understand. In reality, it is necessary to decouple the analog and the digital part as shown in Figure 4–7. This is to avoid digital noise on the analog signals to be measured.
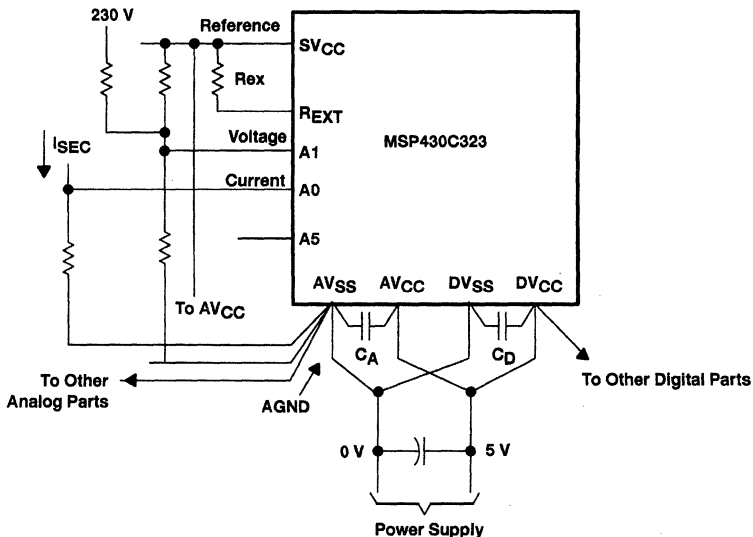
Figure 4–7. MSP430 14-Bit ADC Grounding

4.1.4.2 ADC Input Considerations

The ADC accurately operates up to 1.5 MHz. If the processor clock MCLK is higher than this frequency, it is recommended that one of the prescaled ADC clocks (ADCLK) be used. The possible prescaled frequencies for the ADCLK are MCLK, MCLK/2, MCLK/3 and MCLK/4.

The sampling of the ADC to get the range information takes 12 ADCLK cycles. This means, the sampling gate is open during this time (12 μs at ADCLK = 1 MHz). The input of an ADC terminal can be seen as an RC low-pass filter, 2 kΩ together with 42 pF. The 42-pF capacitor must be charged during the 12 ADCLK cycles to the final value in order to be measured. This means charged within 2^{-14} of this value. This time limits the internal resistance R_I of the source to be measured:

$$(Ri + 2 \text{ k}\Omega) \times 42 \text{ pF} \ < \ \frac{12}{\ln 2^{14} \times ADCLK}$$

Solved for R_I, the result is 27.4 kΩ. This means, to get the full 14-bit resolution of the ADC, the internal resistance of the input signal must be lower than 27.4 kΩ. The given examples use lower source resistances at the ADC inputs.

4.1.4.3 Offset Treatment

If the voltage and current samples contain offsets, the equation for the measured energy W is:

$$W = \sum_{t=0}^{t=\infty} \left(u_n + O_u\right) \times \left(i_n + O_i\right) \times \Delta t$$

$$W = \sum_{t=0}^{t=\infty} \left(u_n \times i_n + u_n \times O_i + i_n \times O_u + O_i \times O_u\right) \times \Delta t$$

Where:

O_u Offset of voltage measurement   [V]
O_i Offset of current measurement   [A]
u_n Sum of the two voltage samples u_{n-1} and u_{n+1} [V]

The terms $(u_n \times O_i)$ and $(i_n \times O_u)$ get zero when summed-up over one full period (the integral of a sine curve from 0 to 2π is zero) but the term $(O_i \times O_u)$ is added erroneously to the sum buffer with each sample result. If one of the two offsets can be made zero then the error term $(O_i \times O_u)$ is eliminated: this is the case for all proposals. Two different ways are used:

❏ Voltage representing 0V is measured (see Sections 4.1.4.4.1 and 4.1.4.4.2)

❏ Summed-up ADC value for a full period is used for this purpose (see Section 4.1.4.4.3).

4.1.4.4 Adaptation to the Range of the Analog-to-Digital Converter

The analog-to-digital converter of the MSP430 is able to measure unsigned voltages ranging from AVss up to the reference voltage applied to the input SVcc. If signed measurements, as for electricity meters, are necessary then a virtual zero point must be provided. Voltages above this zero point are treated as positive ones, voltages below it are treated as negative voltages. A few possibilities are shown how to provide this virtual zero point. For more information see Section 3.8, *Power Supplies for the MSP430*.

Split Power Supply

To get a common reference voltage in the middle of the ADC's voltage range, two voltage regulators with output voltages of +2.5 V and –2.5 V can be used. In this case, the common zero connection is the reference for all current and voltage measurements. This zero point is connected to one of the analog inputs (A0 in Figure 4–8). The measured ADC value of this reference voltage is

subtracted from every voltage and current sample. This way signed, offset corrected measurement values are generated.

The schematic is shown in Figure 4–8.
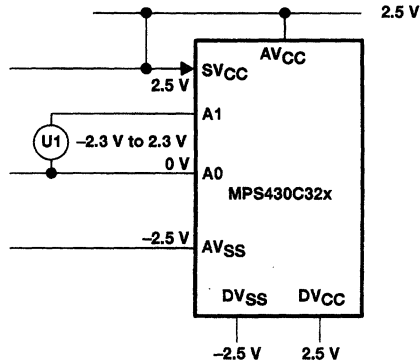
Figure 4–8. Split Power Supply for Level Shifting

Use of a Virtual Ground IC

A virtual ground IC can be used to get a measurement reference in the middle of the ADC range. The TLE2426 is used for this purpose. All current and voltage inputs are referenced to the virtual ground output of this circuit. The main advantage is the ability to measure the ADC value of this reference without the need to switch off the voltage and current inputs.

The measured value (at analog input A0), is subtracted from every measured current or voltage sample, which generates signed, offset corrected results (see Figure 4–9).

Typical electrical characteristics of the TLE2426:

Supply Current	170 µA	No load connected
Output Impedance	0.0075 Ω	
Output Current Capability	±20 mA	For sink and source
Power Rating at 25°C	725 mW	For the Small Outline Package
Derating Factor above 25°C	5.8 mW/°C	
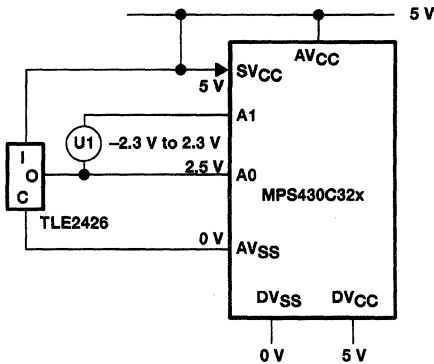
Figure 4–9. Virtual Ground IC for Level Shifting

Resistor Interface (Software Offset)

This method uses the fact that the integral of a sine curve is zero, if integrated over the angle 2π. Two counters add up the ADC results separately for each voltage and current signal. These counters contain the two offsets (in ADC steps) after a full period of the ac frequency. These offsets are subtracted from the appertaining ADC samples. The results are signed, offset corrected samples. The current and voltage signals are shifted into the middle of the ADC range by simple voltage dividers or with the help of the internal current source.

Without A Current Source

The necessary shift of the signed voltage and current signals is made by resistor dividers. The resistor divider of the voltage part is also used for the adaptation of the ac voltage to the ADC range. The current part allows two (or more) current ranges. With the closed range switch, high currents can be measured. With the switch open, a better resolution for the low currents is possible. No dc flows through the current transformer due to the high input resistance of the ADC inputs.
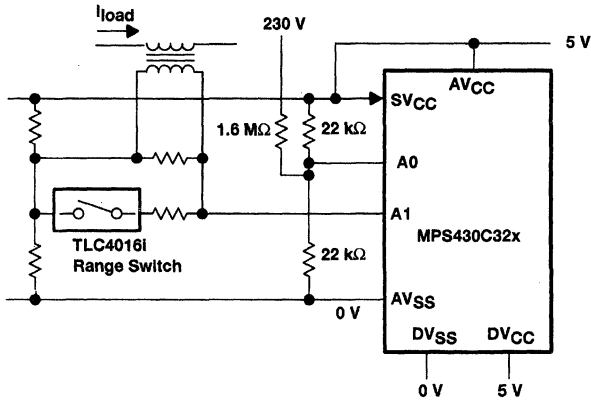
Figure 4–10. Resistor Interface Without Current Source

With A Current Source

Four ADC inputs can be used with the internal current source. A current, defined by an external resistor Rex, is switched to the ADC input and the voltage drop at the external circuitry is measured with the ADC. This current is relative to the reference voltage SVcc and delivers constant results also with different values of SVcc. If a second current range is needed, a reed relay is needed to switch the second load resistor of the current transformer.

Note:
The signal at the current transformer has a negative going part—outside of the ADC voltage range—therefore a TLC4016 cannot be used).

The current Ics flows through the current transformer's secondary windings. This will need to be checked to see if it is usable.

Iload   230 V

SVCC   AVCC   5 V

1.6 MΩ   Rex

REXT

A0

ICS   MPS430C32x

A1   1.25 V

5.6 kΩ

AVSS

DVSS   DVCC

$$I_{cs} = \frac{SV_{CC}}{4 \times Rex}$$ 0 V

0 V   5 V
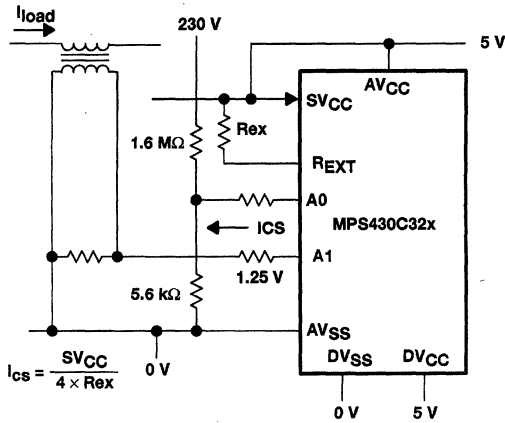
Figure 4–11. Resistor Interface With Current Source

Note:

If the current source is used, only ADC ranges A and B can be used. This is because of the supply voltage the current source needs for operation. The resolution is therefore only one-half of the normal value. The midpoint of the ADC range is then 01000h.

4.1.4.5   Current Measurement

The main problem of the current measurement is the large dynamic range of the input values; ranging from 0.1% up to 1000% of the nominal value. The common methods used to solve this problem are shown in Figure 4–12 and are explained in the following text. If range switches are used, it is recommended that a hysteresis for the range selection criteria be used.

Shunt

The load current I_L flows through a resistor Rshunt (0.3 mΩ to 3.0 mΩ) and the voltage drop of this resistor (shunt) is used for the current measurement. Due to the small voltage drop, especially with low currents, it is necessary to amplify this voltage drop with an operational amplifier. This operational amplifier can have only a very small phase shift (0.1°) to get the needed accuracy. The out-

put voltage V_{Out}, which is proportional to the current I_L, is measured by the MSP430. The amount of V_{Out} is:

$$V_{out} = -I_{load} \times R_{shunt} \times \frac{R2}{R1} \qquad \text{(open switch, low current)}$$

$$V_{out} = -I_{load} \times R_{shunt} \times \frac{R2||R3}{R1} \qquad \text{(closed switch, high current)}$$

The value ki [A/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is:

$$ki = -\frac{SV_{CC}}{2^{14}} \times \frac{R1}{R_{shunt} \times R2} \qquad \text{(open switch, low current, see Figure 4–12)}$$

$$ki = -\frac{SV_{CC}}{2^{14}} \times \frac{R1}{R_{shunt} \times R2||R3} \qquad \text{(closed switch, high current)}$$

❏ Advantages

   ■ Resistive behavior

   ■ Simple

   ■ More than one range possible with switches

❏ Disadvantages

   ■ High losses with high currents

   ■ Very low output voltage with small currents (amplifier necessary)

   ■ Only usable with single-phase meters
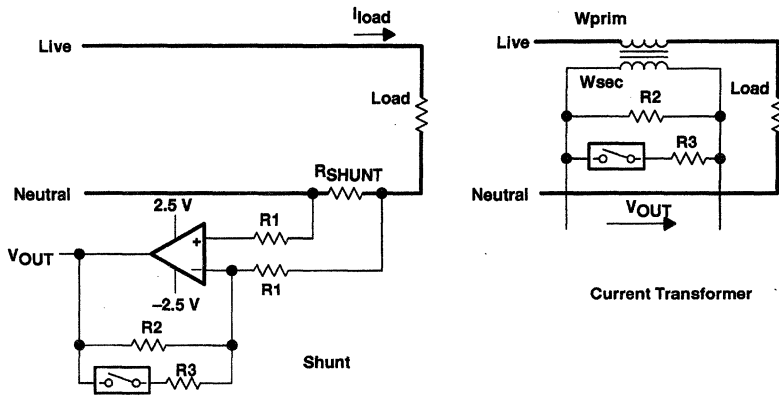
Figure 4–12. Current Measurment

Current Transformer

The secondary current I_{sec} of the current transformer, which is

$$I_{sec} = \frac{w_{prim}}{w_{sec}} \times I_{load}$$

flows through a resistance Rsec (the resulting resistance of the two resistors R2 and R3) and generates a voltage V_{OUT}, which is measured by the MSP430:

$$V_{out} = \frac{w_{prim}}{w_{sec}} \times I_{load} \times R_{sec}$$

Where:

Rsec = R2   (switch open, low currents)
Rsec = R2∥R3   (switch closed, high currents)

The value ki [A/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is:

$$ki = -\frac{SV_{CC}}{2^{14}} \times \frac{w_{sec}}{R_{sec} \times w_{prim}} \qquad \text{(see Figure 4–12)}$$

❑ Advantages

■ Isolation from ac

■ High accuracy for the magnitude of the current (0.1% reachable)

■ More than one range possible with switched resistors

❑ Disadvantages

■ Sensible to dc current: may lead to saturation

■ Costly

Ferrite Core

The load current I_{load} flows through a ferrite core with a single winding. The ferrite core has a small air gap. The magnetic flux crossing this air gap goes through an air-core coil, which is not loaded. The small output voltage V_{fc} of this coil is amplified, integrated, and measured by the MSP430. The voltage gain of the preamplifier is used for the range switching. The ferrite core behaves as an inductivity L i.e. the output voltage V_{fc} is:

$$V_{fc} = \frac{dI_{load}}{dt} \times L$$

This means, the voltage V_{fc} has a leading phase shift of 90° compared to I_{load}. This phase shift can be corrected by two methods:

1) Software shift: All current samples are delayed by the time representing 90° of the ac frequency. This is possible with a circulating buffer and a carefully chosen sampling frequency.

2) Analog shift: An integrator combined with a pre-amplifier is used as shown in Figure 4–13.

The value ki [A/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is :

$$ki = -\frac{SV_{CC}}{2^{14}} \times \frac{C \times R1}{v \times L}$$

The formula is valid only if R2 >> R1 (normal case).

❑ Advantages

■ Isolation from the ac

■ No saturation possible by dc parts of the load current due to the air gap

❑ Disadvantages

■ Low output voltage due to loose coupling

■ Output voltage leads 90° compared to load current

■ Fast load current changes cause relatively high output voltages (di/dt)

■ Circular buffering or amplification and integration necessary
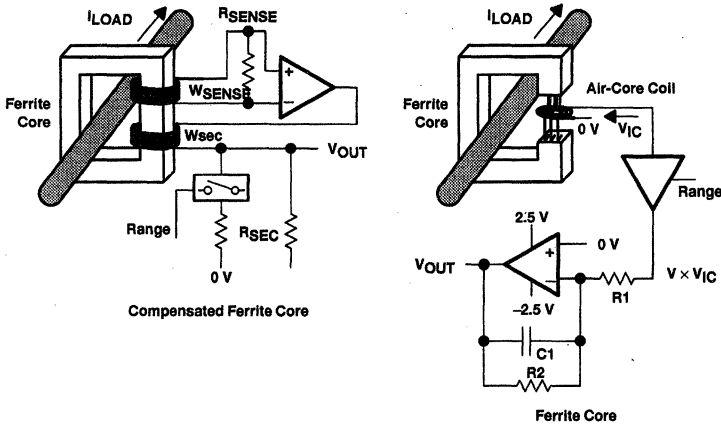
Figure 4–13. Current Measurment With a Ferrite Core

Compensated Ferrite Core

The load current Iload flows through a closed ferrite core with a primary winding w_{prim} (normally a single winding). The magnetic flux created by the primary winding is sensed by the sensor winding w_{sense}. The voltage of the sense winding is amplified and the output current of the amplifier is sent through the secondary winding w_{sec} in a way that compensates the primary flux to (nearly) zero. This means that the driving of the resistor R_{sec} is made by the amplifier and not by the ferrite core. The compensated ferrite core shows only negligible errors. It is only necessary to distribute the two windings in a very equable way over the entire core (not as it is shown in Figure 4–13 for simplicity). Additional current ranges are possible with switched resistors in parallel with R_{sec}. The output voltage V_{out} is:

$$V_{out} = I_{load} \times R_{sec} \times \frac{w_{prim}}{w_{sec} + \frac{w_{sense} \times R_{sec}}{v \times R_{sense}}}$$

The term $w_{sense} \times R_{sec}/v \times R_{sense}$ is the remaining error of the compensated ferrite core.

The value ki [A/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is (the error term is not included due to its low value):

$$ki = -\frac{SV_{CC}}{2^{14}} \times \frac{1}{R_{sec}} \times \frac{w_{sec}}{w_{prim}}$$

❏ Advantages

■ Isolation from ac

■ Nearly complete compensation of the ferrite core's hysteresis and nonlinearity errors

❏ Disadvantages

■ Amplifier necessary

■ Difficulties to stabilize feedback loop

4.1.4.6 Voltage Measurement

The problem of the current measurement, the large dynamic range, does not exist for the voltage measurement. AC voltage always has a nearly constant value. Two measurement methods are used normally.
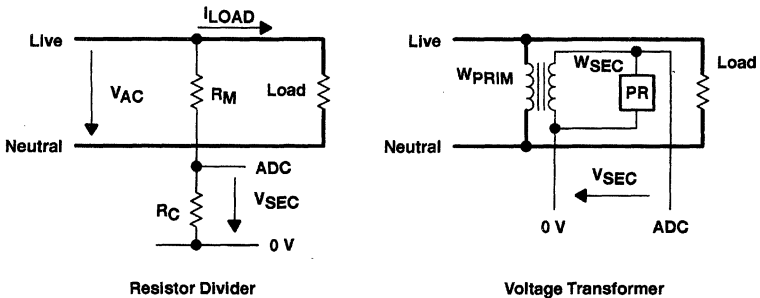
Figure 4–14. Voltage Measurement

Resistor Divider

The ac voltage Vac is adapted to the range of the ADC by a simple resistor divider. All of the examples given use this method. The amount of Vsec is:

$$V_{sec} = \frac{Rc}{Rm + Rc} \times V_{ac}$$

The value ku [V/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is:

$$ku = \frac{SV_{CC}}{2^{14}} \times \frac{Rm + Rc}{Rc} \qquad \text{[V/step] (see Figure 4–14)}$$

Voltage Transfomer

A voltage transformer is used if the ac voltage is very high or if galvanic isolation is needed. Protection (PR) at the secondary side is needed, due to the low output impedance of the voltage transformer.

The amount of Vsec is:

$$V_{sec} = \frac{w_{sec}}{w_{prim}} \times V_{ac}$$

The value ku [V/step], used for the calculation of the meter constant C_Z (see Section 4.1.3.3) is:

$$ku = \frac{SV_{CC}}{2^{14}} \times \frac{w_{prim}}{w_{sec}} \qquad \text{[V/step] (see Figure 4–14)}$$

4.1.5 Single-Phase Electricity Meters

The next two electronic electricity meter proposals are made for the measurement of European ac. From the utility, one phase and ground are wired into the house. In this way a nominal voltage of 230 V is available.

The reduced scan principle is applied exactly as described in Section 4.1.

To measure the electric energy consumed, a current transformer or a shunt resistor is necessary, both solutions are shown. The voltage of the phase is also measured. With this configuration, the energy consumption of the load can be measured exactly.

The measurement sequence for a single-phase meter is shown in Figure 4–2.

The ADC of the MSP430 measures the voltage between the AVss and SVcc connections with a resolution of 14 bits. To shift the signed voltages coming

from the current transformer and voltage divider into the unsigned range of the ADC, a split power supply with +2.5 V and –2.5 V is used. The common ground of the two power supplies has a voltage of one-half of the voltage SVcc. This voltage is used as a base for the ADC voltages. The MSP430 measures this base voltage at regular intervals and subtracts it from every measured current or voltage sample. In this way, signed measurement is possible.

To have a reference for the measurements a reference diode LM385–2.5 is used. The voltage of this diode is measured in regular intervals and the measured value is used as a base for the SVcc relative ADC measurements.

4.1.5.1 Current Measurement With a Shunt

The solution which uses a shunt resistor for the measurement of the load current is shown in Figure 4–15. The load current I_{load} flows through the shunt, which has a resistance of approximately 1.0 mΩ. The voltage drop at the shunt is amplified and measured by the MSP430. The output voltage V_{out} seen at the ADC of the MSP430 is like described in Section 4.1.4.5.1.

If needed, additional current ranges can be implemented (three analog switches of the TLC4016 are not used).

A backup battery allows the time information (provided by the basic timer) to be kept and is also used during power-down periods. All current-consuming peripherals may be switched off. Therefore; the reference diode, the range switch, and the amplifier are switched off by the SVcc output. The EEPROM is switched off with a TP–output.

A prepayment interface is connected to the MSP430. It allows the ac to be switched on after the insertion of a valid prepayment card.
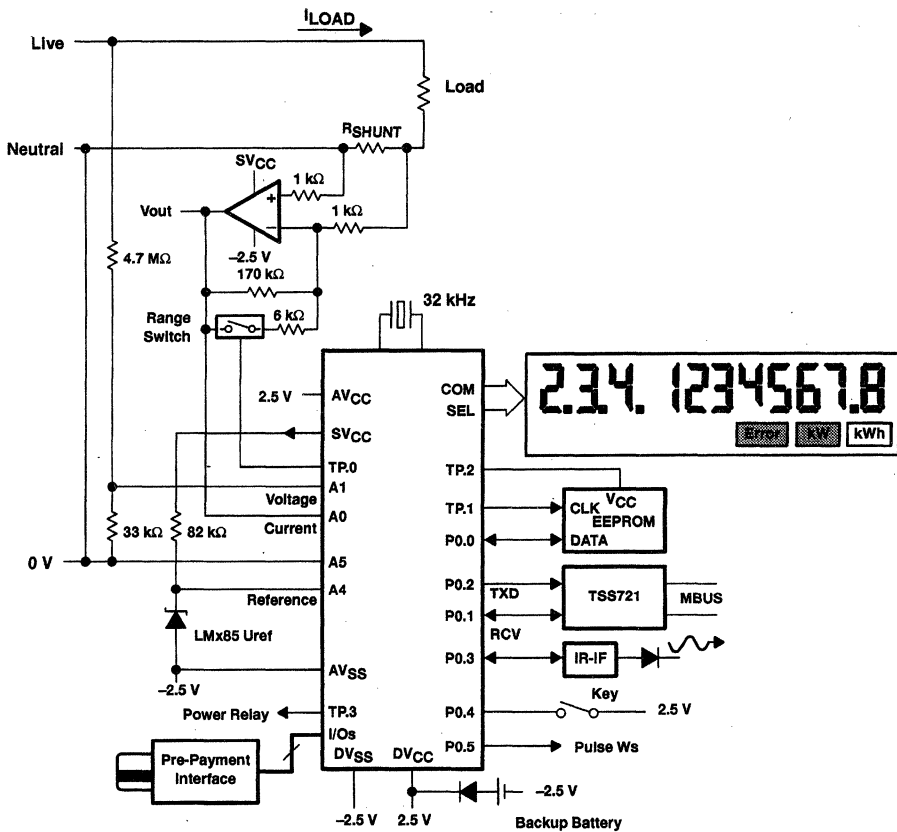
Figure 4–15. Single-Phase Electricity Meter With Shunt Resistor

4.1.5.2 Current Measurement With a Current Transformer

The solution, which uses a current transformer for the measurement of the load current, is shown in Figure 4–16. The secondary current I_{sec} of the transformer flows through two paralleled resistors and generates a voltage V_{sec} which is measured by the MSP430. For currents greater than a certain value, the resistor with the lower value is switched on by the analog switch TLC4016. For low currents, this switch is opened to get a higher voltage and, therefore, a better resolution. The range switch algorithm uses a certain hysteresis to avoid too much switching.

If needed, additional current ranges can be implemented with the three analog switches of the TLC4016 that are not used.

An AC Down signal out of the power supply connected to the interrupt I/O terminal P0.6 allows the MSP430 to save important values (i.e., energy consumption) in the EEPROM in case of a power-fail. See Section 5.7, *Battery Check and Power Fail Detection*.

The RF-readout module is connected to free outputs; this can be an unused segment line, a TP output, or an I/O pin of Port0. The timing for the RF readout is made by the internal Basic Timer. It delivers the needed interrupt frequencies. The supply voltage needed for the RF interface is done with a step-up voltage supply. It transforms the available 5 V to 6 V or more.
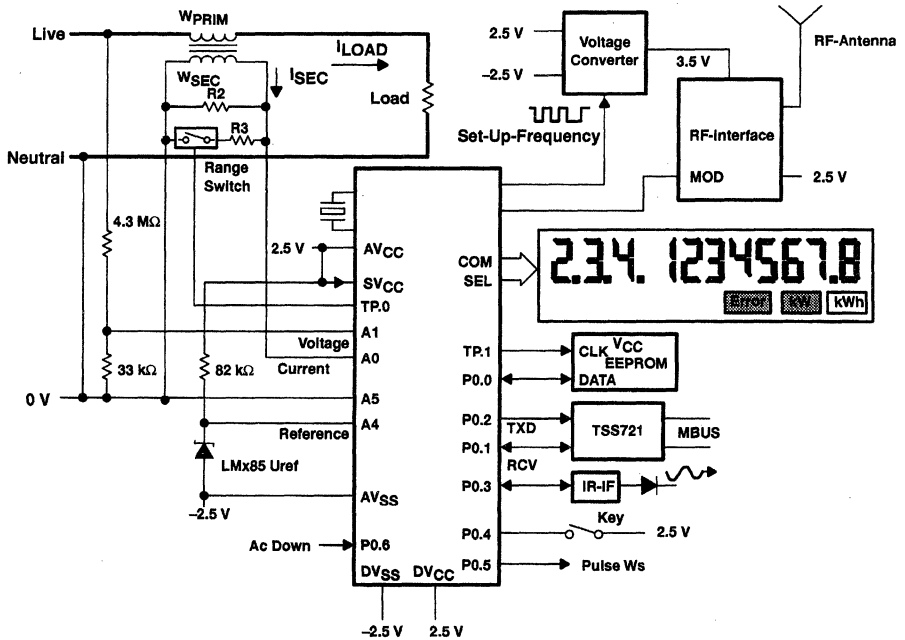
Figure 4–16. Single-Phase Electricity Meter With Current Transformer and RF Readout

4.1.5.3 Calculations

For four single-phase versions, the typical values are calculated:

❑ Version with minimum current consumption (low CPU and ADC speed)

❑ Compromise between current consumption and resolution (medium CPU speed, medium ADC speed). The basic timer is used for the time base.

❑ Compromise similar to 2, but with the use of the universal timer/port module for the time base.

❑ Version with high resolution due to sampling speed. If necessary, the ADC Clock can be up to 1.5 MHz.

Table 4–8. Typical Values for a Single-Phase Meter

ITEM	MINIMUM CONSUMPTION	COMPROMIZE 1	COMPROMIZE 2	HIGH RESOLUTION
AC Frequency	50 Hz	50 Hz	50 Hz	50 Hz
Time Base for ARR	ACLK/18	Basic Timer 2048 Hz	ACLK/9	ACLK/5
MCLK (CPU Clock)	0.754 MHz	0.754 MHz	1.048 MHz	2.195 MHz
ADC Clock (ADCLK)	0.754 MHz	0.754 MHz	1.048 MHz	1.097 MHz
N (MCLK/ACLK)	23	23	32	67
ADC Repetition Rate ARR	1820.4 Hz	2048 Hz	3640.9 Hz	6553.6 Hz
Phase Repetition Rate (ARR/2)	910.22 Hz	1024 Hz	1820.4 Hz	3276.8 Hz
Phase Repetition Time (2/ARR)	1098.63 µs	976.56 µs	549.32 µs	305.18 µs
Measurements per 360° (50 Hz)†	36.4	41.0	72.8	131.1
Sample Phase Shift α	9.88°	8.79°	4.94°	2.75°
Inherent Error‡	−1.5%	−1.2%	−0.37%	−0.11%
ADC Conversion Time tc (14 bits)	175.1 µs	175.1 µs	125.89 µs	120.25 µs
Interrupt Overhead ti 22 MCLKs§	29.2 µs	29.2 µs	10.5 µs	10.0µs
Time per Measurement tc + ti	204.3 µs	204.3 µs	136.4 µs	130.3 µs
Time between interrupts 1/ARR	549.3 µs	488.3 µs	274.7 µs	152.6 µs
ADC Loading (tc + ti) x ARR	37.2%	41.8%	49.7%	85.4%
CPU Loading by MPYs¶	19.3%	21.8%	27.6%	23.8%
Approx. Icc (nominal) for MSP430C323	820 µA	820 µA	1035 µA	1872 µA

† ADC conversions per complete mains period (voltage and current samples)
‡ The Inherent Error—a constant value—is compensated with the calibration values
§ Time from ADC interrupt acknowledge until next conversion is started (after 22 MCLKs)
¶ One signed multiplication per phase repetition time; 160 cycles for each one

4.1.6 Dual-Phase Electricity Meters

The measurement sequence for a dual-phase electricity meter is shown in Figure 4–17.
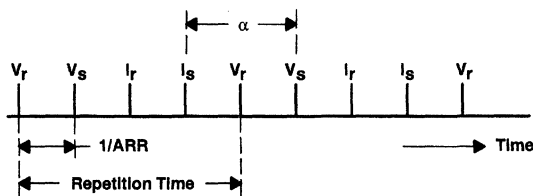
Figure 4–17. Timing for the Reduced Scan Principle (Dual-Phase Meter)

Where:

Repetition Time	1/Phase Repetition Rate.
	Length of a Complete Measurement Cycle
1/ARR	Repetition Rate of the ADC
α	Inherent Phase Shift of the Measurement Method
Vx	Voltage sample Phase x
Ix	Current sample Phase x

Two electronic electricity meters are shown, designed for the measurement of US domestic ac. As power connections, two phases and a neutral line are led into the house. This enables the use of two voltages: 120 V and 240 V.

To measure the electric energy used, two current transformers are necessary. The voltage of each phase is measured directly. With this configuration, the energy consumption of any load connection can be measured exactly. Loads from any phase to neutral (120 V) are measured as well as loads connected between the two phases (240 V).

4.1.6.1 Current Measurement With Current Transformers and Virtual Ground IC

A solution which uses two current transformers for the measurement of the load currents is shown in Figure 4–18. The secondary current I_{sec} of the transformer flows through two parallel resistors and generates a voltage V_{sec}, which is measured by the MSP430. For currents greater than a certain value, the resistor with the lower value is switched on by the analog switch TLC4016I. For low currents this switch is opened to get a higher voltage and, therefore, a better resolution. The range switch algorithm used has a certain hysteresis to avoid too much switching.

The virtual ground IC delivers a voltage exactly in the middle between SVcc and AVss. All measurements refer to'this potential. The virtual ground voltage

itself is measured with the analog input A5 and the measured value is subtracted from each voltage and current sample.

If needed, additional current ranges can be implemented with the two analog switches of the TLC4016 that are not used.

A backup battery allows the time information (provided by the basic timer) to be kept during power-down periods. All current-consuming peripherals can be switched off; the reference diode, the range switches, the virtual ground with the SVcc output, and the EEPROM with a TP output.
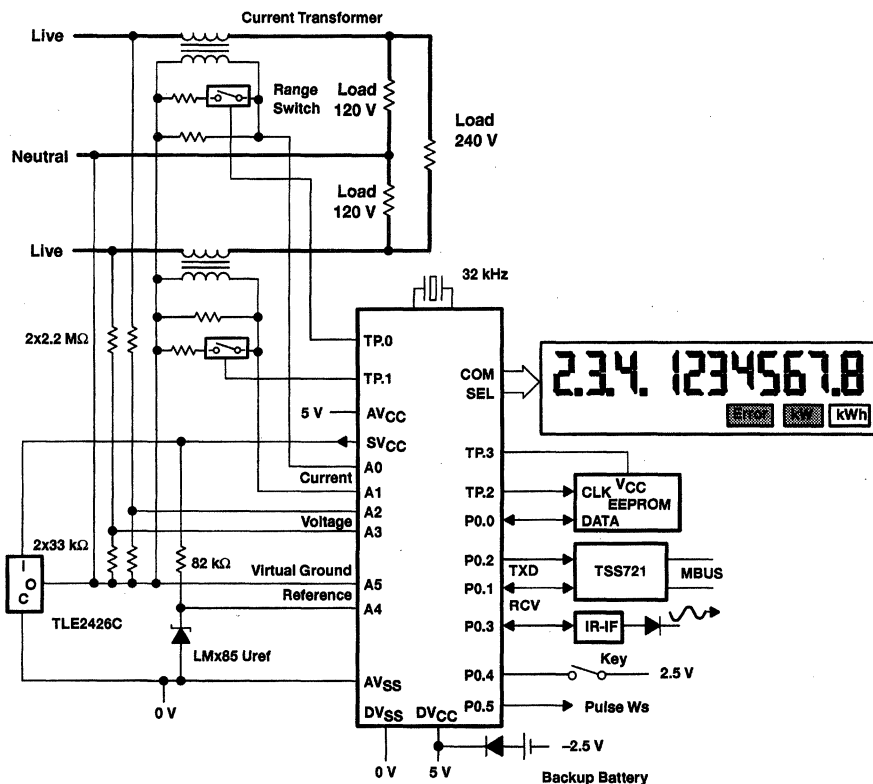
Figure 4–18. Dual-Phase Electricity Meter With Current Transformers and Virtual Ground

4.1.6.2 Current Measurement With Current Transformers and Software Offset

Figure 4–19 shows a two-phase electricity meter that uses voltage dividers to get reference voltages in the middle of the supply voltage for current and voltage inputs. The resistors of this voltage dividers are chosen to be smaller than the maximum source impedance of the ADC (see Section 4.1.4.2, *ADC Input Considerations*). To get the ADC value of the virtual midpoint of the ADC range, the software offset method is used (see Section 4.1.4.4.3, *Resister Interface (Software Offset)*). This value is subtracted from each voltage and current sample to get signed, offset-corrected results.

No backup battery is provided. This means, that in regular time intervals, the actual amount of the energy consumption needs to be stored in the EEPROM. If the power supply used provides an ac down, this storage is only needed when this signal is activated.

An ac down signal from the power supply connected to the interrupt I/O terminal P0.6 allows the MSP430 to save important values (i.e., energy consumption) in the EEPROM in case of a power failure (see Section 5.7, *Battery Check and Power Fail Detection*).
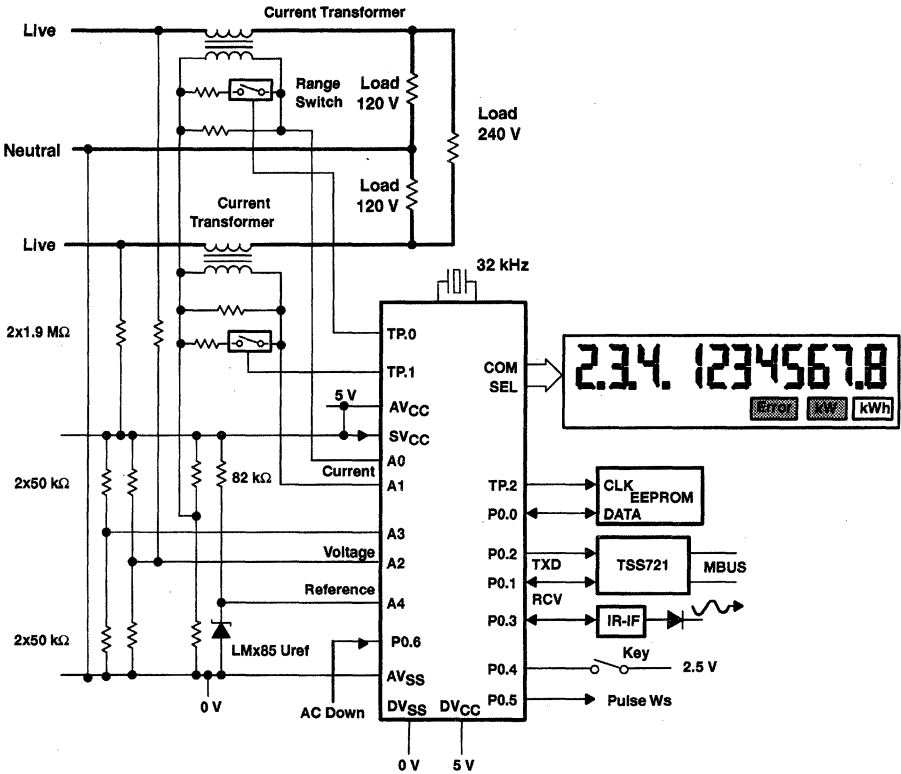
Figure 4–19. Dual-Phase Electricity Meter With Current Transformers and Software Offset

4.1.6.3 Calculations

For three dual-phase versions, the typical values are calculated:

❑ Version with minimum current consumption

❑ Compromise between current consumption and resolution

❑ Version with high resolution due to sampling speed. If necessary, the ADC Clock can be set up to 1.5 MHz (see Table 4–10).

Table 4–9. Typical Values for a Dual-Phase Meter

ITEM	MINIMUM CONSUMPTION	COMPROMISE	HIGH RESOLUTION
AC Frequency	60 Hz	60 Hz	60 Hz
Time Base for ARR	ACLK/9	ACLK/7	ACLK/5
MCLK (CPU Clock)	0.754 MHz	1.048 MHz	2.195 MHz
ADC Clock (ADCLK)	0.754 MHz	1.048 MHz	1.097 MHz
N (MCLK/ACLK)	23	32	67
ADC Repetition Rate ARR	3640.9 Hz	4681.1 Hz	6553.6 Hz
Phase Repetition Rate (ARR/4)	910.22 Hz	1170.3 Hz	1638.4 Hz
Phase Repetition Time (4/ARR)	1098.63 µs	854.5 µs	610.35 µs
Measurements per 360° (60 Hz)†	30.34	39.0	54.6
Sample Phase Shift α	11.86°	9.22°	6.59°
Inherent Error‡	−2.10%	−1.29%	−0.66%
ADC Conversion Time tc (14 bits)	175.1 µs	125.9 µs	120.3 µs
Interrupt Overhead ti§	29.2 µs	21.0 µs	10.0 µs
Time per Measurement tc + ti	204.3 µs	146.9 µs	130.3 µs
Time between interrupts 1/ARR	274.7 µs	213.6 µs	152.6 µs
ADC Loading (tc + ti)xARR	74.4%	68.8%	85.4%
CPU Loading by MPYs¶	38.6%	35.7%	23.8%
Approx. Icc (typical) for MSP430	820 µA	1035 µA	1872 µA

† ADC conversions per complete ac cycle and phase (voltage and current samples)
‡ The Inherent Error, a constant value, is compensated with the calibration values
§ Time from ADC interrupt acknowledge until next conversion is started (after 22 MCLKs)
¶ Two signed multiplications per phase repetition time; 160 cycles for each one

4.1.7 Three-Phase Electricity Meters

Two electronic electricity meters are discussed and designed for the measurement of European domestic ac. As power connections, three phases and a neutral connection are led into the house. This enables the use of two voltages: 230 V (phase to neutral) and 400 V (phase to phase).

To measure the electric energy used, three current transformers or ferrite cores are necessary. The voltage of each phase is measured directly. With this configuration, the energy consumption of any load connection can be measured exactly. Loads from any phase to neutral (230 V) are measured as well as loads connected between the phases (400 V).

The measurement sequence is shown in Figure 4–20.
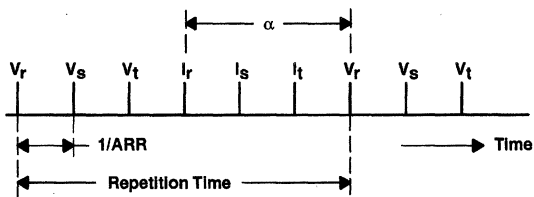
Figure 4–20. Normal Timing for the Reduced Scan Principle (Three-Phase Meters)

Where:

Repetition Time	1/Phase Repetition Rate.
	Length of a Complete Measurement Cycle
1/ARR	Repetition Rate of the ADC
α	Inherent Phase Shift of the Measurement Method

If a more evenly spaced sequence is desired (i.e., for better distribution of the multiplications), the following sequence can be used. Current and voltage samples are made alternating.
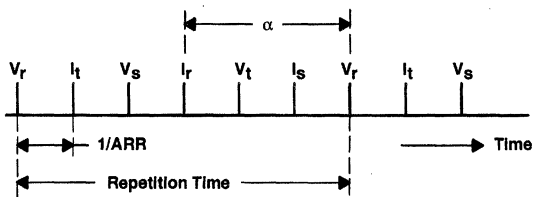
Figure 4–21. Evenly Spaced Timing for the Reduced Scan Principle (Three-Phase Meters)

4.1.7.1 Current Measurement With Ferrite Cores and Software Offset

Figure 4–22 shows a three-phase electricity meter that uses voltage dividers to get reference voltages in the middle of the supply voltage for each voltage input. The resistors of these voltage dividers are chosen to be smaller than the maximum source impedance of the ADC (see Section 4.1.4.2, *ADC Input Considerations*). To get the ADC value of the virtual middle of the ADC range, the software offset method is used. This value is subtracted from each voltage and current sample to get signed, offset-corrected results. The range is selected by different amplifications of the coil preamplifier.

The reference is provided by the stable +5 V supply.

If needed and with more TLC4016 ICs, additional current ranges can be implemented.

A backup battery allows the time information (provided by the basic timer) to be kept during power-down periods. All current-consuming peripherals can be switched off; including, the resistor dividers, the range switches, the amplifiers (integrators) by the SVcc output, and the EEPROM with a TP output.
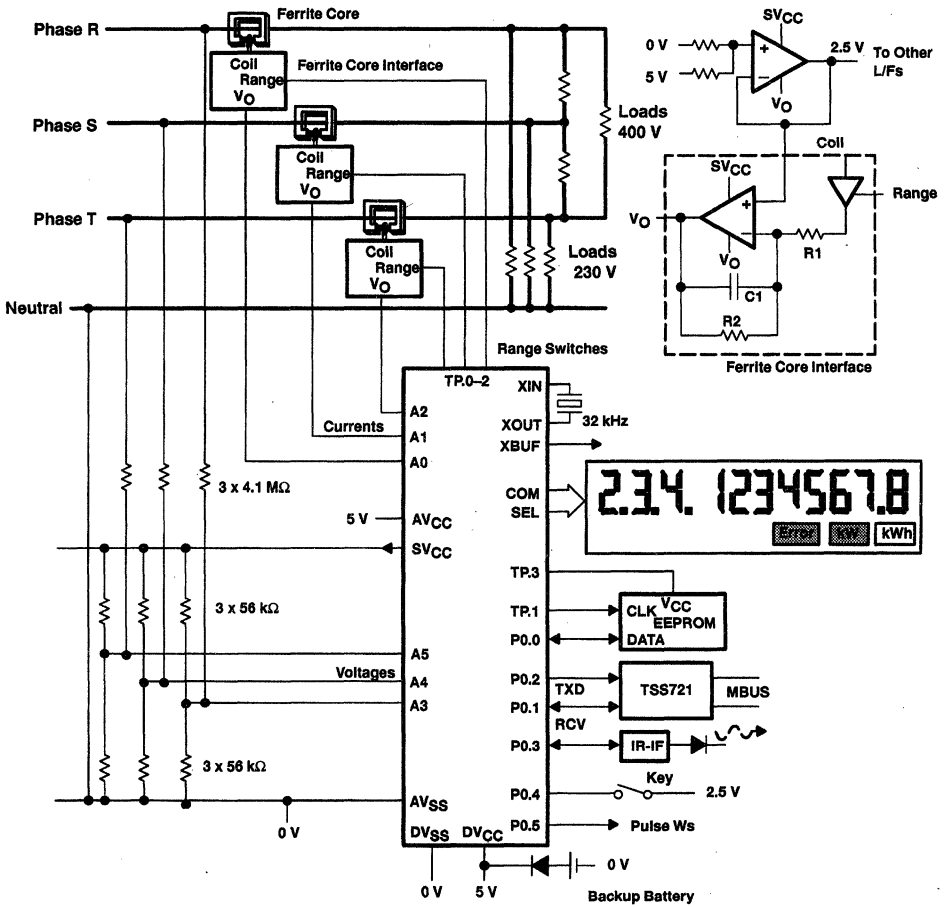
Figure 4–22. Three-Phase Electricity Meter With Ferrite Cores and Software Offset

4.1.7.2 Current Measurement With Current Transformers and Split Power Supplies

The six analog inputs of the MSP430C32x only allow the measurement of the three currents and three voltages without external circuitry. If a reference diode is needed (i.e., because the power supply cannot be used as a reference) or one of the methods using a ground that needs to be measured is used, then an analog multiplexer, like the TLC4016, is needed (see Figure 4–23). With its three outputs (TP.3 to TP.5) the MSP430 selects the phase to be measured.

No backup battery is provided, this means that in regular time intervals, the actual amount of energy consumption needs to be stored in the EEPROM. If the power supply used provides an ac Down, this storage would only necessary when this signal was activated.

The same circuitry can be used with a virtual ground IC. Only a few modifications are necessary (see Figure 4–18).

An ac down signal from the power supply connected to the interrupt I/O terminal P0.6 allows the MSP430 to save important values (i.e., energy consumption) in the EEPROM in case of a power failure *(see Section 5.7, Battery Check and Power Fail Detection).*
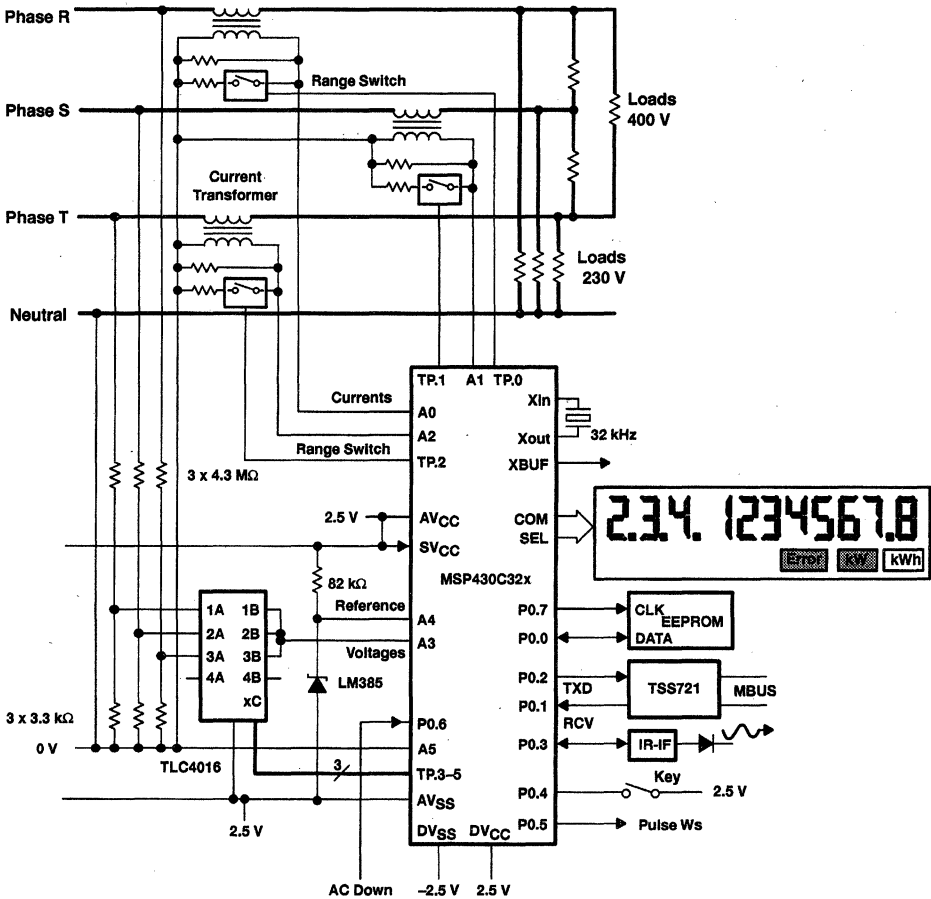
Figure 4–23. Electricity Meter With Current Transformers and Split Power Supply

4.1.7.3 Calculations

For four three-phase electricity meters, the typical values are calculated:

❏ Version with minimum current consumption

❏ Compromise between current consumption and resolution

❏ Version with high resolution. This version can be used with an MCLK frequency of 3.3 MHz, when a maximum calculation performance is needed.

❏ Version with the highest resolution due to the maximum sampling speed

Table 4–10. Typical Values for a Three-Phase Meter

ITEM	MINIMUM SUPPLY CURRENT	COMPROMISE	HIGH RESOLUTION	HIGHEST RESOLUTION
AC Frequency	50 Hz	50 Hz	50 Hz	50 Hz
Time Base for ARR	Basic Timer 4096 Hz	ACLK/6	ACLK/5	Basic Timer 8192 Hz
MCLK (CPU Clock)	1.048 MHz	2.097 MHz	2.195 MHz	2.949 MHz
ADC Clock (ADCLK)	1.048 MHz	1.048 MHz	1.097 MHz	1.475 MHz
N (MCLK/ACLK)	32	64	67	90
ADC Repetition Rate ARR	4096Hz	5461.3 Hz	6553.6 Hz	8192 Hz
Phase Repetition Rate (ARR/6)	682.67 Hz	910.22 Hz	1092.3 Hz	1365.33 Hz
Phase Repetition Time (6/ARR)	1464.8 μs	1098.63 μs	915.53 μs	723.4 μs
Measurements per 360° (50 Hz)†	27.3	36.4	43.7	54.6
Sample Phase Shift α	13.19°	9.88°	8.24°	6.59°
Inherent Error‡	–2.6%	–1.5%	–1.03%	–0.66%
ADC Conversion Time tc (14 bits)	125.9 μs	125.9 μs	120.3 μs	89.5 μs
Interrupt Overhead ti§	21.0 μs	10.5 μs	10.0 μs	7.5 μs
Time per Measurement tc + ti	146.9 μs	136.4 μs	130.3 μs	97.0 μs
Time between interrupts 1/ARR	244.1 μs	183.1 μs	152.6 μs	122.1 μs
ADC Loading	60.2%	74.5%	85.4%	73.3%
CPU Loading by MPYs¶	31.3%	20.8%	23.8%	20.8%
Approx. Icc (typical) for MSP430	1035 μA	1800 μA	1872 μA	2423 μA

† ADC conversions per complete ac cycle and phase (voltage and current samples)
‡ The Inherent Error, a constant value, is compensated with the calibration values
§ Time from ADC interrupt acknowledge until next conversion is started (after 22 MCLKs)
¶ Three signed multiplications per phase repetition time; 160 cycles for each one

4.1.7.4 Timing and Software

The timing in Figure 4–24 is shown for the compromise solution in Section 4.1.7.3, *Calculations,* (see Figure 4–22). The interrupt of the universal timer/port module (UT/PM) reads out the actual ADC result, prepares and starts the next measurement. The ADC interrupt is not used.

The instruction timing is shown in CPU cycles (MCLK = 2.097 MHz). The ADC timing uses an ADCLK = 1.048 MHz (MCLK/2).

Register R5 is used exclusively by the interrupt handler (status word) to reduce the execution time of the interrupt handler.

Figure 4–24 shows the timing without latencies due to other interrupts. If these latencies are included, the overall timing can increase by several cycles. This

makes strict real time programming necessary. For example, any interrupt service handler (except the UT/PM handler) must have the instruction EINT (Enable Interrupt) at its beginning.

Calculations show that the interrupt latency time does not influence the measurements. The statistical distribution results in an error are nearly zero (see Section 4.1.2.5, *Measurement Error in Dependence of the Interrupt Latency Time*).
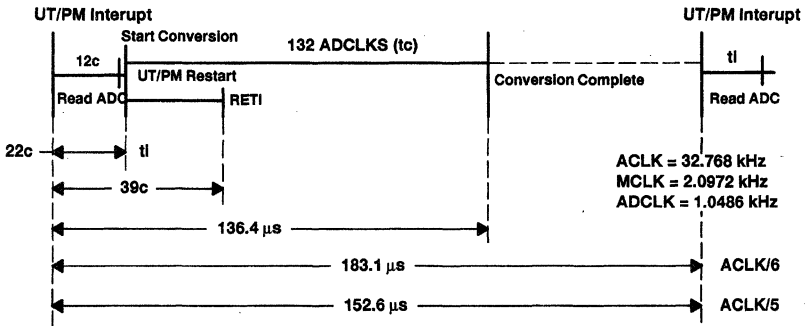
Figure 4–24. Timing for the Reduced Scan Principle

The interrupt software used is: (Register R5 is reserved for the interrupt handling to get the shortest possible time)

```
; Hardware Definitions
;
ADAT      .EQU      0118h             ; ADC 14-bit result buffer
ACTL      .EQU      0114h             ; ADC control word
M2        .EQU      02000h            ; ADC prescaling: ADCLK = MCLK/2
Rauto     .EQU      0800h             ; Automatic range selection
CSoff     .EQU      0100h             ; Current Source off
A5        .EQU      014h              ; Analog input A5: Vt
A4        .EQU      010h              ; Analog input A4: Vs
A3        .EQU      00Ch              ; Analog input A3: Vr
A2        .EQU      008h              ; Analog input A2: It
A1        .EQU      004h              ; Analog input A1: Is
A0        .EQU      000h              ; Analog input A0: Ir
soc       .EQU      001h              ; Conversion Start
```

```
TPCTL     .EQU    04Bh                          ; UT/PM control word
RC1FG     .EQU    002h                          ; UT/PM interrupt flag
TPCNT1    .EQU    04Ch                          ; UT/PM counter
;
; RAM Definitions
;
ADCTAB    .WORD   0,0,0,0,0,0                   ; Ir,Vt,Is,Vr,It,Vs storage;
;                                                              MCLK Cycles
;
;                                               ; Interrupt Latency   6
UT_HNDLR  MOV     &ADAT,ADCTAB(R5)  ; Store act. ADC result       6
          MOV     TAB(R5),PC        ; Go to individual handler    3
TAB       .WORD   Vt,Is,Vr,It,Vs,Ir ; Six meas. handlers
;
; Individual handler parts for each phase (current and voltage)
; The next sample is selected and the measurement started.
;
Vt        MOV     #M2+Rauto+CSoff+A5+CS,&ACTL        ; Select Vt     6
          JMP     UT_COM                                          ;2
Is        MOV     #M2+Rauto+CSoff+A1+CS,&ACTL        ; Select Is     6
          JMP     UT_COM                                          ;2
Vr        MOV     #M2+Rauto+CSoff+A3+CS,&ACTL        ; Select Vr     6
          JMP     UT_COM                                          ;2
It        MOV     #M2+Rauto+CSoff+A2+CS,&ACTL        ; Select It     6
          JMP     UT_COM                                          ;2
Vs        MOV     #M2+Rauto+CSoff+A4+CS,&ACTL        ; Select Vs     6
          JMP     UT_COM                                          ;2
Ir        MOV     #M2+Rauto+CSoff+A0+CS,&ACTL        ; Select Ir     6
          MOV     #-2,R5                       ; Restart sequence with Vt   2
;
; Common part: time base is subtracted from UT/P. UT/P Flag is
; reset. Next measurement is prepared
;
UT_COM    SUB.B   #6,&TPCNT1        ; ACLK/6 is time base            5
          BIC.B   #RC1FG,&TPCTL     ; Reset UT INTRPT flag           4
          ADD     #2,R5             ; To next measurement            1
```

Application Examples

```
        RETI                                  ; Return from INTRPT          5
```

Nearly the same interrupt handler can be used with sample timing defined by the basic timer. The differences are:

❏ No resetting of the interrupt flag is necessary. It resets automatically.

❏ No reloading of the timer register is necessary. The timer runs continuously.

This shortens the interrupt handler by 12 cycles.

```
;                                             ; Interrupt Latency    6
UT_HNDLR MOV    &ADAT,ADCTAB(R5) ; Store actual ADC result             6
         ADD    #2,R5                     ; To next measurement         1
         MOV    TAB-2(R5),PC        ; Go to individual handler          3
TAB      .WORD  Vt,Is,Vr,It,Vs,Ir ; Six measurement handlers
;
; Individual handler parts for each phase (current and voltage)
; The next sample is selected and the measurement started.
;
Vt       MOV    #M2+Rauto+CSoff+A5+CS,&ACTL ; Select Vt                 6
         RETI                             ; Return from INTRPT          5
;
Is       MOV    #M2+Rauto+CSoff+A1+CS,&ACTL ; Select Is                 6
         RETI                             ;                             5
;
Vr       MOV    #M2+Rauto+CSoff+A3+CS,&ACTL ; Select Vr                 6
         RETI                             ;                             5
;
It       MOV    #M2+Rauto+CSoff+A2+CS,&ACTL ; Select It                 6
         RETI                             ;                             5
;
Vs       MOV    #M2+Rauto+CSoff+A4+CS,&ACTL ; Select Vs                 6
         RETI                             ;                             5
;
Ir       MOV    #M2+Rauto+CSoff+A0+CS,&ACTL ; Select Ir                 6
         CLR    R5                        ; Restart sequence            1
         RETI                             ; Return from interrupt       5
```

The previous interrupt handler can also be adapted to the electricity meter shown in Figure 4–23. The input selection with the TP outputs must be in-

cluded into the parts serving the three ac voltages. The selected ADC input is A3 for all voltage measurements.

4.1.8 Measurement of Voltage, Current, Apparent Power, and Reactive Power

The reduced scan principle measures only active power. If reactive power or apparent power is to be measured, other methods have to be used. The implemented measurement method also depends on the main application of the electricity meter. A meter for the measurement of reactive power only probably uses a different algorithm than an electricity meter for active power that does the reactive power measurement as a background task only. This section shows simple methods that use as much as possible the voltage and current samples measured for the active power calculation.

4.1.8.1 Measurement of Voltage and Current

The measurement of voltage and current is possible by summing up the absolute values of the ADC results during integer numbers of full periods. The result is an indication of the average value of the voltage Vavrg respective of the current Iavrg. If corrected as shown, the current and voltage values can be used for other purposes. The formula for a sinusoidal voltage is shown in the following. The one for the current is equivalent to it.

$$V_{avrg} = \frac{V_{peak} \times 2}{p} \; ; \; V_{eff} = \frac{V_{peak}}{\sqrt{2}} \rightarrow V_{eff} = \frac{V_{avrg} \times p}{2\sqrt{2}} \approx 1.11 \times V_{avrg}$$

4.1.8.2 Measurement of the Apparent Power

The apparent power is defined by the formula Papp = U × I. There is no exact definition for the apparent power when harmonics are included. A possible solution is to use the voltage and current samples (see Section 4.1.8.1, *Measurement of Voltage and Current*) taken for the active power measurement. These samples are made absolute and summed up for an integer number of ac periods. If these summed-up values, representing the average value, are multiplied and corrected the apparent power is the result.

The correction is necessary due to the difference of the average value and the effective value of a sinusoidal current or voltage (see Section 4.1.8.1). The apparent energy Wapp is:

$$W_{app} = V_{avrg} \times I_{avrg} \times 1.11^2 \times t$$

4.1.8.3 Measurement of the Reactive Power

Two simple methods exist for the measurement of the reactive power:

❏ Delay of the voltage (or current) samples for the time representing 90° ($\pi/2$) of the ac frequency.

❏ Calculation of the apparent power and the active power

Delay of Samples

With a carefully chosen sampling frequency, the angle 90° ($\pi/2$) can be made an integer multiple of the sampling interval. If each voltage sample is delayed with a RAM-based by this integer number and multiplied with the actual current sample, the result is the reactive power.

EXAMPLE: ac frequency 50 Hz, 90° are 5 ms, with a sampling frequency of 2000 Hz the necessary FIFO buffer is 5 ms x 2000 Hz = 10 words. For every phase 20 bytes of RAM are needed for the FIFO.

Calculation out of the Apparent Power

The apparent power is calculated as described in Section 4.1.8.2. The reactive power is calculated with the values of the active power and the apparent power by the formula:

$$W_{react} = \sqrt{W_{app}^2 - W_{act}^2}$$

Note:

All the calculations described previously can be made with the MSP430 floating-point package. It is available with two lengths of mantissa; 24 bits and 40 bits (see Section 5.6, *The Floating Point Package*).

4.1.9 Calculation of the System Current Consumption

The base of the following current consumption table is derived from the following data sheet information:

Table 4-11. Current Consumption of the System Components

Device	I_{CC}	f_{osc}	V_{CC}	T_A
MSP430C32x (ADC on)†	1000 µA	1 MHz	5 V	–40°C to 85°C
TLC4016	20 µA	N/A	5 V	25°C
TLE2426	170 µA	N/A	5 V	25°C
TSS721‡	18 mA max.	N/A	5 V	–40°C to 85°C
EEPROM 24AA01§	100 µA max.	N/A	5 V	0°C to 70°C
OPAMP TLC1079	40 µA	N/A	5 V	25°C
LM385–2.5¶	30 µA	N/A	5 V	
LCD 20mm×100mm#	26 µA	N/A	5 V	
Crystal‖	2 µA	32.768 kHz	N/A	

† The supply current of the MSP430 (excluding the ADC) depends on the MCLK in a linear manner. The supply current of the ADC depends mainly on the current flowing through the internal resistor divider and is, therefore, treated as constant.
‡ Power for the TSS721 is taken from the M-BUS, so the electricity meter's supply is not used.
§ Standby current of the Microchip EEPROM. Read and write currents are 1 mA and 3 mA. The EEPROM can be switched off completely during the standby periods.
¶ The reference diode can be switched off when not used for the reference measurements.
# A typical current value of 13 nA/mm^2 is used.
‖ A typical driver power of 10 µW is assumed.

With the previous data, the system consumption is calculated under the following conditions:

❏ The compromise solutions shown in Sections 4.1.5, 4.1.6, and 4.1.7 are used for the six hardware proposals

❏ Nominal current consumption is assumed for all system components

❏ The ac voltage has its nominal value

❏ The ac load current is assumed to be zero. This eliminates the influence of different current interfaces.

Table 4–12. System Current Consumption for Six Proposals

Device	Single Phase Shunt	Single Phase Current Transf.	Dual Phase Virtual Ground IC	Dual Phase Software Offset	Three Phase Ferrite Core	Three Phase Current Transf.
MSP430C32x	820 µA	820 µA	1035 µA	1035 µA	1800 µA	1800 µA
TLC4016	20 µA	20 µA	20 µA	20 µA	20 µA	40 µA
TLE2426	–	–	170 µA	–	–	–
TSS721	–	–	–	–	–	–
EEPROM	100 µA	100 µA	100 µA	100 µA	100 µA	100 µA
OPAMPs	40 µA	–	–	–	40 µA	–
LM385–2.5	30 µA	30 µA	30 µA	30 µA	30 µA	30 µA
LCD / Crystal	28 µA	28 µA	28 µA	28 µA	28 µA	28 µA
Resistor Dividers	–	–	–	134 µA	134 µA	–
System Current	1038 µA	998 µA	1383 µA	1347 µA	2152 µA	1998 µA
Voltage Path	12 mW	12 mW	13 mW	15 mW	44 mW	37 mW

The value given for the voltage path shows the power needed for the voltage dividers adapting the ac voltage to the analog inputs. All phases of a system are included as well as dc and ac energy.

4.1.10 System Components

The complete electricity meter system consists of the following parts. The system components not described until now are explained in the following:

☐ The microcomputer MSP430C32x with its 14-bit ADC
☐ The LCD with up to 10.5 digits (4 MUX)
☐ The EEPROM with 128 bytes (256 bytes)
☐ The current interface and the current range switches
☐ The power supply including the ADC offset generation
☐ The M-Bus interface (TSS721)
☐ The infrared interface
☐ The reference diode (LM385–2.5)
☐ Other peripherals

The last four components are not necessary in all applications, they can be omitted when not necessary.

4.1.10.1 The Microcomputer MSP430

The MSP430 is described in detail in Chapter 1.

4.1.10.2 The LCD

Any customized LCD can be connected to the MSP430, as long as it meets the electrical specifications (i.e., maximum capacitance per segment and common lines). Every segment of the LCD can be controlled independently of the others. This means 256 (or 512) (3 MUX) possible combinations. SL means number of segment lines.

The number of digits dependent on the multiplexing scheme:

$$
\begin{array}{ll}
4 \text{ MUX} & \text{digits} = \text{SL}/2 \\
3 \text{ MUX} & \text{digits} = \text{SL}/3 \\
2 \text{ MUX} & \text{digits} = \text{SL}/4 \\
1 \text{ MUX} & \text{digits} = \text{SL}/8
\end{array}
$$

The unused segments H (decimal points) of the digits can be used for the display of complete words (kWh, Ws, Low Tariff, etc.). This is used within the figures showing the electricity meter proposals.

4.1.10.3 The EEPROM

The EEPROM contains data that must not be lost during power down cycles.

❑ Calibration data (slopes and offsets for every range and phase)
❑ Meter number and other device related numbers
❑ Summed-up energy (stored in regular intervals (e.g., every 12 hours))
❑ Other data (e.g., statistical data)
❑ Error characteristics (current transformer, ADC, etc.).

For the summed-up energy, a kind of circular buffer can be used, which avoids the use of the same cells for every update. No pointer should be used for this purpose. A simple check for the lowest stored energy value determines the next storage location. This check can be made during the power-up sequence and the result is stored in the RAM for later use. This pointer is updated after each write cycle to the EEPROM. A checksum or an CRC (cyclic redundancy check) can be used for the safety of the stored data.

The tables containing the error characteristics of the current transformer and the ADC can be used for correction purposes if needed.

Dependent on the amount of data to be stored, an EEPROM with 128 bytes or 256 bytes is required. The EEPROM is driven with a software handler. Clock and data lines are set and reset by software (also see Section 3.2, *Storage of Calibration Constants*, and Section 3.4, *I2C Bus Connections*). The EEPROM can be switched off by an output, if it is not in use to save current.

4.1.10.4 The Range Switches

The resolution and accuracy of an electricity meter can be increased if more than one current range is used. The analog switch TLC4016 with its four channels is suited very well for this purpose. The MSP430 decides independently, for any phase, which current range is to be used. The ranges should be overlapping and the design should use a SCHMITT-trigger characteristic for the change in ranges. This is done to avoid too many changes.

4.1.10.5 Power Supplies

The stability of the power supply used decides if a reference voltage (see Section 4.1.10.8) is necessary or not. If the power supply is stable enough, it can be used as the reference also. This simplifies the system in three ways:

☐ No reference measurements are necessary in regular time intervals (e.g., every second) that makes the omission of one sample necessary.

☐ No correction calculations are needed to correct the summed-up energy values.

☐ No reference diode and additional hardware is necessary. The analog input can be used for other purposes.

The stability of the power supply should be better than a factor of 4 of the desired accuracy of the electricity meter. This is due to the quadratic influence of SVcc. See Section 4.1.3.3. More information is given in the Section 3.8, *Power supplies for the MSP430.*.

4.1.10.6 The M-Bus Interface TSS721 (Option)

The M-Bus interface allows the connection of the electricity meter to networks. The M-Bus interface uses the on-chip UART or one input and one output with a software driven protocol.

Applications of the M-Bus interface:

☐ Calibration: connection to the calibration hardware

☐ Automatic readout by a host: the actual consumption and other interesting values can be read out with a customer defined protocol.

☐ Tariff switching: the host defines the actual tariff by sending the appropriate information

☐ Test: start of ROM-based testing routines or down-loading and starting of RAM-based test routines

Instead of the M-Bus, any other bus can be used with the MSP430.

4.1.10.7 The Infrared Interface

The infrared interface allows bidirectional data transfer for calibration, test, and readout. One of the P0-ports can be used with its interrupt capability for bidirectional transfers.

4.1.10.8 The Voltage Reference

To have a reference for the measurements, a reference diode LM385–2.5 can be used. The voltage of this diode is measured in regular intervals and the measured value is used as a base for the SVcc relative ADC measurements. To reduce the supply current, the LM385 can be switched on only during the reference measurements.(see Figure 4–15).

No reference diode is necessary if a 5-V voltage regulator (or two 2.5-V regulators) is used with the necessary accuracy and long term stability (see Section 4.1.10.5, *Power Supplies*).

The stability of the reference should be better than a factor of 4 as the desired accuracy of the electricity meter.

4.1.10.9 Peripherals

Some options show how to interface the MSP430 to other devices.

❏ Pulse Output: this output changes its state when a certain energy amount is consumed and is usable during calibration or accuracy checks. Mechanical displays can use this pulse output.

❏ Key Interface: keys can be interfaced very simply to the inputs of the MSP430. Interrupt is possible with all Port0 inputs.

❏ LEDs: currents up to 1.5 mA @ 0.4 V voltage drop can be driven without external buffers..

❏ Relays: driving is possible with a simple npn transistor and two resistors.

The crystal buffer output XBUF provides four different, software selectable frequencies that may be used for the peripherals. These frequencies are: MCLK, ACLK (32.768kHz), ACLK/2 (16.384kHz), ACLK/4 (8.192kHz).

4.1.10.10 Summary

As this chapter shows it is possible to build cost-effective, domestic electricity meters based on the ultra-low-power mixed-signal RISC-processor MSP430.

The 14-bit ADC and the reduced scan principle eliminate the need for a special front end. All necessary system functions are realized with the on-chip peripherals of the MSP430C32x. With appropriate calibration methods, the deviations of the ADC can nearly be eliminated. This allows electricity meters to be built for classes 2, 1 and 0.5 ranging from single-phase meters to three-phase meters.

A customer developed a single-phase electricity meter with the following properties:

☐ Class 0.5 meter for the range 0.5 A to 130 A
☐ Error within 0.2% from 300 mA to 40 A
☐ Use of the full 14-bit ADC range for current and voltage
☐ Reduced scan principle is used for the energy calculation (inclusive correction formula)
☐ Use of a virtual ground and use of the measurement result for the offset correction
☐ Current transformer is used for current measurement (a low cost version is planned with a shunt)
☐ Single range only for current measurement (no range switches)
☐ Meaningful current measurements down to 25 mA

4.1.11 Electricity Meter With an External ADC

Modern three-phase electricity meters of the upper classes must provide an enormous calculation power. Additional to the multiplications of the current and voltage samples for the active power measurement, it is necessary to calculate additional, very computation intensive values:

☐ Apparent power for all phases
☐ Sum of the capacitive reactive power
☐ Sum of the inductive reactive power
☐ Calculation of the cosφ for every phase

These calculations need to be done in real time, a microcomputer with high throughput like the MSP430C33x is necessary.

With an external ADC three different scan principles can be used:

☐ The reduced scan principle that was used with all electricity meters shown in the previous sections. Only one ADC is necessary for current and voltage with this method.

☐ The alternating scan principle—a TID invention—that also needs only a single ADC. Due to pending patent reasons, it cannot be explained further. The hardware is identical to the hardware for the reduced scan principle.

❏ The typical way with two ADCs: one for the voltage path(s) and one for the current path(s).

Figure 4–25 shows a three-phase electricity meter with a 16-bit ADC. This single ADC allows the application of the reduced scan principle and the alternating scan principle. With this hardware in use, class 0.2 is reachable.
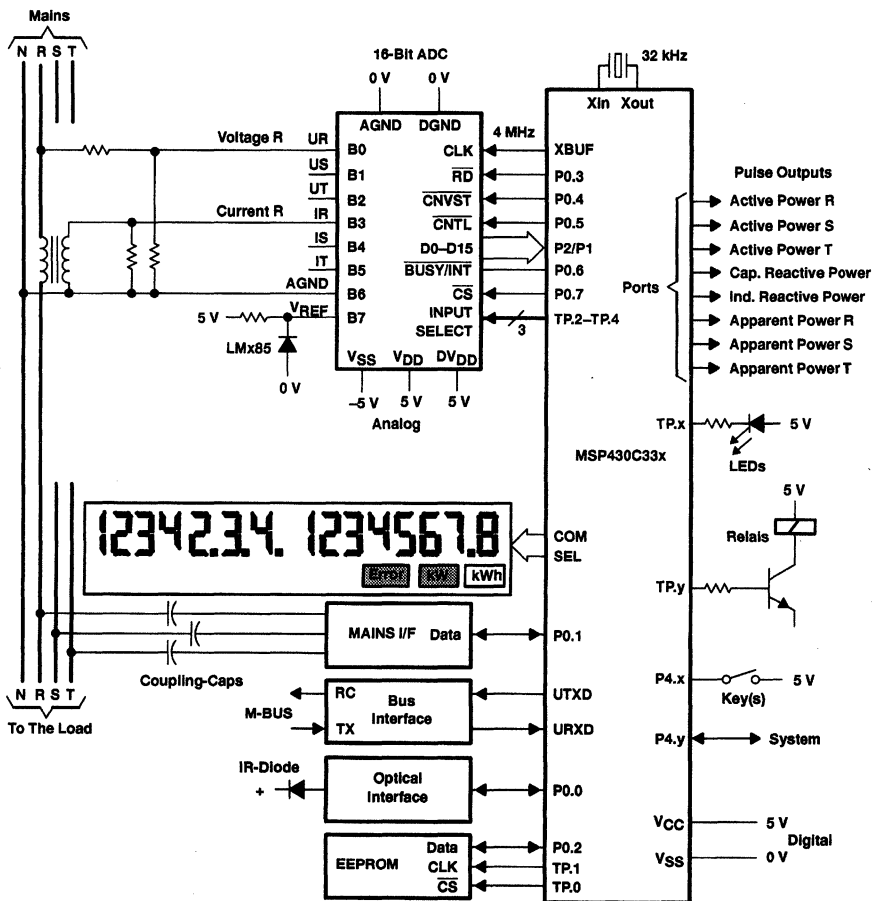
Figure 4–25. Electricity Meter With an External 16-Bit ADC

4.1.12 Error Simulation for an MSP430C32x-Based Electricity Meter

The simulation methods that lead to the results shown in this chapter are explained in detail.

4.1.12.1 Abstract

The way the calculation of the error of a simulated electricity meter built with the MSP430C32x family is shown in detail. A single-phase is simulated; this can be the only phase or one phase of a poly-phase meter. The error simulator (ES) simulates nearly exact an MSP430C32x working as an electronic electricity meter. All influences due to the MSP430 hardware are taken into account.

❏ The error due to the characteristic of the ADC

❏ The error due to the interrupt latency of the MSP430 interrupt system

❏ The error due to the range transition for samples at the boundaries of the four ADC ranges

❏ The error due to the used reduced scan principle for the measurement

4.1.12.2 Common Measurement Conditions

The ES asks at the beginning for the conditions that are used for all measurement points (they are written to the listing file):

❏ Listing path name

❏ The characteristic of the ADC; the ADC errors (in steps) at the five range boundaries are defined. See Figure 4–25 for explanation.

❏ The time interval between voltage and current sample pairs (sampling interval)

❏ The nominal ac frequency

❏ The maximum interrupt latency time; the worst-case value for the time interval from an interrupt request to the actual start of the interrupt handler.

❏ The correct ADC value for the external reference voltage 0 V

❏ The measurement time for each measurement

4.1.12.3 Calibration

The next step is the calibration for the simulated system. Two calibrations are necessary for the complete current range (the ranges shown in the following can be changed if needed):

❑ One for the current range from 0% to 5% of the maximum current. The calibration points are 0.5% and 5% of the maximum current.

❑ One for the current range from 5% to 111% of the maximum current. The calibration points are 5% and 100% of the maximum current.

The calculated energy for the low and the high calibration points are summed-up for five seconds each. The conditions are:

❑ Voltage = 100%   Nominal ac voltage

❑ Cosφ = 1   Resistive load

❑ Frequency deviation = 0   Nominal ac frequency

❑ Third harmonic in current = 0 No distortion, pure sine

❑ Interrupt latency time = 5 µs No interrupt activity except from the basic timer

❑ Measurement time for each calibration point = 5 s

The calculation for the calibration part is made exactly the same way as described in Section 4.1.4 for the error calculations.

The slope and the offset for the correction formulas are calculated with the errors calculated for these two calibration points when compared to the correct energy values. The correct energy W is calculated with the formula:

$$W = U \times I \times t \times \cos\varphi$$

4.1.12.4 Conditions for the Load Curve

Next the ES asks for the special conditions used for a load curve:

❑ The start current, the delta current, and the maximum current as a percentage of the maximum current

❑ The voltage value as a percentage of the nominal voltage

❑ The phase angle between voltage and current in degrees

❑ The frequency deviation from the nominal ac frequency

❑ The percentage of the 3rd harmonic overlaid to the current path

❑ The running time for each measurement

With the input of the values, the load curve is calculated and the results are written to the listing output.

4.1.12.5 Energy Calculation

The ES now calculates the error for the measurement points the same way the MSP430 hardware does it:

❏ The ES calculates the real ADC value for the given 0 V reference and truncates it. This is the ADC value including the ADC error

Calculation of the Voltage Sample

The voltage sample used by the ADC to define the actual ADC range (bits 13 and 12 of the ADC result) is taken:

❏ The sampling time for the next voltage sample is calculated with the defined value of the sampling interval; last voltage sample time + 2 × sampling interval.

❏ To this calculated voltage sampling time, a random value ranging from zero up to the defined maximum interrupt latency time is added. This simulates the interrupt latency time of the MSP430's interrupt system.

❏ The correct value of the next voltage sample is calculated including the frequency deviation and the given voltage value. The result is a signed ADC value (steps, but with a fractional part).

❏ The 0-V value defined during the common measurement conditions is added to the voltage value. This shifts the signed voltage value into the unsigned ADC range.

❏ This voltage value is modified with the ADC error defined by the ADC characteristic and truncated afterwards to get an integer value between 0000h to 3FFFh. Overflow and underflow leads to the results 3FFFH and 0000h respectively. This value is used for the range transition check.

❏ The real ADC value of the 0-V reference (as seen by the ADC) is subtracted from the truncated voltage value. This is the signed integer voltage value used for the calculations.

The same procedure as shown above is made for a second voltage sample measured 36μs later: this second sample is used by the ADC for the 12-bit conversion. If the two samples are located in different ADC ranges—due to the fast changing input voltage—then the saturated ADC result for the range of the 1st sample is used, exactly like the MSP430 hardware does it. This treatment simulates the range transition error of the ADC. If the two samples are located in the same ADC range, then the ADC value of the second sample is used for the energy calculation.

Calculation of the Current Samples

Each voltage sample is multiplied with the sum of two current samples; the current sample, one sampling interval before it, and one sampling interval after it. Both are sampled the same way (in reality each current sample is used twice, only one measurement is made).

The current sample used by the ADC to define the actual ADC range is taken:

☐ The sampling time for the next current sample is calculated with the defined value of the sampling interval; actual voltage sample time + sampling interval.

☐ To this calculated current sampling time, a random value ranging from zero up to the defined maximum interrupt latency time is added. This simulates the interrupt latency time of the MSP430.

☐ The correct value of the next current sample is calculated including the percentage of the current, the frequency deviation, the phase angle, and the percentage of the 3rd harmonic. The result is a signed ADC value (steps, but with fractional part).

☐ The 0-V value defined during the common measurement conditions is added to the current value. This shifts the current value into the ADC range

☐ The current value is modified with the ADC error defined by the ADC characteristic and truncated to get an integer value between 0000h and 3FFFh. Overflow and underflow lead to the results 3FFFH and 0000h, respectively. This value is used for the range transition check.

☐ The real ADC value of the 0-V reference (as seen by the ADC) is subtracted from the truncated current value. This is the signed integer current value used for the calculations.

The same procedure, as previously shown, is made for a second current sample measured 36 µs later. This second sample is used by the ADC for the 12-bit conversion. The same steps are used, as previously described, for the voltage samples. This second sample is used for the energy calculation.

Calculation of the Energy Value

☐ The two calculated current samples described in Section 4.1.4.2 are added and are multiplied afterwards with the voltage sample located between them (see Section 4.1.4.1). The result is divided by two (two samples are added) and summed-up to the energy buffer containing W_{error}

☐ After the defined measurement time, the energy buffer is multiplied by the slope and corrected with the offset (both from the calibration part)

❑ Then the error calculation is made with the normal error formula:

$$e = \frac{W_{error} - W_{correct}}{W_{correct}} \times 100$$

Where:

e	Measurement error in per cent
W_{error}	Summed-up energy during simulation
$W_{correct}$	Calculated energy with the formula shown at Section 4.1.2

4.1.12.6 Explanation Figures

Figure 4–26 shows the placement of the current and voltage coming from the voltage and the current interfaces into the ADC's range. All calculations are based on a use of 90% of the ADC range for nominal (100%) values of current and voltage. This means up to 111% of the magnitude of the nominal values are still measured correctly. This allocation can be changed if necessary.

Figure 4–27 shows the ADC characteristic that produces the worst case errors. The ADC characteristic is defined at the boundaries of the four ADC ranges. The values shown are As = 0, Ae = +10, Be = 0, Ce = −10 and De = 0.
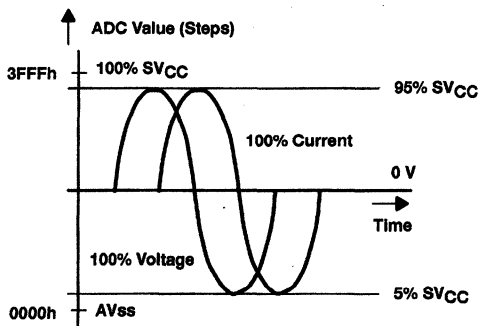
Figure 4–26. Allocation of the ADC Range
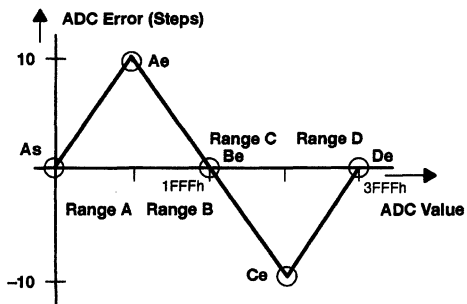
Figure 4–27. Explanation of the ADC Deviation

4.1.12.7 Conclusion

The previous description shows that all steps made from the hardware of an MSP430 ADC are also included in the simulation of the ADC's performance.

4.2 Gas Meter

A gas meter is shown in Figure 4–28 that contains all peripherals modern gas meters can have. The volume interface is shown as a mechanical meter and on the left-hand side for an electronic solution:

❑ The mechanical interface uses contacts to give the volume information to the MSP430. The output Oz is used for scanning, reducing this way the current flow if one or more contacts are closed permanently.

❑ The electronic interface outputs electrical signals to the MSP430 as long as the enable input is high. The signals V1 and V2 are 90° out of phase to allow a reliable distinction of the gas flow direction.

The gas temperature is measured with the ADC of the MSP430. This allows a much better accuracy for the volume measurement, because the dependence of the gas volume to the temperature can be taken into account (laws of Boyle-Mariotte and Gay-Lussac).

Any combination of the peripherals shown can be used for a given solution. It is not necessary to have all of them implemented.

The MSP430 is normally in low-power mode 3 (Icc = 1.6 µA nominal), but all enabled interrupt sources wake it up:

❑ Volume Interface: any change of the volume interface when the output Oz is active (Hi)

❑ Basic Timer: this continuously running timer can regularly wake-up the MSP430 in a very large, programmable time range (2^{-16}s up to 2 s). Its frequency is derived from the crystal frequency (32 kHz).

❑ Key push: all Port0 inputs have an interrupt capability.

❑ M-BUS Activity: via the P0.0 interrupt (RCD).

❑ Insertion of a card into the card interface. All Port0 inputs have interrupt capability.
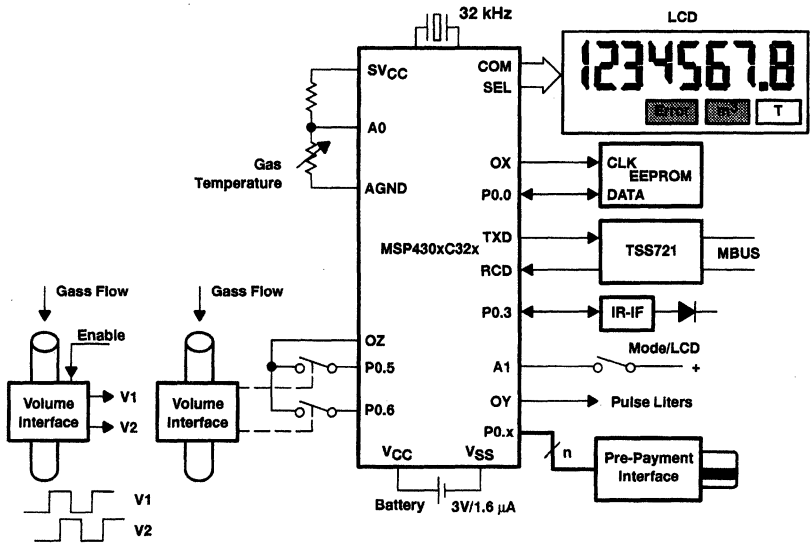
Figure 4–28. Gas Meter With MSP430C32x

The gas meter can also be built-up with the MSP430C31x or MSP430C33x versions. The only difference is the connection of the temperature sensor to the MSP430. The followimg figure shows this configuration.
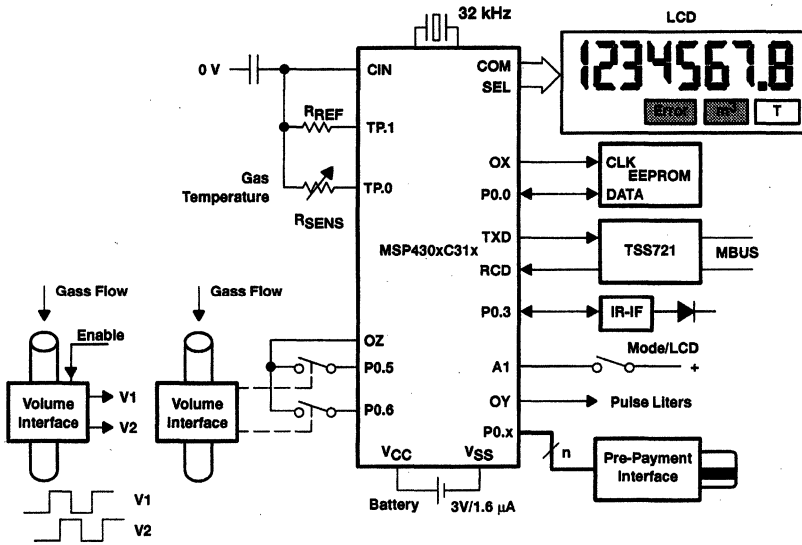
Figure 4–29. Gas Meter With MSP430C31x

Figure 4–30 shows the gas meter of the next generation. All system parts—with the exception of the EEPROMs—are contained on-chip with the MSP430C33x. The interface for the measurement of the gas volume is the same one as shown in Figure 4–29

The MSP430 normally uses the low power mode 3 (I_{CC} = 2 µA typically), but enabled interrupts wake-up the CPU within 8 cycles. See Section 1.4.2, *The Low Power Mode 3* for an explanation.
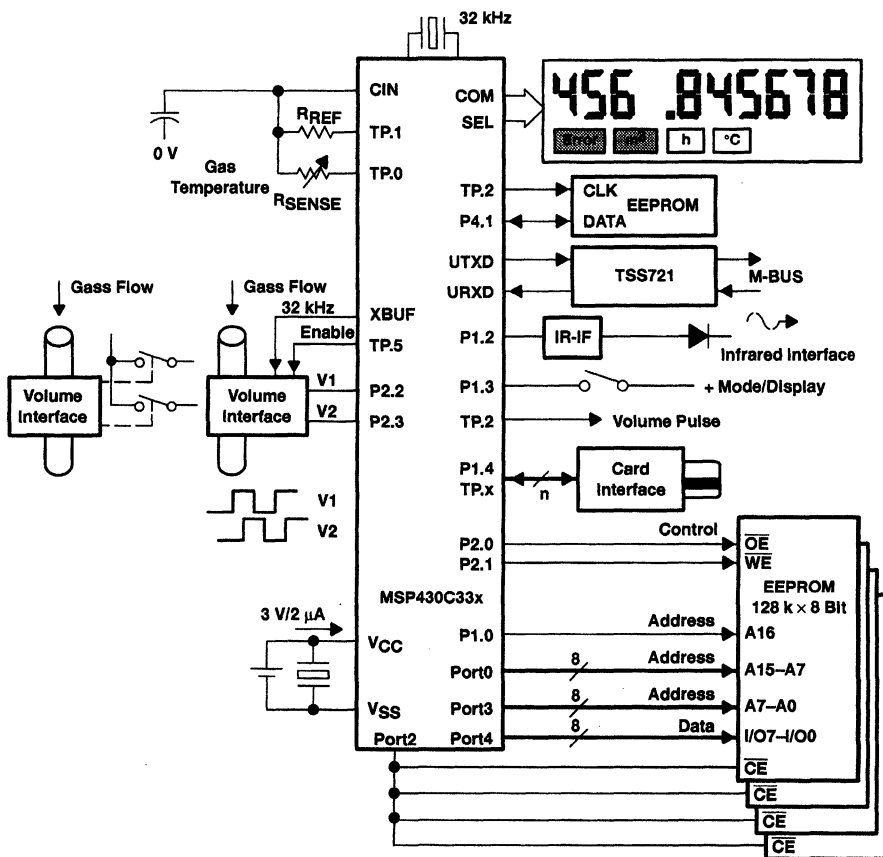
Figure 4–30. MSP430C336 Gas Meter

The EEPROMs (128K × 8 Bits) connected to the MSP430 store the usage profile of the gas meter. The consumed gas volume is recorded in dependence of time. By a regularly occurring read-out—via infrared interface, M-BUS or USART—it is possible for the gas producer to predict the gas consumption very precisely. With applications that do not need this feature, only a small EEPROM (128×8 Bit) is necessary. It contains the system and calibration data and a security copy of the summed-up consummation. The supply voltage for the EEPROMs is switched on and off with unused select lines (O-outputs) of the on-chip LCD driver.

External peripherals which are not necessary for a given application can be omitted.

The MSP430 is normally in low-power mode 3 (Icc = 2 µA nominal), but all enabled interrupt sources will wake it up:

❑ Volume Interface: any change of the volume interface when the output TP.5 is active (high). All Port2 inputs have interrupt capability.

❑ Basic Timer: this continuously running timer can regularly wake-up the MSP430 in a very large, programmable time range (2^{-16}s up to 2 s). Its frequency is derived from the crystal frequency (32 kHz).

❑ Key push: all Port1 inputs have interrupt capability.

❑ M-BUS Activity: via the USART interrupt.

❑ Insertion of a card into the card interface: all Port1 inputs have interrupt capability

4.3 Water Flow Meter

The water flow meter uses an electronic interface to the rotating part of the me-
ter. These signals are 90° out of phase for a reliable scanning of the flow direc-
tion. The MSP430 is in low-power mode 3 normally, but every change coming
from the volume interface wakes it up.

The water flow meter can also be built up with the MSP430C31x version of the
MSP430 family. The only difference is the connection of the sensor for the wa-
ter temperature. See the previous gas meter solution with the MSP430C31x
version for details.
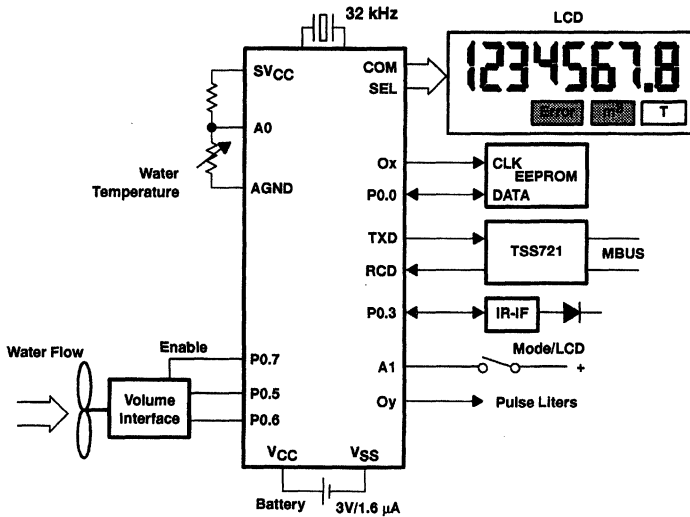
Figure 4–31. Electronic Water Flow Meter

4.4 Heat Allocation Counter

A heat allocation counter with the possibility of sending out the consumption information via RF frequencies is shown in the following figure. The RAM information is scrambled by the DES standard and sent out using the biphase code with 19.2 kBaud. The software routines used for the scrambling and the transmission are contained in Section 5.5.7, *Data Security.*

The heat consumption is computed from the measured room temperature and the heater temperature. The heat consumption is summed up in the RAM and can be read out by the LCD, the M-BUS connection or the RF interface.

The calibration constants and all other important data are contained in the MSP430's RAM. Low-power mode 3 (CPU off, oscillator on) is used normally; the CPU wakes-up at regular intervals (e.g., 3 minutes), measures the heater and the room temperature, and then calculates the actual energy consumption of the radiator. The formulas used take into account the non-linear characteristics given by the thermodynamic theory. This is possible by the use of tables or quadratic or cubic equations.
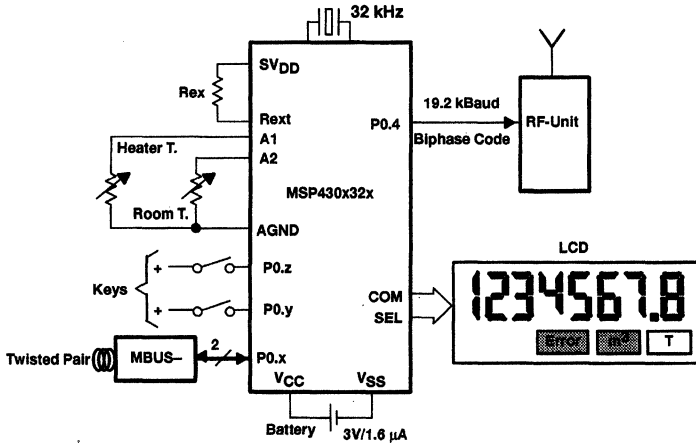
Figure 4–32. Electronic Heat Allocation Meter With MSP430C32x

The heat allocation meter can be built-up also with the MSP430C31x version. Figure 4–33 shows the schematic for this configuration.
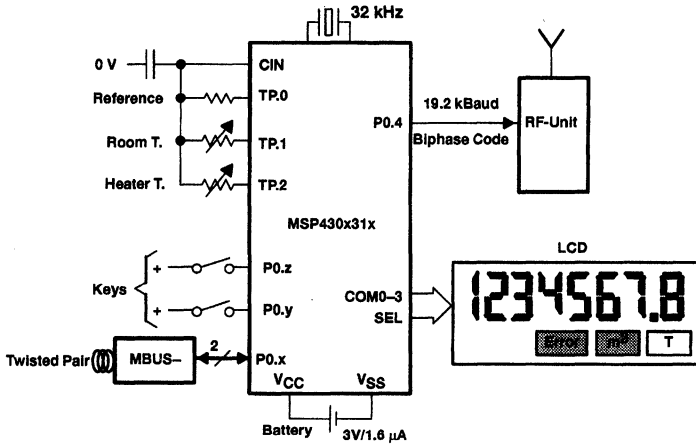
Figure 4–33. Electronic Heat Allocation Meter With MSP430C31x

4.5 Heat Volume Counter

The heat volume counter shown in Figure 4–34 was developed for relatively long sensor lines. An LC filter is used to prevent spikes and noise at the analog inputs of the MSP430. The system normally runs in low-power mode 3 (CPU off, oscillator on) but any change at one of the inputs will wake-up the MSP430.

Every platinum sensor from 100 Ω to 1500 Ω can be used with the MSP430. The current source is able to drive them.
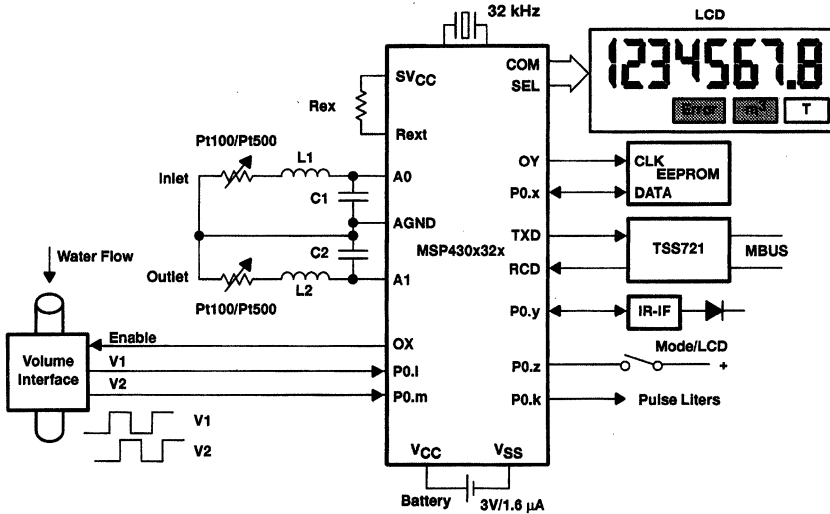
Figure 4–34. Heat Volume Counter MSP430C32x

The four-wire circuitry can also be used here. It is possible to use only five analog inputs with the following schematic. The signals at A2 and A5 can share one input and one resistor connected to AVss.
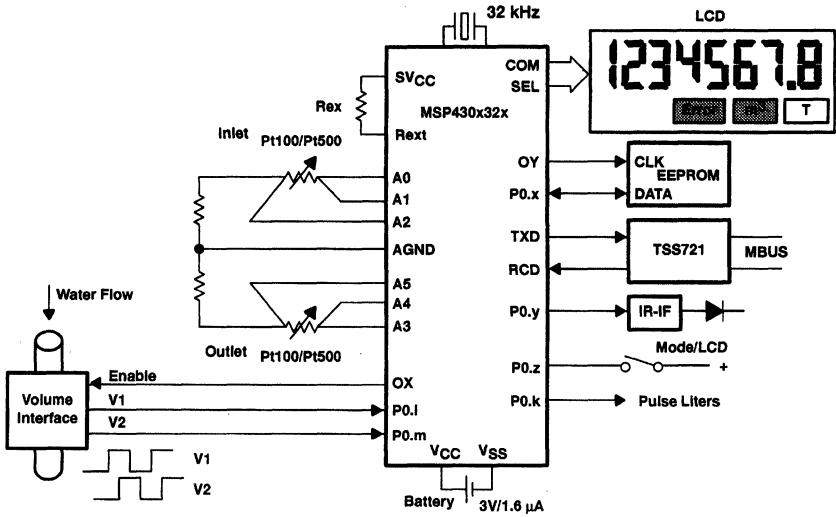
Figure 4–35. Heat Volume Counter With 4-Wire-Circuitry MSP430C32x

Figure 4–36 shows the same heat volume counter as Figure 4–35 but with an enlargement of the ADC resolution to 16 bits. The principle is explained in Chapter 2.
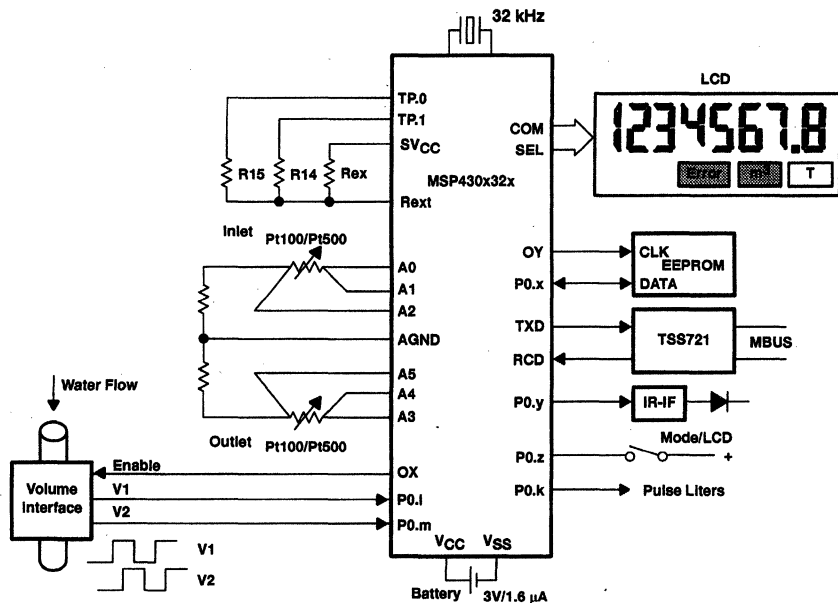
Figure 4–36. Heat Volume Counter With 16 Bits Resolution MSP430C32x

4.6 Battery Charge Meter

The battery charge meter shown in Figure 4–37 monitors the charge of a battery by means of the measurement of all relevant parameters:

❑ Battery voltage is measured with the voltage divider R1/R2. This voltage is used for the recognition of the end of charge (the battery voltage reduces in a defined manner) and for safety reasons.

❑ Battery current: the voltage across a shunt gives an exact indication of the current flowing. The low shunt voltage is shifted into the ADC range by a resistor R3 using the current source of the MSP430. The battery current is measured signed (positive sign means charge, negative sign means discharge) to give the possibility of treating charge and discharge currents differently.

❑ Battery temperature: the resistance of the temperature sensor is measured with the current of the current source.

The battery charge meter shown is not restricted concerning the magnitude of voltage, current, or capacity of the batteries controlled. These depend only on the design of the shunt resistor, the voltage divider, and the calibration constants used. It can be used for cascaded batteries as well as for single ones. This means, it is applicable from camcorders to forklifts.

The reference voltage for the system is delivered by the voltage regulator output. Therefore, the voltage needs to be sufficiently stable. Referencing by a reference diode (LMx85) is also possible. This reference diode can be measured at regular intervals and the result stored. It is not necessary to have the reference always switched on.

The charge indication can be given with a numerical LCD or, as shown in the following, with a battery symbol showing 20% steps. Other methods for indication are also possible (e.g. LEDs with different colors that are enabled for a short time by a key stroke).

The voltage regulator needs to have a very low supply current, not exceeding some micro amps. This is needed because of the long periods the system can be in rest mode (no load). The charge part shown is not necessary for all applications. It can be omitted if, for example, the available space is not provided.

The charge transistor Q1 is switched on by the MSP430 if a certain (low) charge level is reached. The charge current can be fine tuned by PWM. If the charge current is above the maximum current the transistor is switched off due to safety reasons.

The host connection (for example via 232 using the MSP430's UART) can be used for the transfer of data; charge, temperature, voltage, current, and other

system related data. In the other direction, the host can transfer instructions; stop or start of charge, start of data transmission, etc.

The EEPROM contains the characteristic of the controlled accumulator (maximum current, nominal capacity, end of charge criteria etc.) The EEPROM also contains the actual capacity (dependent on age and charge cycles) and a safety copy of the actual charge register. For additional hardware proposals see Section 5.7, *Battery Check and Power Fail Detection*.
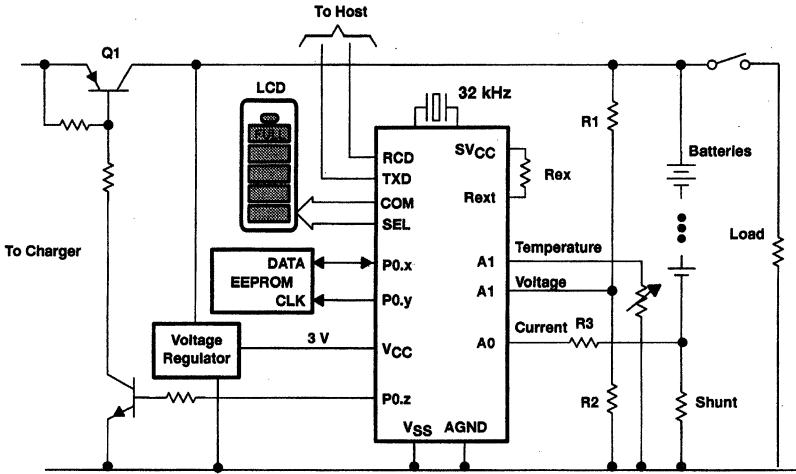
Figure 4–37. Battery Charge Meter MSP430C32x

4.7 Connection of Sensors

The MSP430 family allows the connection of nearly all types of sensors. Some special connections are shown in the following sections.

4.7.1 Sensor Connection and Linearization

Figure 4–38 shows the connection of simple resistive sensors to the MSP430C32x. The current source resistor Rex needs to be calculated in a way that allows its use for both sensor circuits (Rsens2 and Rsens3).

The different connections, shown in Figure 4–38, are described in detail in Chapter 2, *The Analog-to-Digital Converters.*
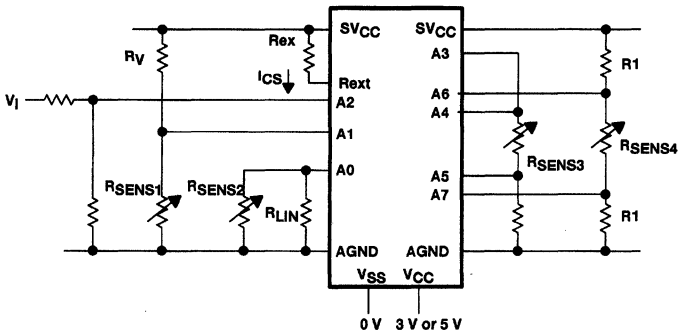
Figure 4–38. Resistive Sensors Connected to MSP430C32x

4.7.1.1 Voltage Supply

The sensor Rsens1, in Figure 4–38, is connected this way. Resistor Rv supplies the sensor and is used for the Linearization also. The optimum value of Rv with dependence of Rsens is:

$$Rv = \frac{Rm \times (Ru + Ro) - 2 \times Ru \times Ro}{Ru + Ro - 2 \times Rm}$$

Where:

Ru	Sensor resistance at the lower temperature limit Tu
Ro	Sensor resistance at the upper temperature limit To
Rm	Sensor resistance at the midpoint temperature (To + Tu)/2

The ADC values measured are independent of the supply voltage Vcc because the measurements are made relative to Vcc.

4.7.1.2 Current Supply

Sensor Rsens2, in Figure 4–38, is connected this way. If a linearization of the sensor is desired, the same formula used for the resistor Rv with voltage supply can be used for the resistor Rlin (see Section 4.7.1.1, *Voltage Supply*).

4.7.1.3 Use of Reference Resistors

Two measurement methods with reference resistors are possible; use of one reference resistor, and, use of two reference resistors:

☐ Measurement with one reference resistor: the reference resistor is chosen so that it equals the sensor resistance at the most important measurement point. Eventually, sensor and reference resistors are selected as pairs. The offset error is completely eliminated. So, only the slope error needs to be corrected.

☐ Measurement with two reference resistors: the two reference resistors represent the sensor resistances at the limits of the measurement range. This method also corrects the influence of the internal resistance (RDSon of the TP outputs). If sensor and reference resistors are paired, no calibration is necessary with this method.

With two reference resistors Rref1 and Rref2 it is possible to compute slope and offset and to get the value of an unknown resistors Rx exactly:

$$Rx = \frac{Nx - Nref1}{Nref2 - Nref1} \times (Rref2 - Rref1) + Rref1$$

Where:

Nx	ADC conversion result for Rx	
Nref1	ADC conversion result for Rref1	
Nref2	ADC conversion result for Rref2	
Rref1	Resistance of Rref1	[Ω]
Rref2	Resistance of Rref2	[Ω]

As previously shown, only known or measurable values are needed for the calculation of Rx from Nx. Slope and offset of the ADC are corrected automatically.
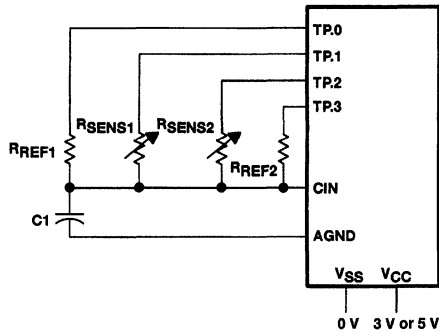
Figure 4–39. Measurement With Reference Resistors

4.7.1.4 Connection of Bridge Assemblies

This kind of sensor is best known for pressure measurement: the voltage difference of the bridge legs changes with the pressure to be measured.
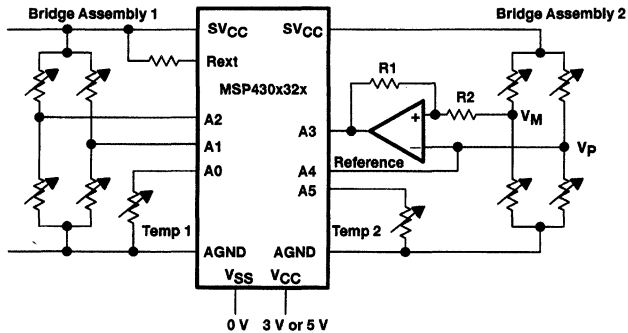
Figure 4–40. Connection of Bridge Assemblies

Figure 4–40 shows on its left side a bridge assembly that creates a voltage difference that is big enough to be measured by the ADC of the MSP430. The measurement result is the difference of the two results of the analog inputs A2 and A1. Due to the temperature dependence of most bridge assemblies, a compensation of this dependence is necessary. The sensor, Temp1, is used to measure the temperature of the bridge legs. It is integrated in some bridge assemblies.

The used formula is:

$$P = MWP \times (Ye + (t - Tk) \times Tke) + Yo + (T - Tk) \times Tko$$

Where:

P	Pressure to be measured
MWP	Difference of the measured values at A2 and A1
Ye	Sensitivity of the pressure sensor
T	Temperature of the sensor
Tke	Temperature coefficient of the sensitivity
Yo	Offset
Tko	Temperature coefficient of the offset
Tk	Temperature during Calibration (e.g. $+25^{\circ}C$)

The units depend on the system used (hP, kg/m^2, kg/mm^2 a.s.o.)

If the difference of the two measurements is too small to be used, an operational amplifier, as shown in Figure 4–40, can be used. Here the ability to measure the reference voltage (one of the two bridge legs) is shown also. Analog input A4 measures the reference that can be used for the A3 input. The same formula as shown previously can be used when MWP is calculated as shown in the following.

MWP   Difference of the measured values at A4 and A3:
$$MWP = (A3 - A4)$$

The actual measured voltage difference ΔV between the analog inputs A3 and A4 is:

$$\Delta V = V_{A3} - V_{A4} = v \times (Vp - Vm) + Vp - Vp = v \times (Vp - Vm)$$

Where:

ΔV	Voltage difference between analog inputs A3 and A4	[V]
v	Amplification of the operational amplifier: $v = R1/R2$	
Vp	Voltage at the bridge leg connected to the non-inverting input	[V]
Vm	Voltage at the bridge leg connected to the inverting input	[V]

The use of the reference input A4 results in correct values for the measurements. If just the differences of two A3 measurements are used, the result needs to be corrected due to the following behavior:

$$\Delta V = V_{A32} - V_{A31} = v \times (Vp2 - Vp1 - (Vm2 - Vm1)) + Vp2 - Vp1$$

$$\Delta V = (v + 1)(Vp2 - Vp1) - v \times (Vm2 - Vm1)$$

It is shown that the voltage differences of the two bridge legs are amplified by different factors (v resp v+1).

4.7.1.5 Fixing of Bridge Assemblies Into One ADC Range

Bridge assemblies normally output only small signals, which makes amplification necessary, and have a relatively high temperature dependency. Both effects together can shift the small amplifier output range over a large input range of the ADC. The four ranges A, B, C, and D of the ADC do not necessarily conform (slope and offset). Figure 4–41 shows a simplified characteristic of the ADC of the MSP430. Two different output ranges of the operational amplifier are indicated. The simplest way to get high accuracy is to fix the output range of the amplifier to only one ADC range; the one where the calibration was made.
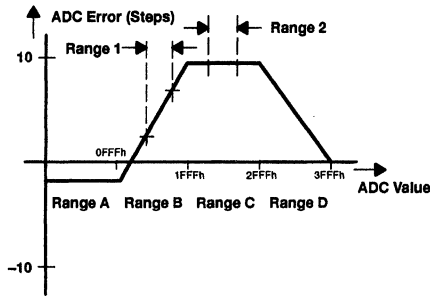
Figure 4–41. Simplified ADC Characteristic

This fix is made by two TP outputs with the resistor values R and 3R (see Figure 4–42). The software modifies the output state of these two TP outputs in a way that for a known state of the bridge (e.g., no load for a scale), the amplifier output is within a certain range of the ADC. Due to the possible TP-port output states Vcc, Vss and high impedance, nine different and nearly equally spaced correction currents Icorr are available. The correction is possible for the positive and for the negative direction. The correction current Icorr can also be fed into the bridge leg, Vm. The equation to calculate the correction resistors R and 3R is:

$$\frac{Rb}{2} \times v \times \frac{SV_{CC} - \frac{SV_{CC}}{2}}{Rl \parallel 3R + \frac{Rb}{2}} \geq \frac{SV_{CC}}{4} \rightarrow R < Rb \times \frac{4v - 2}{3}$$

Where:
Rb   Resistance of a bridge leg   [Ω]

R   Resistance of the correction resistor   [Ω]

v   Amplification of the operational amplifier: v = R1/R2
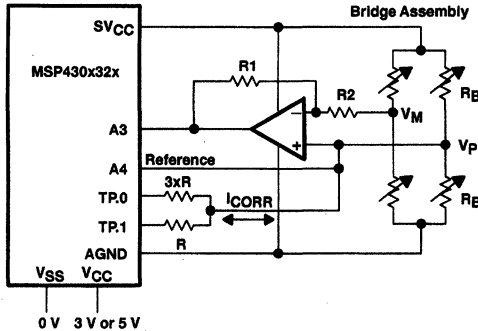
Figure 4–42. Fixing of Bridge Assemblies Into One ADC Range

4.7.2 Connection of Special Sensors

Not just analog sensors can be connected to members of the MSP430 family. Nearly all existing sensors can be connected to the MSP430 in a simple way. The following examples show this.

4.7.2.1 Gas Sensors

The Figure 4–43 shows the connection of two gas sensors (CH_4, hydrogen, alcohol, carbon monoxide, ozone, etc.). The gas sensor at the right side of the figure (connected to A0) is supplied by the internal current source of the MSP430C32x, where the current flowing through the sensor is defined by the resistor, Rex. The gas sensor shown on the left side of Figure 4–43 (connected to A1), owns a load resistance, RL, where the output voltage can be measured with the ADC input A1.

Both sensors are heated by a pulse-width modulated voltage. The midpoint current is 133 mA, the power is 120 mW. The measurement of the sensor resistances is always made during the period with no current flow.

The temperature dependence of the sensor is corrected by the measurement of the sensor temperature. This is made by sensor Temp2.

Only the MSP430C32x can be used for this kind of sensors. They are not potential free so the Universal Timer/Port cannot be used.
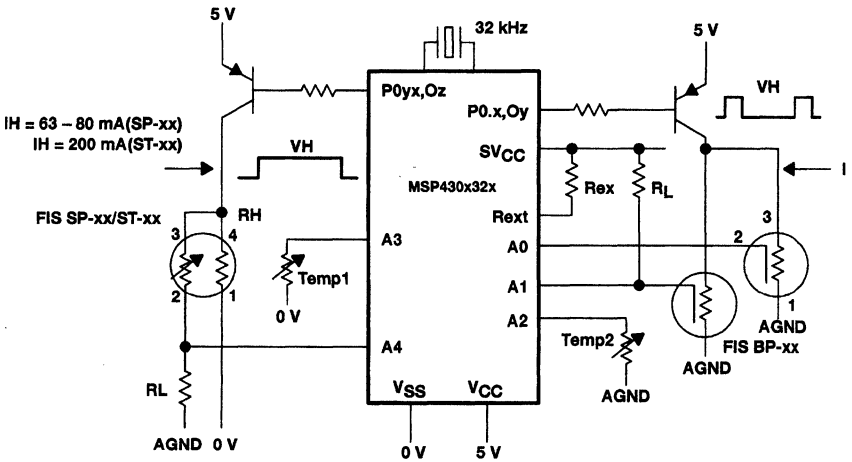
Figure 4–43. Gas Sensor Connection to the MSP430C32x

The left part of Figure 4–43 shows the connection of another gas sensor. The heating of the sensor is done with 5-V dc. The connection is only possible as shown. Therefore, the current source cannot be used. Temperature compensation of the measurement result is needed here also. Sensor Temp1 is used for this purpose.

4.7.2.2 Digital Sensors

Figure 4–44 shows two digital thermometers. They are controlled by instructions via the data bus DQ. The signed measurement result (9 bits) and other internal registers are accessible via the data bus DQ. The circuit shown on the left uses a clock line for the data transfer, the right one differs the signals by their length (short is 1, long is 0).
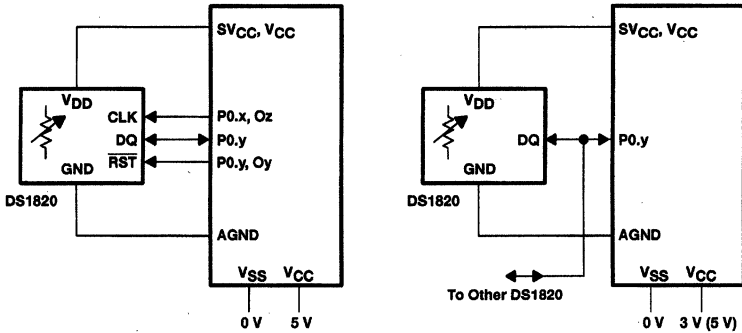
Figure 4–44. Connection of Digital Sensors (Thermometer)

4.7.2.3 Sensors With Frequency Output

The output signal of these sensors is a frequency that is proportional to the measured value. This output frequency can be connected to any of the eight inputs of Port0 and counted via interrupt with a simple software routine. The frequency is the number of interrupts occurring in a one second window defined by the basic timer.

If the frequencies to be measured are above 30 kHz, the Universal Timer/Port or the 8-bit Interval Timer/Counter can be used for counting.

The left part of Figure 4–45 shows the connection of the linear light-frequency converter (TSL220) to the MSP430. The TSL220 outputs a frequency proportional to the incoming light intensity. The range of this output frequency is defined by the capacitor, Cf. Timer_A is ideally suited for these applications (see Section 6.3, *The Timer_A*).
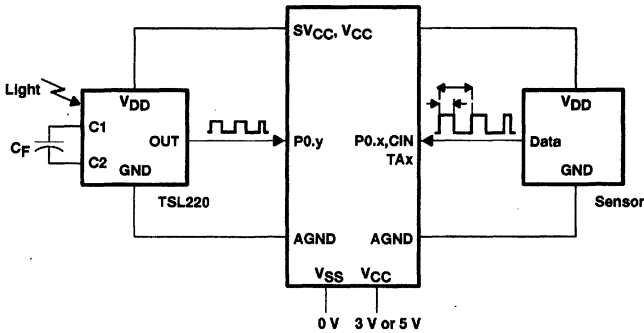
Figure 4–45. Connection of Sensors With Frequency Output Respective Time Output

4.7.2.4 Time Measurements

If the information to be measured is represented by pulse distances or pulse widths, it is also easily measured with the MSP430. The right side of Figure 4–45 shows how this is done.

The signal to be measured is connected to one of the eight inputs of Port0. Each one of these I/Os allows interrupt on the trailing and on the leading edge. With the basic timer, an appropriate timing is selected for the desired resolution and the measurement is made.

The Universal Timer/Port can be used for this purpose also. The pulse to be measured is connected to terminal CIN and the time is measured from edge to edge.

Even better resolution is possible with Timer_A. The input signal is connected to one of the TA inputs and a capture register is used for the time measurements (see Section 6.3, *The Timer_A*).

4.7.2.5 Hall Sensors

Digital hall sensors have an output signal that indicates when the magnetic flux flowing through them is larger or smaller than a certain value. They normally show a hysteresis.

Figure 4–46 shows the connection of a revolution counter realized with the TL3101. Everytime one of the wings breaks the magnetic flux through the TL3101, a negative pulse is generated and outputed. These pulses are counted by the MSP430 with interrupts.
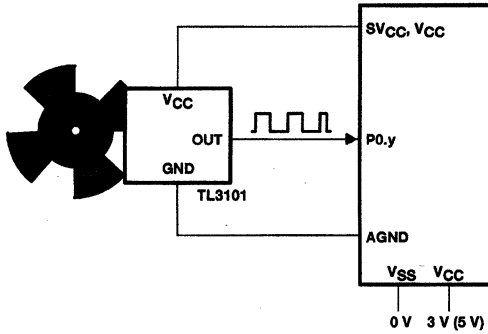
Figure 4–46. Revolution Counter With a Digital Hall Sensor

Analog hall sensors output a signal that is proportional to the magnetic flux through them. For these applications, only the MSP430C32x with its 14-bit ADC can be used. During the calibration, the ADC value at a known magnetic flux is measured and used for the correction of the slope. The ADC value measured at magnetic flux zero is subtracted from any measured value. The calculated correction values are stored in the RAM or in an external EEPROM. For the correction of the temperature coefficient of the hall sensor, a temperature sensor can be used.

Figure 4–47 shows the connection of an analog hall sensor to the MSP430C32x and the typical output voltage dependent on the magnetic flux.
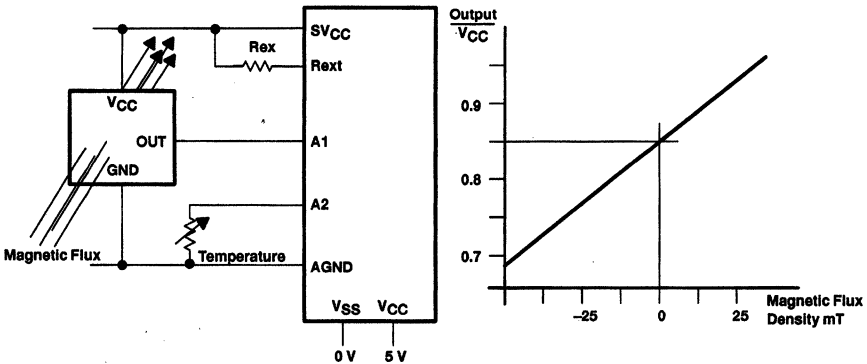
Figure 4–47. Measurement of the Magnetic Flux With an Analog Hall Sensor

4.8 RF Readout

The read-out of metering devices gets more and more important. The next proposals show electricity meters having the possibility to send their consumption information via an RF-transmitter to a host having an RF-receiver. For data security reasons the information sent is normally encrypted with the DES encryption algorithm (data encryption standard). The normally used frequency for this purpose is 433 MHz. But, any other allocated frequency can be used. The modulation mainly used is amplitude modulation, but Biphase modulation can also be used (see Section 4.8.4, *RF-Interface Module*).

4.8.1 MSP430 Electricity Meter

Figure 4–48 shows an electricity meter with the MSP430C32x. This single-chip microcomputer contains all necessary peripherals on-chip except the EE-PROM. The measurement of voltage and current is made with an internal 14-bit ADC.

The interface to ac is shown only for the phase R. The other two phases S and T use the same interfaces.

The RF readout is connected to a free output. This can be an unused segment line or an output bit of Port0. The timing for the RF readout is made by the internal basic timer. It delivers the necessary interrupt frequencies. The supply voltage needed for the RF-interface is done with a step-up voltage supply that transforms the available 5 V to 6 V or more.

The shown hardware proposal uses the reduced scan principle a way of measurement that needs only one ADC. Every measured current sample is used twice; once with the voltage sample measured before and once with the one measured after it. This reduces the number of needed samples without loosing accuracy. A second advantage is reducing by half the number of needed multiplications. This advantage is especially important for microcomputers that do not have a hardware multiplier like most members of the MSP430 family (see Section 4.1, *Electricity Meters*).
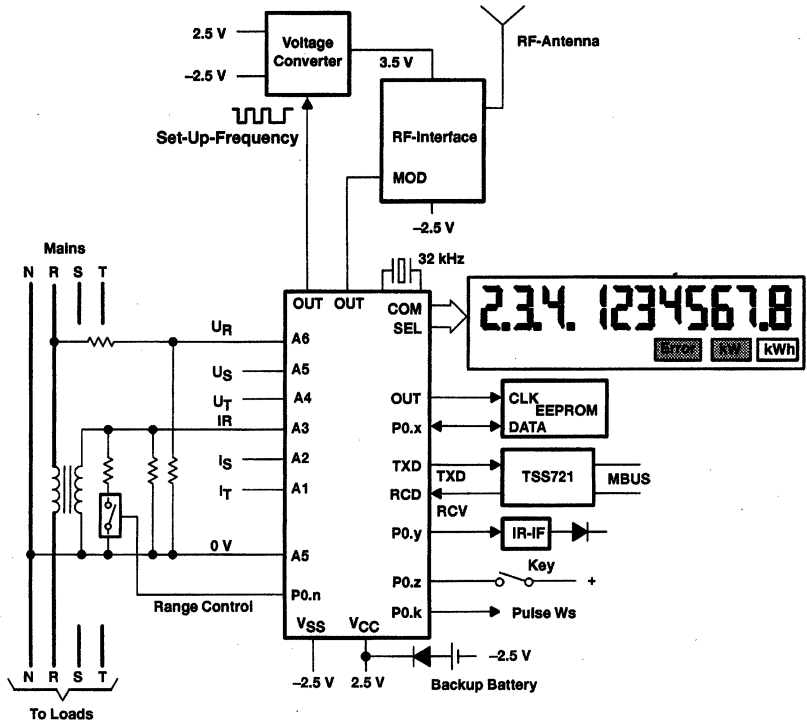
Figure 4–48. MSP430C32x EE Meter With RF Readout

4.8.2 MSP430 Electricity Meter With Front End

Figure 4–49 shows an electricity meter with the MSP430C31x. This single-chip microcomputer contains all necessary peripherals on-chip except the EE-PROM and an ADC. The measurement of voltage and current is made with an external front end. This front end does the scanning, multiplying, and delivers pulses to the MSP430 with a defined value per pulse (Ws/pulse, kWs/pulse etc.). The MSP430 counts and accumulates these pulses.

The interface to the ac is shown only for the phase R. The other two phases S and T use the same interface.

The RF readout is connected to a free output. This can be an unused segment line or an output port of Port0. The timing for the RF readout is made by the internal basic timer, the Timer_A, or by the Universal Timer/Port Module. All of these are able to create the necessary interrupt signals.
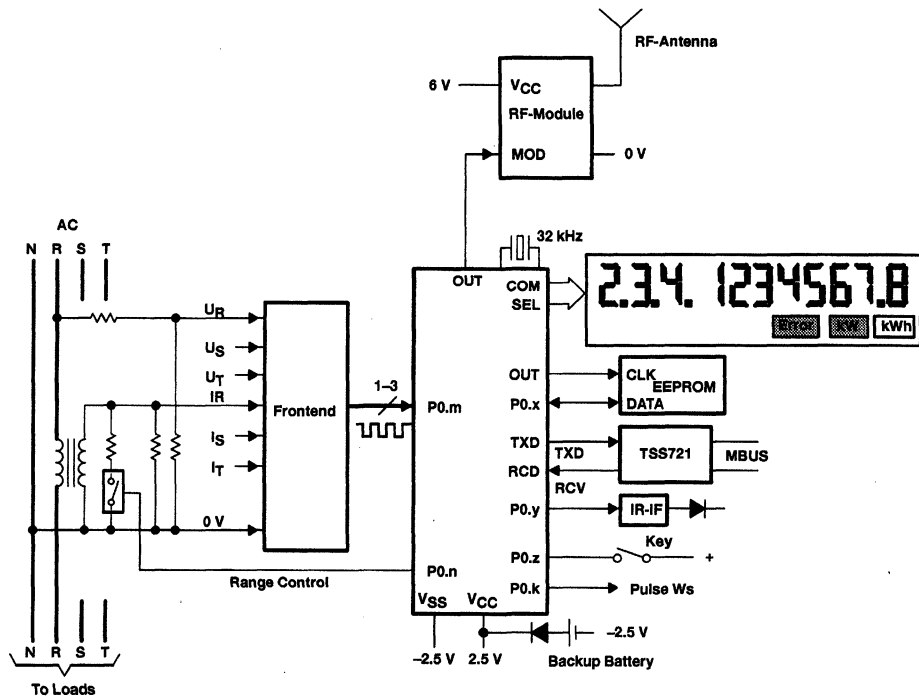
Figure 4–49. MSP430 EE Meter With RF Readout

4.8.3 MSP430 Ferraris-Wheel Electricity Meter With RF Readout

If an RF readout is needed for conventional Ferraris-wheel electricity meters, an MSP430C31x can be used. An optical or magnetic pick-up counts the revolutions of the disk and outputs a signal to the MSP430. The MSP430 computes the used energy and displays it on the LCD. In regular intervals, the measured energy is transmitted using the RF module.
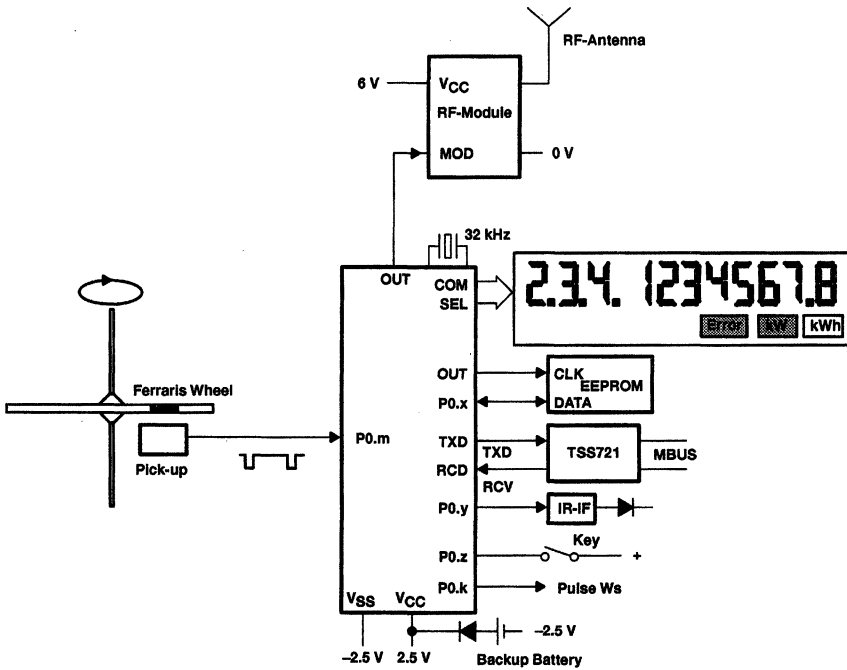
Figure 4–50. MSP430 With a Ferraris-Wheel Meter and RF Readout

4.8.4 RF-Interface Module

The RF-interface module is normally connected to a supply voltage coming from the power supply of the EE Meter. If this voltage is not available, the step-up power supply (shown in Figure 4–51) can be used. An existing supply voltage (here 3 V) is transformed by the step-up circuit to 8 V and regulated down to the desired 6 V. The step-up frequency is delivered by the microcomputer. This frequency starts at a relatively high value and is then lowered to get a good power efficiency.

Complete RF-interface modules are available from several sources.
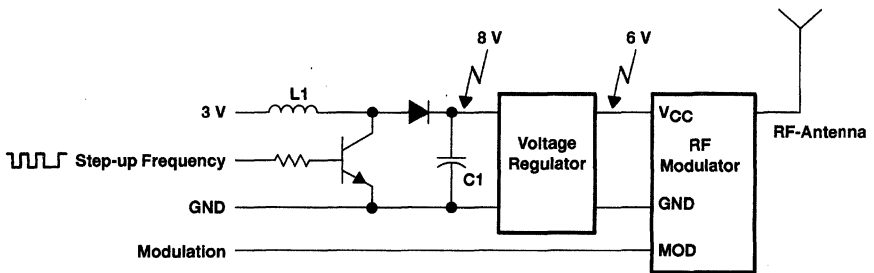
Figure 4–51. RF-Interface Module

Modulation modes used are:

❏ Amplitude modulation—the 433-MHz oscillator is switched on for a logic 1 and switched off for a logic 0 (100% modulation).

❏ Biphase modulation—the information is represented by a bit time consisting of one-half bit without modulation and one-half bit with full modulation. A logic 1 starts with 100% modulation, a logic 0 starts with no modulation.

❏ Biphase space—a logic 1 (space) is represented by a constant signal during the complete bit time. A logic 0 (mark) changes the signal in the middle of the bit time. The signal is changed after each transmitted bit. A stationary transmitter off is also treated as a mark state.

The last two modulation modes do not have a dc part. Figure 4–52 shows all three modulations modes.
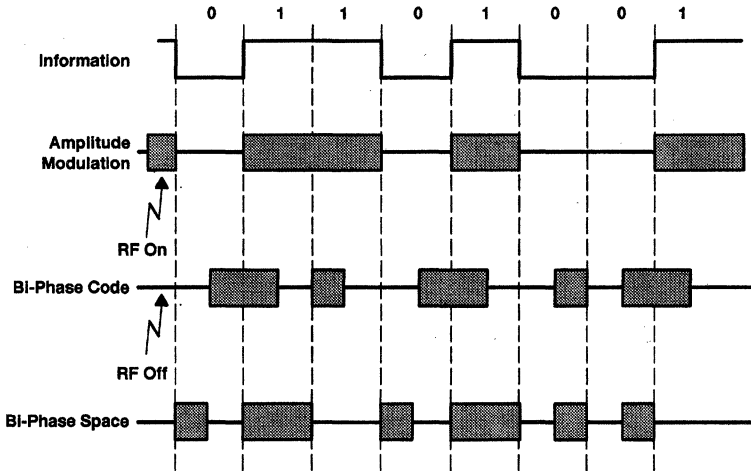
Figure 4–52. RF-Modulation Modes

4.8.5   Protocol

The protocol used with the RF readout is not yet defined. One way is to split the data into blocks of 64 bits and transmit these 64-bit blocks (with or without encrypting them with the DES encryption algorithm).

Another way is to use the M-BUS protocol. If the long-frame format is used, data blocks up to 252 bytes can be transmitted in one frame. If the DES algorithm is used, these data blocks should have a length of 8 bytes (64 bits) due to the definition of the DES algorithm. Figure 4–53 shows a long frame block for 8 bytes of data. This block can be adapted to the needs of the wireless readout, for example the use of a 16-bit checksum due to the higher error probability of this kind of transmission.

Start 68h	Start Character
L Field	Length of Data (8 Bytes)
L Field	Length of Data (Repitition)
Start 68h	Start Character (Repitition)
C Field	Function Field
A Field	Address Field
CI Field	Control Information Field
User Data	Data Field
8 Bytes	Encrypted or Non-Encrypted Data
Checksum	Checksum of Data Field
Stop 16h	Stop Character

Figure 4–53. M-BUS Long Frame Format

To allow the receiver hardware to adapt to the signal strength, a preheader needs to be transmitted in front of the data. This preheader has the same length as the data block and transmits customer owned information (e.g., a series of marks, which results in an alternating sequence of transmitter on and off signals in bit length). Figure 4–54 shows the sequence of a data transfer.

Pre-Header   First Data Block   Second Data Block

Figure 4–54. Sequence of Data Transmission

4.8.6 RF Readout With Other Metering Applications

The RF readout solutions shown can also be used for gas meters, water meters, heat allocation meters, and other metering applications. The voltage supply for the RF part needs to be adapted to the battery used. A battery with a high internal resistance needs a large capacitor in parallel to deliver the necessary current.

4.9 Ultra-Low-Power Design With the MSP430 Family

To get all the low-power advantages that the MSP430 family can provide, some rules need to be kept in mind. This section gives an overview of these rules.

4.9.1 The Ultra Low Power Concept of the MSP430

A lot of microcomputer applications need to be driven by a battery. It is important for these applications to run as long as possible with as small a battery as possible. To reach a battery life longer than 5 years, often the configuration shown in Figure 4–55 is used:

❑ A low-frequency oscillator feeds a prescaler that outputs a pulse every second (or in longer intervals), which sets a flip-flop. This pulse delivers the needed time base with the accuracy of the crystal.

❑ The flip-flop switches on the supply voltage of the microcomputer. The microcomputer measures the necessary system values with an external ADC (necessary if the accuracy needed exceeds 8 bits) and calculates the results afterwards.

❑ The results are summed-up in an external RAM and are displayed with an external LCD driver.

❑ After the completion of all necessary activities, the microcomputer resets the flip-flop and switches itself off in this way. The current consumption of the system reduces to the value that is drawn by the oscillator, the RAM and the LCD driver.

The system shown in Figure 4–55 needs a relatively large battery due to the high number of external system components (>5 Ah).
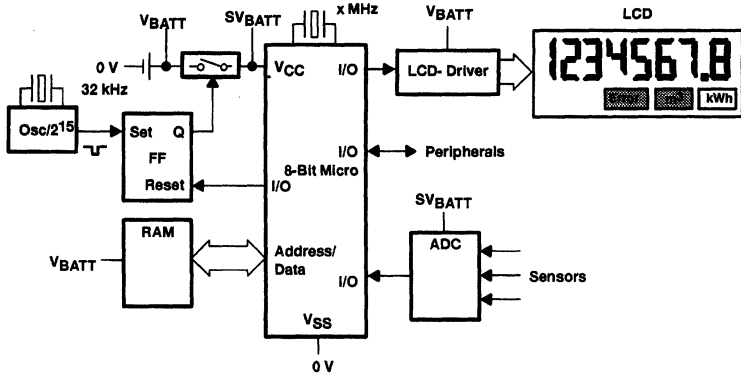
Figure 4–55. Conventional Solution for a Battery-Driven System

The MSP430 family allows the realization of the system shown previously as a one-chip solution with all the external components shown being on-chip peripherals. Figure 4–56 shows this advanced MSP430 solution. The cost advantage with fewer external components is obvious.
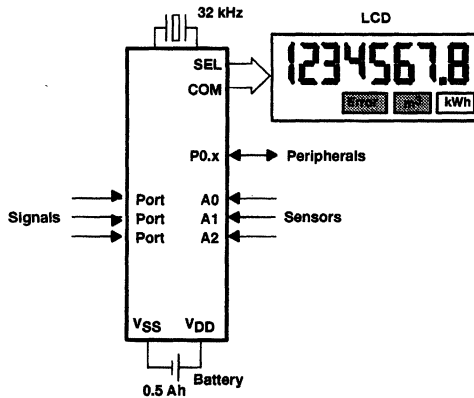
Figure 4–56. Solution With MSP430 for a Battery-Driven System

The only constantly active components are the 32-kHz oscillator, the basic timer (which wakes-up the CPU in regular time intervals), the RAM, the LCD driver, and the interrupt circuitry. The CPU, the ADC, and other peripherals are switched-on only when needed.

The advantages of this concept are:

- Smaller boards due to reduced chip count
- Lower assembly cost with fewer components
- Simplicity of design
- Lower current consumption (smaller power supply or battery needed)
- Faster development

The following examples use the current characteristic shown in Figure 4–57 (these values are only approximated and are not assured).

NOMINAL CHARACTERISTICS OF LPM3 AND LPM4, NO PERIPHERAL MODULE ACTIVE
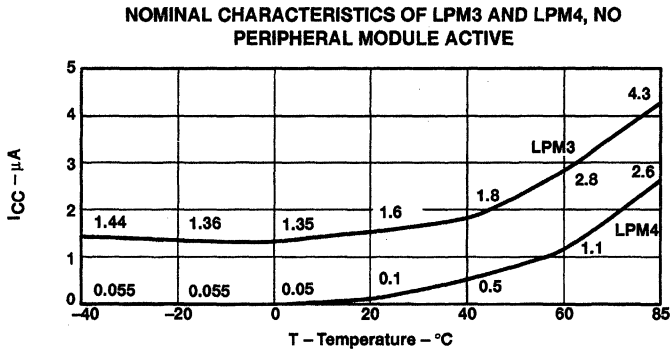
Figure 4–57. Approximated Characteristics for the Low Power-Supply Currents

4.9.2 Current Consumption and Battery Life

To reduce the current consumption of an MSP430 system as far as possible, it is necessary to use low power mode 3 nearly all the time. The basic timer, LCD, and interrupt circuitry are switched on. The CPU is switched off and is active only in programmed time intervals (e.g., every second). The current consumption characteristic of such a system, that is active every second and that measures and calculates only once a minute, looks as seen in Figure 4–58.
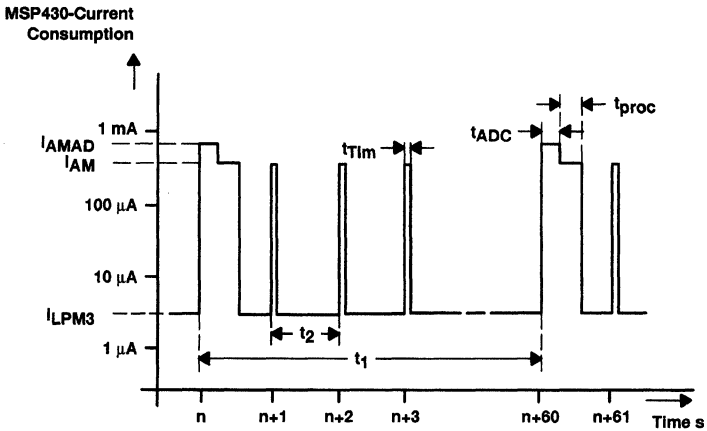
Figure 4–58. Current Consumption Characteristic

Where (all times in seconds):

t_1 Time interval between two measurements (here 60 s)

t_2 Time interval between two wake-ups (here 1 s)

t_{Tim} Processing time after the wake-up. Typically 25 µs to 1 ms.
       (e.g., incrementing of a second counter, check if t_1 elapsed)

t_{ADC} Processing time with switched-on ADC
       (100 µs to 150 µs per measurement)

t_{proc} Processing time with enabled CPU. Typically 1 ms to 100 ms.
       (e.g., calculations after measurements)

t_{LPM3} Time, the system runs in low power mode 3

The average current Icc taken out of the battery by the MSP430 is:

$$I_{CC} = \frac{1}{t1}\left(\frac{t1}{t2} \times t_{Tim} \times I_{AM} + t_{proc} \times I_{AM} + t_{ADC} \times I_{AMAD} + \left(t1 - \frac{t1}{t2} \times t_{Tim} - t_{ADC} - t_{proc}\right) \times I_{LPM3}\right)$$

This can be simplified, if t_{Tim}, t_{ADC} and t_{proc} are much shorter than t_1 (normal case):

$$I_{CC} \approx \frac{1}{t2} \times t_{Tim} \times I_{AM} + \frac{1}{t1}\left(t_{proc} \times I_{AM} + t_{ADC} \times I_{AMAD}\right) + I_{LPM3}$$

EXAMPLE: with $T_A = 20°C$, $t_1 = 60$ s, $t_2 = 1$ s, $t_{Tim} = 0.5$ ms, $t_{ADC} = 0.15$ ms and $t_{proc} = 10$ ms a medium current Icc results:

$$I_{CC} \approx \frac{1}{1s} \times 0.5 \text{ ms} \times 0.35 \text{ mA} + \frac{1}{60s}(10 \text{ ms} \times 0.35 \text{ mA} + 0.15 \text{ ms} \times 0.8 \text{ mA}) + 1.6 \text{ µA} = 1.83 \text{ µA}$$

With the previous example, the current consumption increases by 15% when compared to the consumption of low power mode 3 (1.6 µA).

4.9.3 Minimization of the System Consumption

The overall current consumption of an MSP430 system is composed of three components:

❏ The consumption of the MSP430
❏ The self-discharge of the battery
❏ The consumption of the other system components

The minimization of the current consumption of each of these three parts is discussed in detail.

4.9.3.1 Consumption of the MSP430

The low power mode 3 needs to be the normal mode. Active mode and active mode with ADC are used only when necessary. The rules for the minimization of the current consumption are:

❏ Leaving of the low power mode 3 (wake-up) as rarely as possible, for example only every two seconds.

❏ The program executed after the wake-up should be as short as possible (e.g. incrementing of a counter and test, if other activities are necessary). If this is not the case, immediately return to the low power mode 3.

❏ The time intervals between active periods (calculations) should be as long as possible (e.g., 60 s or longer).

❏ Only the necessary peripherals should be switched-on (e.g., the ADC should be on only during a conversion). After the completion of a conversion, the ADC should be switched-off. This can be supported by the use of the ADC interrupt. The interrupt service routine of the ADC switches off the ADC supply SVcc after the conversion is completed.

❏ Use of the interrupt capability of Port0 to react to external changes. The inputs can interrupt using the leading or trailing edge of an input signal. This ensures the detection of any changes at the inputs without current-wasting polling.

❏ Extremely long calculations (McLaurin-series, Taylor-series) should be avoided. Instead tables should be used. The seven addressing modes, provided by the MSP430, are tailored especially for fast table processing.

❏ Subroutine CALLs should be avoided in frequently used software parts due to the overhead time they need. Instead, the code should be rein-

serted two or three times (like a MACRO). More ROM space may be needed, but fewer CPU cycles are needed.

❑ Short loops should be avoided due to the overhead needed for the loop control. Instead, the loop should be placed into a linear code sequence.

❑ For longer software parts, the working registers R4 to R15, should be used. This results in shorter execution times and in less needed ROM space.

❑ Immediate stop of the calculation if one of the factors—and therefore the result too—is zero

If the previouly mentioned recommendations are applied, the current consumption of the active mode is of second order only. The exceptionally high calculation power of 660 million instructions per Ws (MIPS/W) allows it to ignore the influence of a single instruction. Much more important is the current consumption during the low power mode 3.

4.9.3.2 Self Discharge of the Battery

The self discharge element of the current consumption can not be influenced. The battery manufacturer recommendations should be followed. It is recommended that the battery be placed in a relatively cool location inside of the case. This means do not place the battery next to hot parts (e.g., the radiator to be measured with a heat cost allocator).

An estimation value often used for the self discharge of a battery during 10 years, is to calculate only with 70% of the nominal charge. This relates to 3.5% self discharge per year. Expressed by a discharge current this means 2 μA for a 0.5 Ah battery.

4.9.3.3 Current Consumption of Other System Components

This current is composed of different parts. The most important ones are discussed.

Crystal

The desire for good quality and a low frequency (32,768 Hz) results in a need for a driver power ranging from 1 μW to 10 μW. With a supply voltage of 3 V, the current consumption ranges from 333 nA to 3.33 μA (with an average of 1 μA). This current is always being consumed, because the 32-kHz oscillator is used for the time base.

Liquid Crystal Display

The desire for good quality and a low frequency (128 Hz) results in a need for current consumption of approx. 13 nA/mm^2 segment area. For an LCD with 100 mm^2, this means a current near 1 μA (1.3 μA).

The MSP430 allows the optimization of the chosen LCD with external resistors for the threshold generation.

External Circuitry

Keys and switches connected to inputs that can be closed during long periods (e.g. the contact of a flow meter with no consumption) should have the possibility to be switched-off. This is done to avoid the current flowing through the internal or external pulldown resistor. Figure 4–59 shows three examples:

If a contact is closed longer than a defined time, the external pulldown resistor or the contact is switched off. From then on a regular polling is necessary to monitor contact. If the contact opens again, the normal mode is installed again.
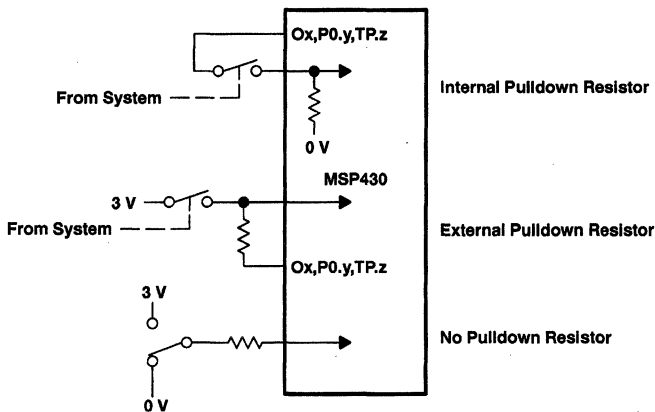
Figure 4–59. Connection of Keys to Inputs

Inputs of the MSP430 should always have a defined potential, otherwise a current flows inside of the input circuitry. Figure 4–59 shows three possible ways to connect an input to a defined potential.

❑ Input with an internal pulldown resistor: The key is switched off with an output. This is made by switching the output to Vss (DVss) or to high impedance.

❑ Input with an external pulldown resistor: The resistor itself is switched to the potential given by the switch. This means that if the switch is open Vss potential, when it is closed Vcc potential.

❑ No pulldown resistor: The switch connects to defined potentials in both positions

External circuitry (e.g., sensors) should be turned off if not in use. This can be established by the use of the SVcc terminal. (Figure 4–60, left). If the current is too high for the SVcc terminal (I >10mA), then a pnp transistor may be used for this purpose (Figure 4–60, right). The SVcc terminal is used then as a reference input for the ADC.

While a 1-kΩ sensor sinks 3 mA when always connected to a voltage Vcc = 3V, the same sensor sinks a very low average current if connected only every 60 s during the conversion time of the ADC (135μs @ ADCLK = 1 MHz):

$$I_{sensor} = \frac{3 \text{ V} \times 135 \text{ μs}}{1 \text{ kΩ} \times 60 \text{ s}} = 6.75 \text{ nA}$$

The average current through the sensor is now only 6.75 nA if it is consequently switched on only during the conversion time.
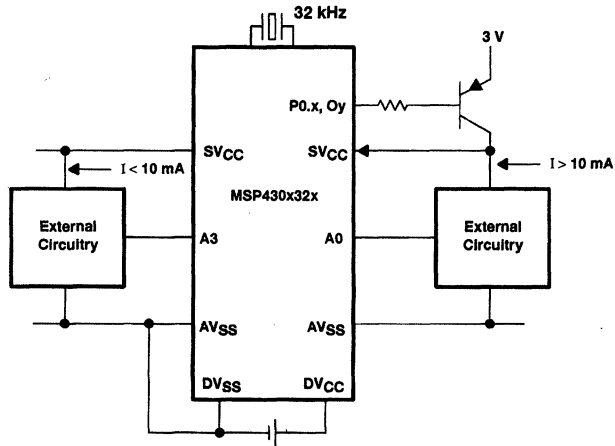
Figure 4–60. Turnoff of External Circuits

With the consumption values now known, the lifetime of the battery can be calculated:

$$t_{Batt} = \frac{Q_{Batt}}{I_{CC} + I_{Sys}}$$

Where:

t_{Batt} Lifetime of the battery in hours

Q_{Batt} Usable charge of the battery in Ah
            (70% of 0,5 Ah for this example)
Icc   Supply current of the MSP430 in A
            (1.83µA for this example)
I_{Sys} Current through the external circuitry
            (crystal, LCD, peripherals) in A (2.3 µA for this example)

For a free-air temperature $T_A = 20°C$ and the consumption values calculated before the lifetime of the battery is:

$$t_{Batt} = \frac{0.7 \times 0.5 \text{ Ah}}{1.83 \text{ µA} + 2.3 \text{ µA}} = 84745 \text{ h}$$

This number of hours is equivalent to 9.6 years.

For ambient temperatures deviating from $T_A = 20°C$ the typical values for I_{LPM3} can be seen in Figure 4–57. The exact values for the self-discharge of a battery can be found in the device specification.

4.9.4   Correct Termination of Unused Terminals (3xx Family)

MSP430 terminals not used need to be treated in a defined manner. Table 4–13 defines the correct termination for every terminal not used in a given application. The termination shown assures lowest supply current.

Table 4–13. Termination of Unused Terminals

PIN	POTENTIAL	COMMENT
AVcc	DVcc	Necessary for EPROM programming also
AVss	DVss	Same as AV_{CC}
SVcc	Open	Can be used as a low impedance output
Rext	Open	
A0 to A7	Open	Switched to analog inputs: AEN.x = 0
Xin	Vcc	If no crystal is used
Xout	Open	If no crystal is used
XBUF	Open	Output disabled
CIN	Vss	Can be used as a digital input
TP0.0 to TP0.5	Open	TP.5 switched to output direction, others to high impedance
P0.0 to P0.7	Open	Unused ports switched to output direction
R03	Vss	Display off: LCDM0 = 0
R13	Vss	
R23	Vss	
R33	Open	
S0 to S1	Open	
S3 to S20	Open	Switched to output direction
Com0 to Com3	Open	
RST/NMI	DVcc resp. Vcc	Pull-up resistor 100 kΩ
TDO		Refer to the specific device data sheet
TDI		Refer to the specific device data sheet
TMS		Refer to the specific device data sheet
TCK		Refer to the specific device data sheet

4.10 Other MSP430 Applications

4.10.1 Controller for a Heating Installation

A very big part of the energy consumed in Europe is used for the heating of rooms. An intelligent controller for heating is a good investment. The controller shown in Figure 4–61 has the following possibilities for the optimum alignment:

☐ Opening and closing of the mixing valve (regulates the mix of hot boiler water with the warm reflux).

☐ Control of the burner (off/on).

☐ Control of the circulation pump (off/on respective of the speed control with a TRIAC). If the outdoor temperature is above a programmable limit (e.g. 20°C) then the circulation pump is off.

☐ Supervision of the boiler temperature: measurement with a temperature sensor.

☐ Supervision of all temperature sensors (feasibility checks)

The criteria for all of these decisions comes from the following inputs:

☐ The measured outdoor temperature is the most important value. It is measured with an outdoor temperature sensor.

☐ The system and calibration data stored in an EEPROM:

■ The individual characteristic of the building stored as the slope and offset for the characteristic of the temperature of the circulating water to the outside temperature

■ The dependence of the boiler temperature on the outdoor temperature

■ The outdoor temperature that stops the activity of the circulating pump (above this temperature only the warm water supply stays active)

■ The minimum switch-on time of the burner (a burner must be on for a minimum time to stay within given environmental limits)

■ Recording of errors for the field service (maintenance)

☐ The mean value of the outdoor temperature for the last 24 hours. This gives a value for the storage of heat in the walls and influences the necessary amount of energy.

❑ The chosen mode:

■ Summer Mode: Only the warm water supply is on, the mixing valve is always closed, the circulation pump is always off

■ Winter Mode: Normal heating is on

■ Maintenance Mode: For repair and maintenance only

■ Day Mode: the heating installation runs always independent of the time

■ Night Mode: the heating installation runs always with the lowered values for the night

■ Switch-off or temperature lowering during the night (circulating pump on resp. off)

The advantages of a microcomputer controlled heating installation are:

❑ Self calibration of the complete system is possible: learning phase and final optimization.

❑ Exact tuning of the optimum mixing temperature due to the involvement of all relevant data

❑ Exact knowledge of the timing for the temperature lowering at evening

❑ Optimum usage of the heating material with minimum pollution

❑ Different concepts are possible for the control of the heating installation
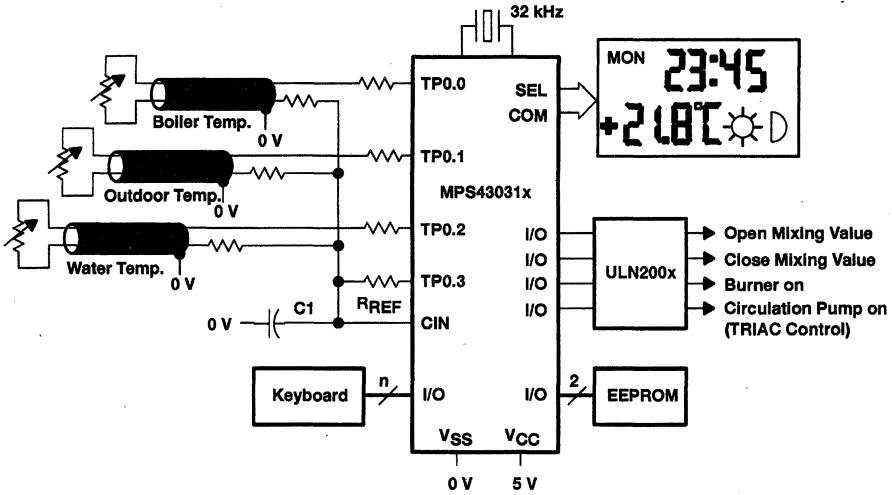
Figure 4–61. Intelligent Heating Installation Control With the MSP430

In a very similar manner to the heating installation controller of Figure 4–61, for a house with more than one zone or area, the MSP430 can also be used in a temperature controller for a single-family home. Figure 4–62 illustrates this example. The boiler control is made by a second MSP430 or by the same MSP430 as shwon in the figure (with the 232 driver shown dotted). The room temperature selection is made with a potentiometer or a small keypad. Both possibilities are shown in Figure 4–62.
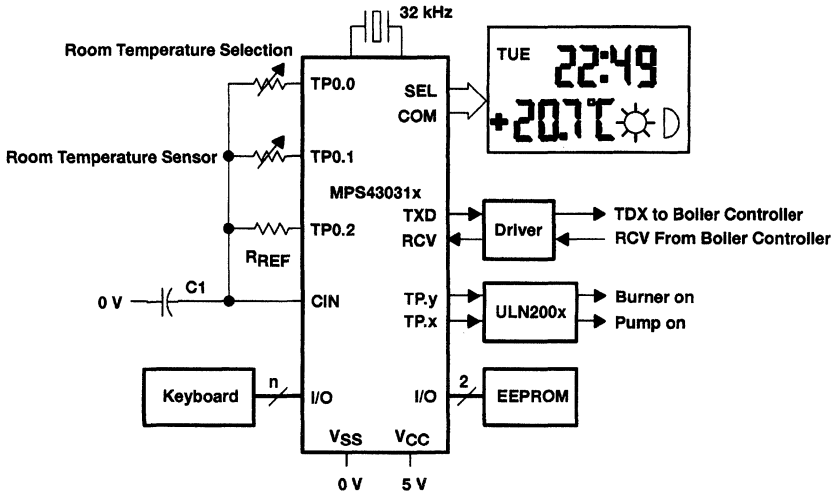
Figure 4–62. Heating Installation Controller for a Single-Family Home

4.10.2 Pocket Scale

Figure 4–63 shows a simple battery-driven scale. The measurement is made with a strain gauge bridge that changes its resistance ratio when loaded with a weight. A tare key allows it to zero the scale in an unloaded state. The measured zero value of the ADC is stored in the RAM and subtracted from every measurement value. To hold the highly temperature-dependent bridge assembly in the ADC range where the calibration was made, a simple hardware fix is added to the MSP430. The fixing of the bridge output is made by two TP outputs with the resistor values R and 3R (see Figure 4–63). The software modifies the output state of these two TP outputs in a way, that for a known state of the bridge (e.g. no load), the amplifier output is within a certain range of the ADC. Due to the possible TP-port output states Vcc, Vss and high impedance, nine different and nearly equally spaced correction currents Icorr are available. The correction is possible for the positive and for the negative direction (signed correction). The correction current Icorr can also be fed into the bridge leg Vm if needed.

The calibration data (e.g., slope and offset) is located in the RAM or—if existent—in an external EEPROM.

The software normally uses the low power mode 3 (LPM3) whenever possible to reduce the current consumption:

❑ After the completion of a measurement and accompanying calculation, the result is moved to the LCD controller and the external components are switched off with SVcc. The CPU is then switched off (with the LCD staying on).

❑ With the On/Off key the scale can be switched off, which means that the LPM3 is used until the On/Off key is pressed once more.

It is also possible to use the low power mode 4 (Icc = 0.1μA) instead of the LPM3.
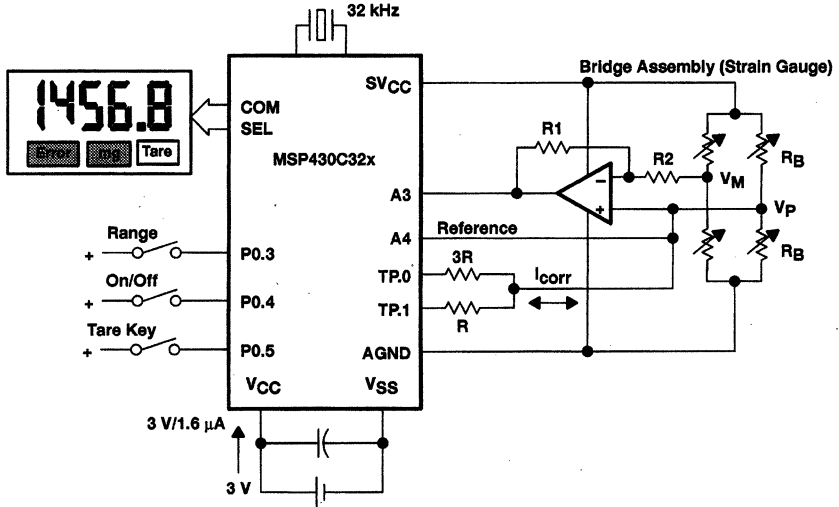
Figure 4–63. Simple Battery Driven Scale

4.10.3 Remote Control Applications

The MSP430 can also be used for remote control applications like a car lock or a TV remote control.

❑ During the inactive time periods LPM3 or LPM4 may be used. This prolongs the life of the battery—even for relatively small ones—to several years.

❑ The interrupt capability of all Port0 inputs (8) ensures an extremely fast response to a pressed key. Without polling of the keypad, the software is within 8 cycles at the start of the interrupt handler.

Figure 4–64 shows a transmitter for security applications. The storage of the secret code for the execution of the function is shown in three ways (only one is actually in use):

❏ Storage in a small EEPROM

❏ Storage in a diode matrix. An inserted diode means a 1 otherwise a 0. The maximum number of diodes defines the number of possible codes (2^n).

❏ Rolling Code: The software generates random numbers and stores them in the RAM. These random numbers are synchronized for the transmitter and the receiver. Any activation of the key means a step to the next code. No external hardware is necessary

If the current through the infrared diode is less than 20 mA, then the npn transistor can be replaced by parallel TP ports. This is shown in Figure 4–64.
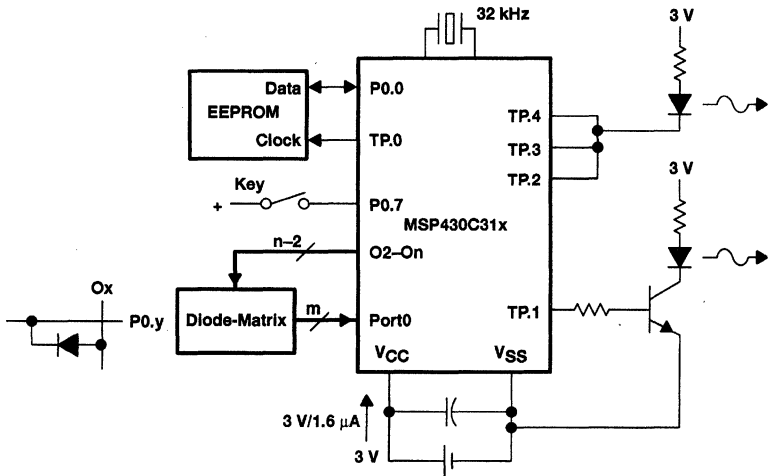
Figure 4–64. Remote Control Transmitter for Security Applications

A remote control transmitter for audio or video sets is shown in Figure 4–65. Normally for this purpose, no external memory is necessary just a keypad with a lot of keys. The high currents through the IR diode need another power stage with a very large capacitor in parallel. This is necessary because the battery cannot deliver the high peak currents.
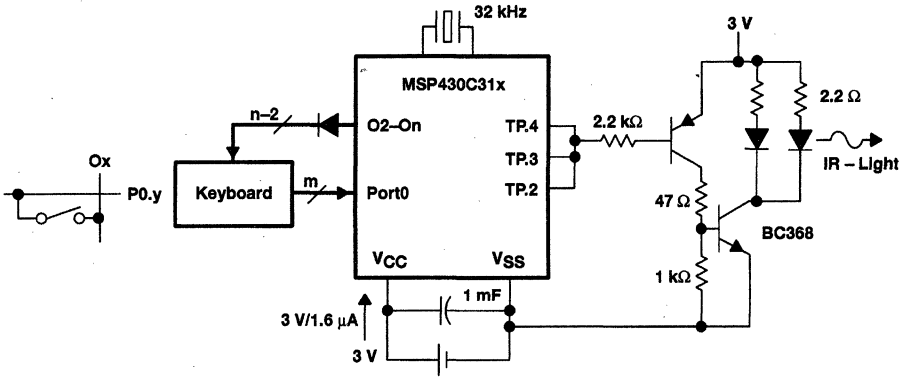
Figure 4–65. Remote Control Transmitter for Audio/Video

The MSP430 can be used also for the remote control receiver. Only a simple, inexpensive IR receiver without decode logic is necessary for this application. The received instruction can be decoded directly by the MSP430 with its high calculation power. This is possible for all the modulation modes used (amplitude modulation, biphase code, biphase space code). The decoded signal is used immediately (car lock application) or given to a host computer (video set).
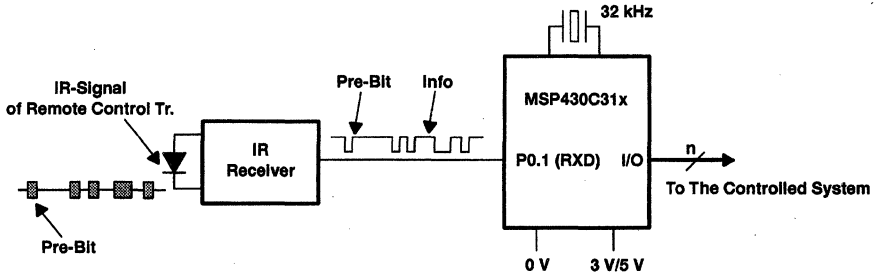
Figure 4–66. Remote Control Receiver With the MSP430

4.10.4 Sub-Controller for a TV Set

The following functions of a TV set can be handled by a an MSP430:

❑ Receive of the IR remote control signals. Only a simple and inexpensive IR receiver without decode logic is necessary. The received information is decoded by software and is sent to the host computer in serial or parallel form.

❏ Keypad scan

❏ Channel display and control of LEDs.

❏ Protection for children: This is done by programming a key sequence that protects the TV set against unauthorized use.

❏ Security function for the host computer: If the host computer does not respond within a given time interval, the MSP430 resets the host.

❏ Real-Time Clock: The 32-kHz ACLK frequency is used for the clock function of the complete system. This can also be used for turn-off sequences; if for longer than 10 minutes, no sender was active, or no remote control input was received.

If an MSP430 is used for the functions described previously, then anything can be switched off except the IR receiver. The power consumption decreases to few milliwatts.

All necessary data and instructions use the infobus (watchdog response, keyboard inputs, remote control signals, LED information etc.).
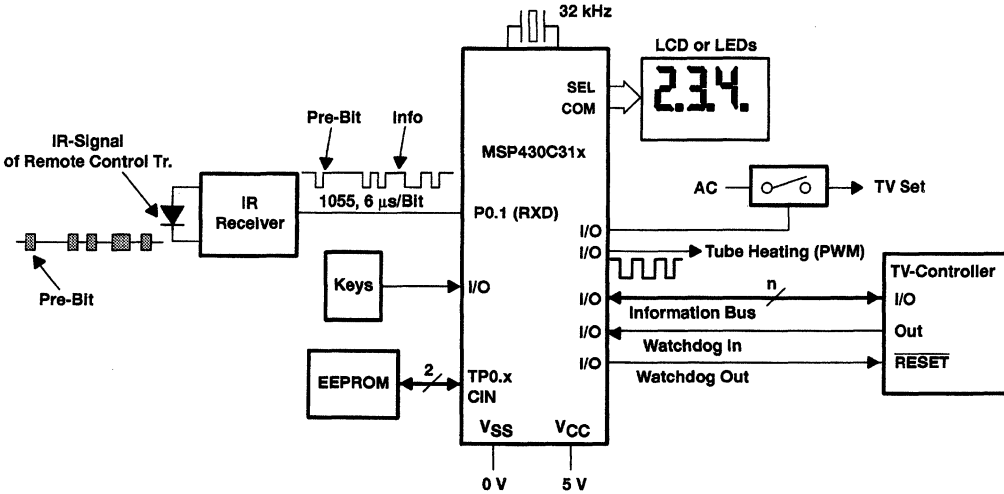
Figure 4–67. MSP430 in a TV Set

4.10.5 Sub-Controller of a Personal Computer

The MSP430 can handle the energy management and switch off all currently unused peripherals (disk, screen, CPU, etc). When they are needed again, it can switch them on in a defined manner. Within an ac-powered PC, the MSP430 can take over the following functions:

☐ Switching off the PC when it is not used for a defined time period.

☐ Watchdog function for the host computer

☐ Defined switch off for all currently unused peripherals

☐ Defined turn on procedure for needed peripherals

☐ Keyboard/keypad scan

☐ Real time clock: The basic timer with its accurate crystal frequency is used for the time base
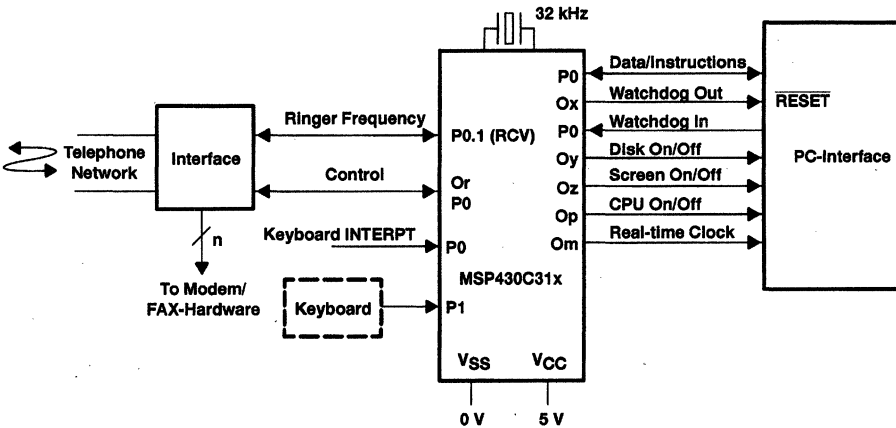
Figure 4–68. MSP430 in an AC-Powered Personal Computer

If the personal computer is powered by a battery (i.e., a Laptop computer), an MSP430C32x can take over the complete battery management:

☐ Turn on and turn off of the charger as indicated by the charge state of the battery

☐ Calculation of the actual charge state out of weighted charge and discharge currents

❏ Battery protection against overcharge, overload, and excessive temperatures

❏ Measurement of current, voltage, and temperature of the battery. The on-chip 14-bit ADC is used for this purpose.

❏ Transmit of the measured and calculated battery state to the host computer.
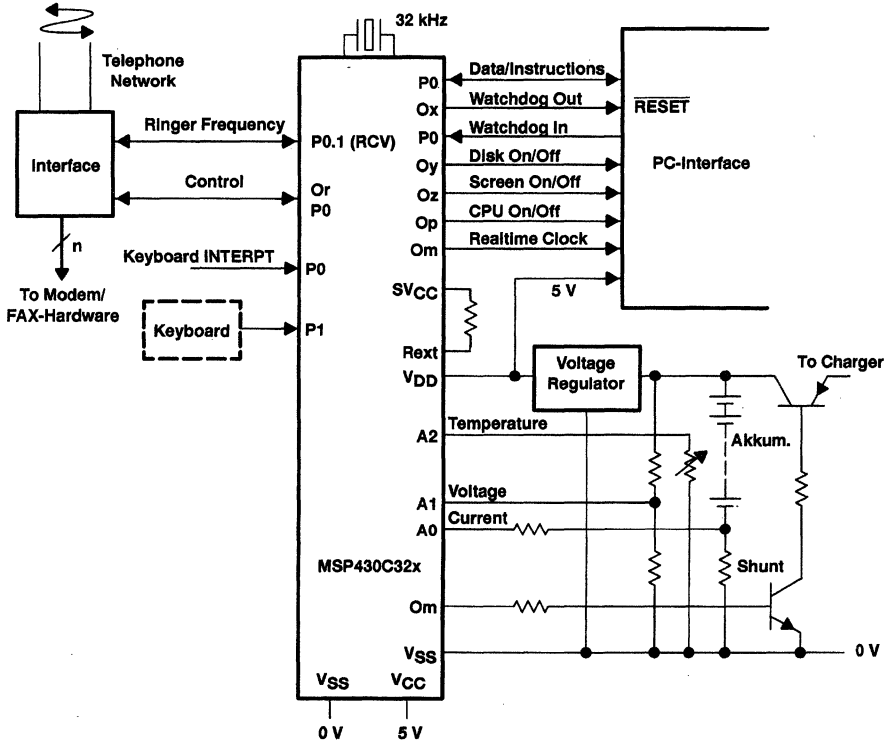
Figure 4–69. MSP430 in a Battery-Powered Personal Computer With Battery Management

4.10.6 Subcontroller of a FAX Device

Within a FAX device, the MSP430C31x can take over the following functions:

☐ Switch off of the device if no activity is needed

☐ Switch on for a recognized telephone call

☐ Keyboard scan and information transmit to the host computer

☐ Display control for a 12.5-digit LCD display

☐ Real-time clock for the complete system

☐ Watchdog function with the on-chip watchdog
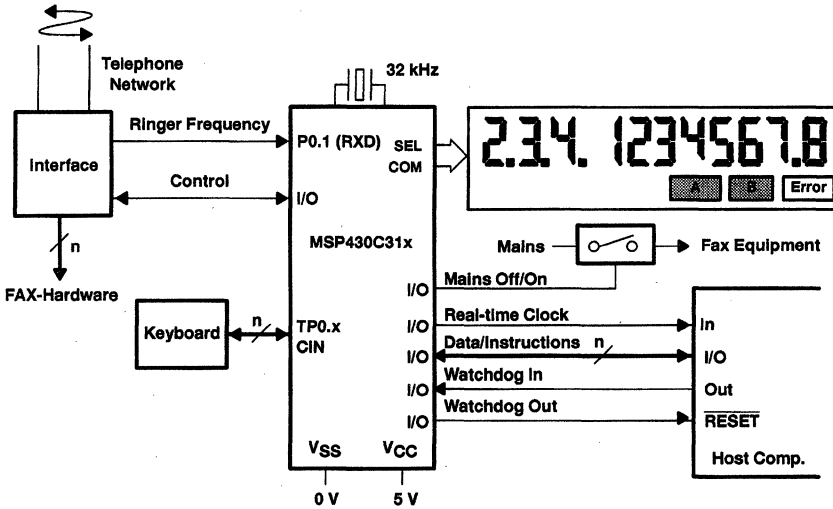
Figure 4–70. MSP430-Controlled FAX Device

4.11 Digital Motor Control

The MSP430 family is shown with digital motor control (DMC) applications. Several hardware proposals are given for pulse width modulation (PWM) and TRIAC-control applications for electric motors. Numerous circuit and pulse diagrams show the application of the MSP430 family for different electric-motor types and control concepts. For each hardware proposal the applicable motor types are named.

4.11.1 Introduction

The application of DMC has some advantages compared to conventional concepts for motor control:

❏ Better energy efficiency
❏ Better control of motor behavior (speed, torque, direction of rotation)
❏ Easy supervision of important motor conditions (temperature, current, speed)
❏ Use of smaller motors due to the better adaptability to the given application
❏ Use of motor types not applicable without DMC (brushless dc motors, reluctance motors)

If the accuracy of fixed-point calculations is not sufficient then a floating point package (FPP), designed especially for real time applications, is available from TID. This memory and speed optimized FPP can be configured for two different number formats: 32 bits or 48 bits. The high speed results from the RISC-mode it uses (mainly single-cycle instructions) and the involved hardware multiplier (see Section 5.6, *The Floating Point Package*).

Additionally a C Compiler with a very good code efficiency is available.

4.11.1.1 The MSP430 Family

The MSP430 family with its 16-bit RISC architecture is capable to realize very advanced control concepts. This is especially true for the MSP430C33x with its hardware multiplier (16 x 16 bits) and its 16-bit Timer_A allowing four independent PWM-outputs. All MSP430 family members use the same instruction set and the same CPU. This eases the use of existing user software enormously.

Operating frequencies up to 3.8 MHz and single-cycle instructions when the register/register addressing mode is used for the source and the destination—the normal addressing combination for real-time applications—results in calculation speeds formerly only known by DSPs. This high throughput allows calculations and algorithms needing more than 16 times the capability of 8-bit microcomputers.

Actually, the MSP430 family consists of three different subfamilies. The hardware peripherals of the different subfamilies are listed in Table 4–14. The instruction set is the same for all members of the family.

Table 4–14.   Peripherals of the MSP430 Sub-Families

HARDWARE ITEM	MSP430x31x	MSP430x32x	MSP430x33x
LCD Segment lines	23	21	30
14-Bit ADC	No	Yes	No
Universal Timer/Port Module	Yes	Yes	Yes
I/Os with Interrupt	8	8	24
I/Os without Interrupt	0	0	16
16-Bit Timer_A	No	No	Yes
USART (SCI or SPI)	No	No	Yes
HW/SW UART	Yes	Yes	Yes
Watchdog Timer	Yes	Yes	Yes
16 × 16 HW Multiplier	No	No	Yes
Basic Timer	Yes	Yes	Yes
Oscillator FLL	Yes	Yes	Yes
LPM3 (Sleep Mode)	Yes	Yes	Yes
LPM4 (Off Mode)	Yes	Yes	Yes
Package	56 SSOP	64 QFP	100 QFP

The Low Power Modes

The MSP430 family is designed for minimum power consumption. This feature—which allows full CPU activity with only 400-μA current consumption (730 μA for the MSP430C33x) for an MCLK frequency of 1 MHz—reduces the size of the power supply to a minimum. Five different LPMs are implemented. The nominal supply currents I_{AM} are shown for the MSP430C33x. The supply voltage is 5 V and the temperature range is from T_A = –40 to 85°C.

❏   LPM0: the CPU is switched off, the 32-kHz oscillator (ACLK) the main clock MCLK (with enabled loop control) and the peripherals are active. I_{AM} = 120 μA

❏   LPM1: the CPU is switched off, ACLK, peripherals, and MCLK (with disabled loop control) are active. I_{AM} = 120 μA.

❏   LPM2: the CPU and MCLK are switched off, ACLK and peripherals are active. I_{AM} = 18 μA.

❏   LPM 3: the CPU is switched off, ACLK is active, the basic timer, the watchdog, and the interrupt hardware can be active (if enabled). I_{AM} = 5.2 μA.

❏ LPM4: all parts of the MSP430 are off, only the RAM and the interrupt hardware are powered. $I_{AM} = 0.4 \ \mu A$.

The motor control software can use these low-power modes to reduce the power consumption to a minimum after the completion of the necessary calculations and control functions.

4.11.1.2 The 16-Bit Timer_A

The features of the MSP430 Timer_A are very important for the pulse width modulation necessary for the DMC (see Section 6.3, *The Timer_A*, for explanations of the possibilities of this timer.

4.11.1.3 The Universal Timer/Port Module

MSP430 family members that do not contain the Timer_A, contain at least the Universal Timer Port/Module (UTPM), a combination of two 8-bit timers with a common control unit and inputs and outputs. The UTPM is primarily thought as an ADC but it is also able to handle timing tasks that are not too complex. To get an interrupt request after a certain number of MCLK or ACLK cycles it is only necessary to load the negated number of cycles into the count registers TPCNT1 and TPCNT2. When the 16-bit counter (used with MCLK) or one of the 8-bit counters (used with ACLK) overflows to zero, the corresponding interrupt flag (RC2FG or RC1FG) is set and an interrupt is requested. This method allows precise timings for TRIAC control or PWM control in the range of 128 Hz to 4000 Hz (repetition rate). This frequency range allows the replacement of PWM-control arrangements realized by relays, a solution sometimes needed in automotive applications.

The UTPM can be used for:

❏ Low-frequency pulse width modulation (see Section 3.6.4, *PWM DAC With Universal Timer/Port Module*); up to two independent PWM outputs are possible.

❏ Measurement of the MCLK frequency when used without a crystal (see Section 6.5.8, *Use Without Crystal*)

❏ TRIAC-triggering: time measurement starting with the zero crossing of the ac voltage

❏ Other time measurements
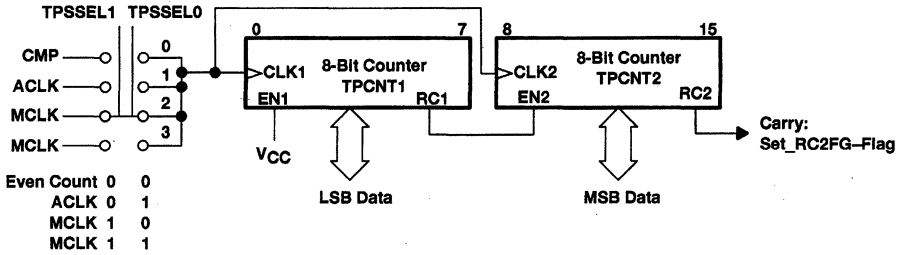
Figure 4–71. Block Diagram of the UTPM (16-Bit Timer Mode)

Figure 4–72 shows the generation of a low-frequency PWM with the UTPM alone. The timing for the period and for the pulse width is made by it. If the ACLK frequency is used for the timing, then two PWM outputs with up to 256-Hz repetition rate are possible. The resolution for this case is 128 steps.

The formula for the period t of the PWM frequency is:

$$t = \Delta t1 + \Delta t2 = \frac{n1 + n2}{f_{clk}}$$

The formulas for the pulse width $\Delta t1$ and the corresponding value $n1$ are (the negative value of $n1$ is loaded into TPCNTx):

$$\Delta t1 = \frac{n1}{f_{clk}} \rightarrow n1 = f_{clk} \times \Delta t1$$

$n2$ and $\Delta t2$ are calculated the same way as $n1$ and $\Delta t1$.
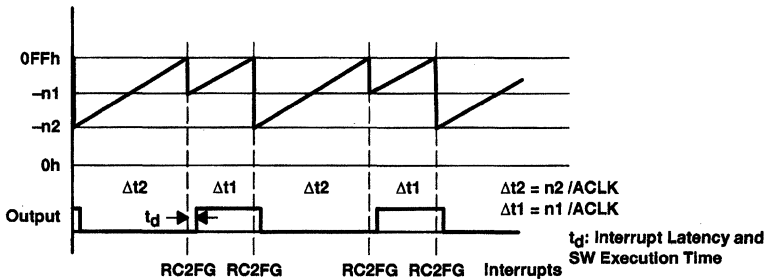
Figure 4–72. Low-Frequency PWM-Timing generated With the Universal Timer/Port Module

Figure 4–73 shows a solution that is synchronized by the basic timer (only one PWM timing is shown). Its interrupt (here with 256 Hz) sets the enabled PWM outputs and loads TPCNT1 and TPCNT2 with the corresponding negated clock cycles. The PWM outputs are reset by the interrupt software of the UTPM. The software is described in the Section 3.6.4, *PWM DAC With the Universal Timer/Port Module*).
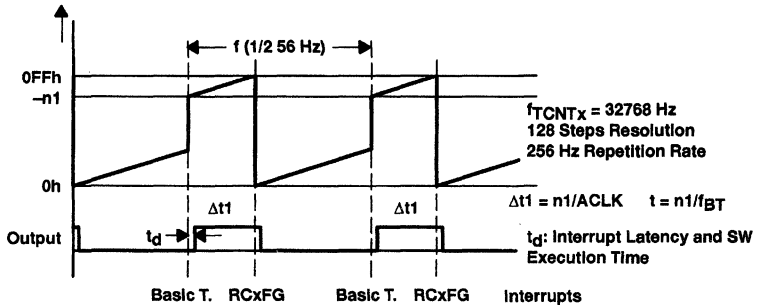
Figure 4–73. Low Frequency PWM Timing by Universal Timer/Port Module and Basic Timer

4.11.1.4 The Basic Timer

Additional to the timers mentioned previously a third timer exists that is responsible for the time base (date and time). This timer runs completely independent of the other timers and outputs frequencies (0.5 Hz to 65536 Hz) derived from the crystal (ACLK) or the system clock generator (MCLK). This way the Timer_A and the UTPM are completely free for real time operations.

4.11.1.5 The Watchdog Timer

This 15-bit timer can be used for simple timer tasks or for security purposes. If it is not reset during a selectable time interval, then the watchdog timer resets the MSP430. This allows to reinstall the lost system integrity. The watchdog timer is switched on during the powerup and is active immediately.

4.11.2 Digital Motor Control With Pulse Width Modulation (PWM)

Two modes of the Timer_A—the Up Mode and the Up/Down Mode—are developed especially for PWM generation. These two modes are used with all hardware proposals of this section.

4.11.2.1 Single Output Stages

If only one direction of rotation is necessary, or the change of the direction of rotation can be made with a relay having change over contacts, then a single output stage can be used.

The direction of rotation of the motor is changed by a relay that switches the polarity of the field winding. For only one direction of rotation, this relay is omitted and the field winding is connected in a fixed way. All of the examples shown can use the high-frequency PWM (> 15kHz) or the low-frequency PWM (100 Hz and higher).

The formulas for the coming circuit proposals are:

$$Vm = \frac{n_{CCRx}}{n_{CCR0}} \times V_{motor} \rightarrow n_{CCRx} = \frac{Vm}{V_{motor}} \times n_{CCR0}$$

Where:

Vm	Mean voltage at the motor	[V]
V_{motor}	Voltage of the motor power supply	[V]
n_{CCRx}	Content of Compare Register x	
n_{CCR0}	Content of Compare Register 0 (Period Register)	

If the Up Mode of the Timer_A is used, then n_{CCR0} must be substituted by $n_{CCR0}+1$.

Single Output Stage With a Bipolar Power Transistor

Figure 4–74 shows a single output stage with an npn power transistor. The PWM signal generated by Timer_A is amplified by two inverters and connected to the base of the power transistor. The inverters used must be able to drive the relatively high base current of the power transistor. Eventually several inverters need to be connected in parallel at the outputs, where serial resistors force an equal current distribution. Figure 4–74 shows such a driver stage at the lower right-hand corner. A simple configuration with only a pnp and an npn transistor is possible. This driver stage is shown in Figure 4–74 at the lower left-hand corner. The EEPROM connected to the MSP430 contains the characteristic of the controlled motor.
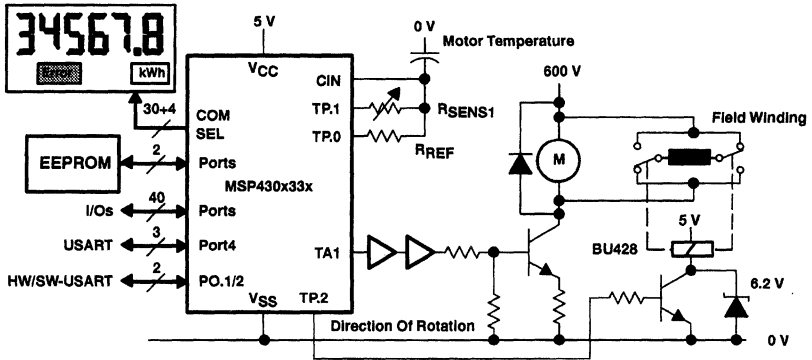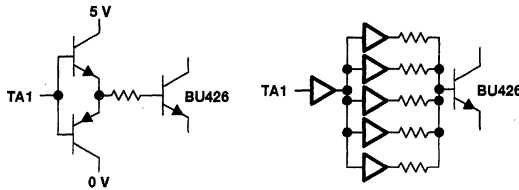
Figure 4–74. Transistor Output Stage Allowing Both Directions of Rotation

❑ Applicable for

■ DC motors, universal motors

❑ Advantages

■ Minimum component count for only one direction of rotation

Single Output Stage With a MOSFET Power Transistor

Instead of npn power transistors it is possible to use power MOSFETs or IGBTs. Figure 4–75 shows a circuit with a dual MOSFET TPIC2202 and the appropriate MOSFET driver SN75372. An MSP430C33x controls two PWM outputs. If the calculations for the control of the motors are not too complex then it is possible to control up to four motors with a single MSP430C33x.

If a change of the rotation direction is needed then a relay can be used as shown in Figure 4–74.

The temperature of the motors can be observed with a temperature sensor, e.g., an NTC sensor. Figure 4–75 shows the circuitry needed for the connec-

tion of two temperature sensors to the ADC inputs of the MSP430C33x. The motor temperatures are measured in appropriate time intervals to be sure, that typical circumstances are present. In case of a overly high motor temperature, the microcomputer switches off the MOSFET power transistor and switches on a fault indication LED.

The observation of the motor current is realized with an operational amplifier working as a comparator. The circuitry shown allows eight different thresholds, a number that can be modified easily if necessary. The control ports P3.x switch between the high state and the high-impedance state to get eight different thresholds (corresponding to eight different temperatures).
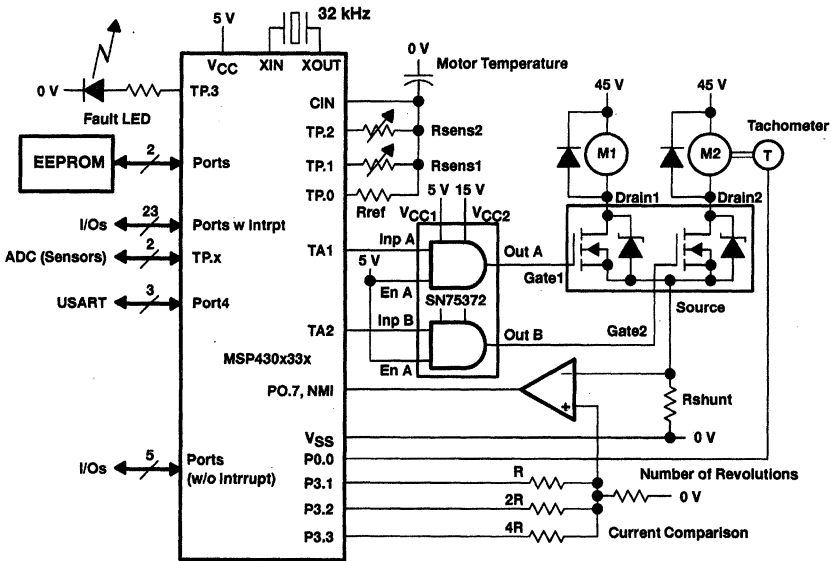
Figure 4–75. Control for Two MOSFET Output Stages

❑ Applicable for

■ DC motors, permanently excited dc motors

❑ Advantages

■ Control for two motors with minimum chip count

The MOSFETs shown in Figure 4–75 allow up to 7.5-A continuous current simultaneously for both transistors. The TPIC2202, having only one source ter-

minal, allows only the measurement of the sum of both motor currents. If it is necessary to observe the motor currents independently, then the TPIC5201 can be used. This dual power MOSFET features two source terminals. For motor currents up to 3 A, the 15-V supply for the SN75372 is not necessary, a 5-V supply is sufficient for switching on of the MOSFETs.

4.11.2.2 H-Bridge Output Stages

An H-bridge means the fourfold expense for power drivers compared to a single output stage. But if integrated drivers are used, the resulting expense is often lower than with a single output stage because the change of the direction of rotation is included with the H-bridge.

H-Bridges for Low Motor Voltages

For voltages up to 36 V, TI offers several solutions. Into this range belong the automotive sector and industrial control applications working with 24-V supply voltage.

Output Stage With a MOSFET Bridge

Motor control applications working with relatively low voltages can use the Texas Instruments H-bridge TPIC5424. This device is able to switch currents up to 3 A at a maximum voltage of 60 V. The complete circuit diagram is shown in Figure 4–76.

The gate voltage necessary for the turn-on of the upper MOSFETs of the bridge is generated by a bootstrap circuit. This gate voltage must be at least 5 V higher than the motor voltage. The MSP430 generates this support voltage using two capacitors and the low impedance power driver BT1.

Three simple solutions are possible for the generation of the higher gate voltage:

❏ PWM-output TA3 is used with the full PWM frequency (e.g. 19.2 kHz)

❏ PWM-output TA0 is used with the divided PWM frequency (e.g. 9.6 kHz for 19.2 kHz). This is due to the only possible output mode for the period register, CCR0: Toggle/Toggle Mode. This way has the advantage that no other timer output is needed. Figure 4–76 illustrates this solution.

❏ One of the available output frequencies of the XBUF output is used: ACLK, ACLK/2 or ACLK/4 corresponding to 32.768 kHz, 16.384 kHz or 8192 Hz.

Solutions 2 and 3 have the advantage of an always usable output voltage. They are independent of the PWM output driving the electric motor.

> **Note:**
>
> The power inverter shown, BT1, can be replaced by some paralleled—not used otherwise—output ports of the MSP430C33x. They are toggled by a software routine, driven by the interrupts of the period register CCR0.

A possible driver circuit for a pulldown output is shown on the upper left-hand corner in Figure 4–76: the additional npn transistor lowers the output impedance for the positive supply voltage 12 V. In this way the supply voltage V_{CC2} (18 V), which is needed for the output voltage of the SN75372, is generated.

The two lower MOSFETs of the H-bridge are driven in a static manner from the MSP430 with 5-V signals. This is possible because the TPIC5424 is designed for logic drive signals (0 to 5 V).

As shown in Figure 4–76, the TPIC5424 has integrated all the necessary protection diodes on-chip, therefore, no external components are needed. The signals at the MOSFET gates are shown in Figure 4–76 in the lower right-hand corner.

An exceedingly high motor current is detected by the overcurrent detection circuit. If a fixed voltage level, according to a maximum current value, is exceeded (e.g. by a blocking of the motor or by a current flow through one of the H-bridge halves), the comparator output switches off the lower drivers T2 and T4 and the additionally requested P0.0 or NMI interrupt (highest priority) takes steps to switch off the output stages completely. The overcurrent detection can be realized with more than one level as shown in Figure 4–75. The motor temperature can be measured the same way as shown in Figure 4–75.

❑ Applicable for

■ Permanently excited dc motors

❑ Advantages

■ Few components necessary

■ Both directions of rotation possible
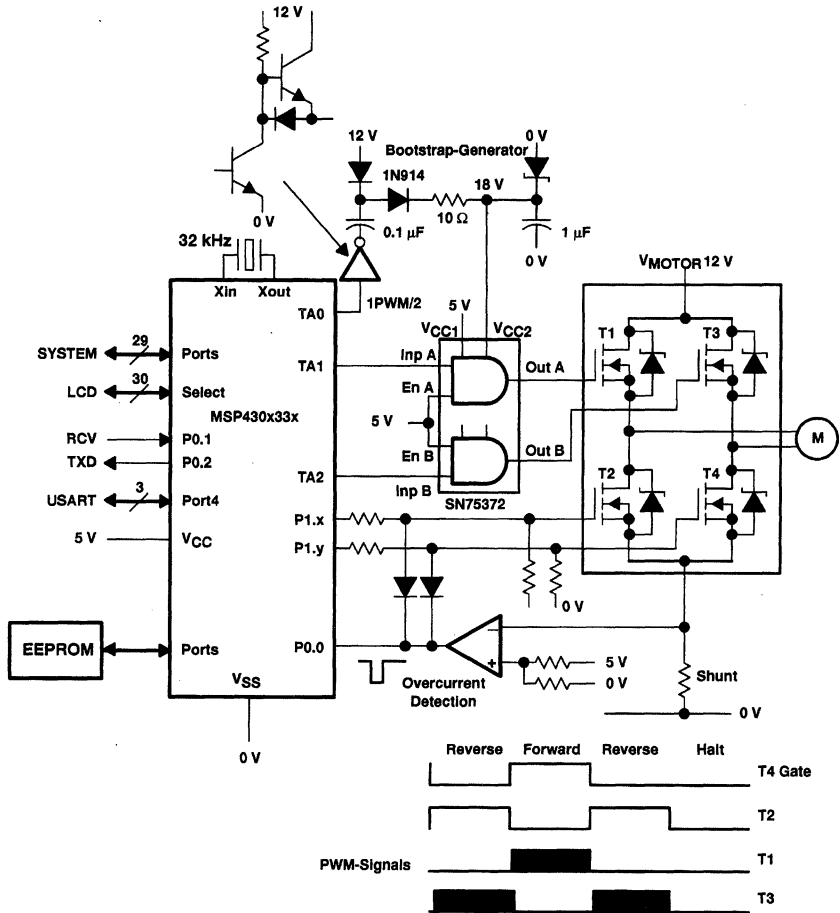
Figure 4–76. PWM Motor Control With a MOSFET H-Bridge

H-Bridge for a Brushless DC Motor

Figure 4–77 shows the control circuitry for a brushless dc motor. Complementary power MOSFETs are used for the output stages. The advantage of this solution is that no gate voltage above the motor voltage Vmotor is necessary. The H-bridge is switched over dependent on a position indicator working with

the Hall effect. If the change of polarity is necessary for the motor voltage, the Hall sensor requests interrupt and the MSP430 switches over the H-bridge.

The necessary delay times (ts in Figure–77), which prevent a short circuit in the H-bridge halves, are generated by software. If this happens, the over current detection switches off the two lower MOSFET drivers T2 and T4. The simultaneously requested interrupt at input P0.0 will force the software to switch-off all of the MOSFET drivers completely until the lost synchronism is built-up again.

The direction of rotation can be reversed by changing the current flow direction relatively to the location of the rotor. This is possible because the field is generated normally with a permanent magnet (see pulse diagram in Figure 4–77).

The circuitry shown does not need a tachometer because the signal of the Hall sensor can be used for the measurement of the number of revolutions per second.

The TLE2144 operational amplifiers are used as MOSFET drivers with their outputs and as AND gates with their inputs. The circuitry used with the eight equal resistors allows the two control outputs P0.4 and P0.5 to switch the PWM signal to the proper MOSFET transistors. The PWM-output TA1 is able to switch off both operational amplifier drivers and to switch on the driver prepared by P0.4 or P0.5. This solution leaves three compare/capture latches for other purposes.

The type of control (only one-half of the bridge is switched by the PWM signal, the other half is switched on in a static manner) decreases the switching losses.

❑ Applicable for

   ■ Brushless dc motors

❑ Advantages

   ■ Both directions of rotation possible

   ■ Robust motors without brush wear usable

   ■ Control of the motor speed without expense (Hall sensor)
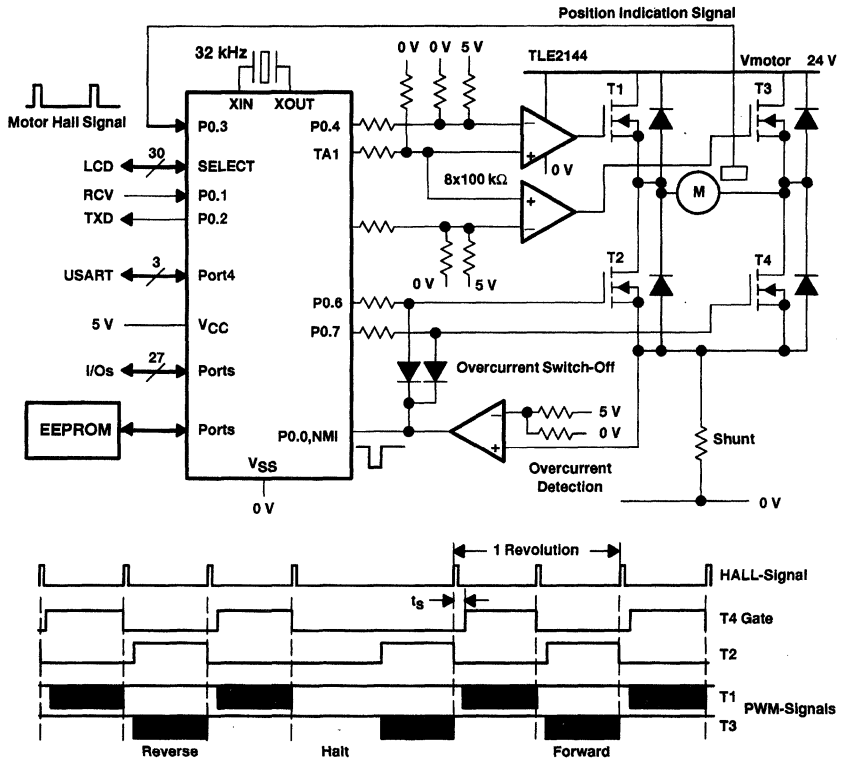
Figure 4–77. PWM Motor Control for Brushless DC Motor

The circuitry shown in Figure 4–77 can be used for other kinds of motors too. The driving signals for the H-bridge need to be changed in this case (see Figure 4–76) for a dc motor. The measurement of the motor temperature is possible the same way as shown in Figure 4–75.

H-Bridge With Integrated Output Stages

Figure 4–78 shows an integrated H-bridge motor controller made with an L293. Two H-bridges of this type are integrated in a single package. The rotation direction of the motor is controlled with the static output P1.1. The pulse width of the PWM output TA1 defines the effective output voltage for the motor.

If the rotation direction is changed then the PWM signal at output TA1 needs to be inverted. The output signal that represents the highest output voltage for

the forward direction translates to the lowest voltage for the reverse direction (see Figure 4–78). This inversion can be made by software (INV dst) or by the change of the drive mode of the output unit (see Section 6.3.5, *The Output Unit*).
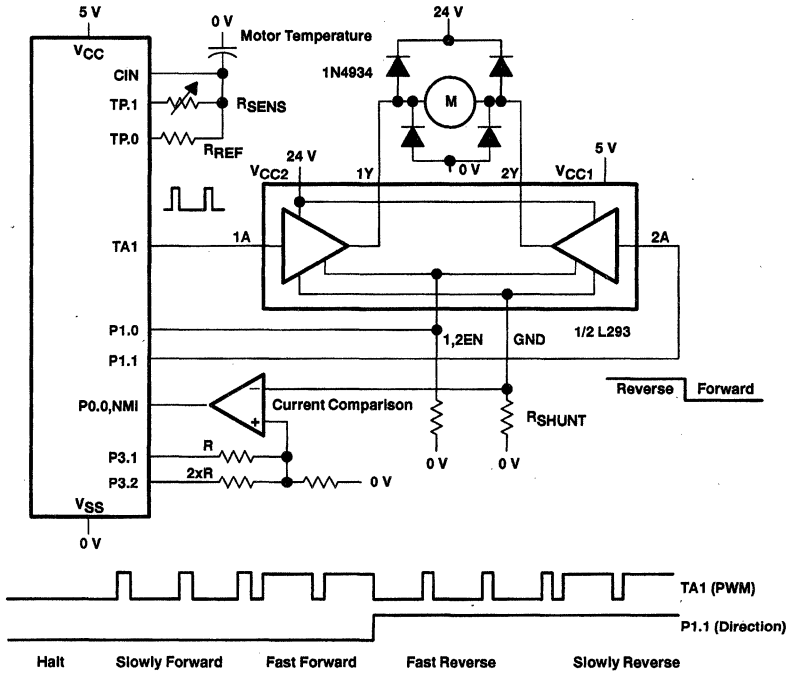
Figure 4–78. Integrated PWM Motor Control With Static Rotation Direction

The motor current is observed with a threshold detection circuit (analog comparator at input P0.0). The motor current can be compared to four analog thresholds. The resistor connected to output P1.0 ensures that the enable input of the L293 is switched off during the initialization of the MSP430. No current can flow through the motor during this time.

❏ Applicable for

■ Permanently excited dc motors

❏ Advantages

■ Change of rotation direction is included

■ Minimum hardware, single chip only

■ Built-in overheating protection

■ Full PWM resolution for both rotation directions

■ No generation of delay times necessary (included in the L293)
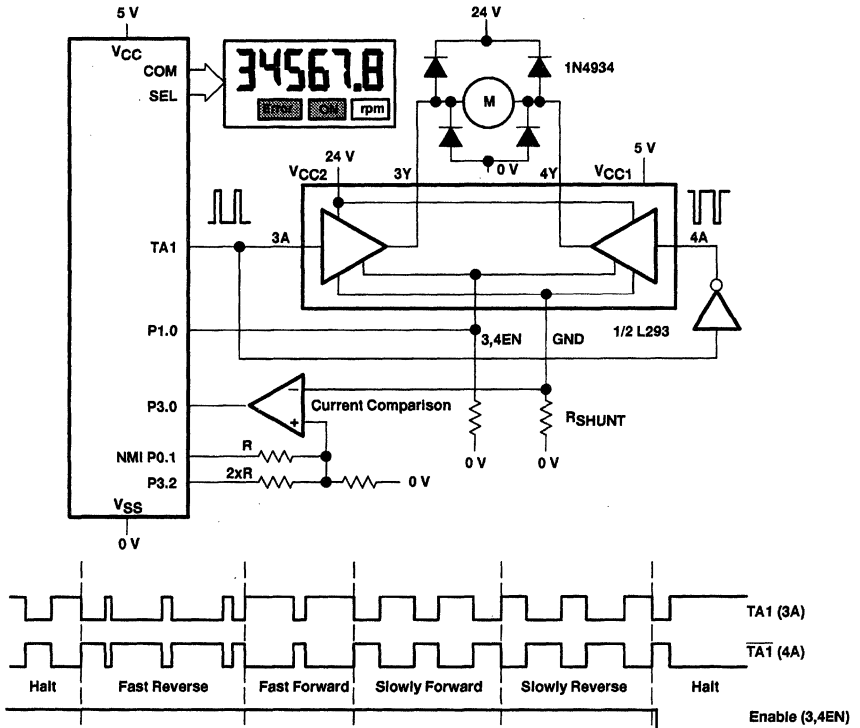
Figure 4–79. Integrated PWM Motor Control With Dynamic Rotation Direction

Figure 4–79 shows the L293 used with a dynamic definition of the rotation direction. Input 4A is always driven with the inverted signal of input 3A. The inverter at input 4A can be omitted if the inverted signal at 4A is generated by a second PWM output (e.g. TA2). The Capture/Compare Latch CCR2 portion is always loaded with the same value as CCR1, but the inverted output mode of the output unit 1 is used (e.g. set/reset instead of reset/set).

Motor standstill can be done in two ways (see the diagrams in Figure 4–79):

❑ With a PWM impulse ratio equal to one (see the left-hand side of the diagram)

❑ By switching the enable input of the L293 low (here P1.0, see the right-hand side of diagram)

With the same PWM output frequency, the dynamic control allows only half of the resolution when compared to the static control shown in Figure 4–78. One resolution bit is necessary for the sign of the direction of rotation.

❑ Applicable for

■ Permanently excited dc motors

❑ Advantages

■ Change of rotation direction is included

■ Minimum hardware, single chip only

■ Built-in overheating protection

■ Sliding transition possible for the change of the rotation direction

■ No generation of delay times necessary (included in the L293)

The two circuits shown in Figures 4–78 and 4–79 can be controlled together with a single MSP430C33x. Both figures can be integrated into one schematic with only slight modifications.

If a lower motor current is sufficient, the L293D can be used.

Both the L293 and the L293D feature built-in overheating protection that switches off the device during over temperature events.

H-Bridge for High Motor Voltages

Electric motors with voltages above 60 V need completely different driver concepts. The motor drive is most often made with IGTBs. The voltages and currents necessary for driving these semiconductors are delivered from special driver ICs. These ICs also contain the necessary safety circuits. An example for such a driver circuit is shown in Figure 4–80. The MSP430 defines the rotation direction with the PWM outputs TA1 and TA2. Only one of the PWM outputs is active, the other switches on the lower transistor of the other half of the bridge (static). This way the circuitry shown in Figure 4–80 is able to run the motor in both directions. As the supply voltage for the motor, the rectified ac voltage of 230 V is used.

The capture/compare register 4 works as a capture latch. It is used for the speed measurement of the motor (input TA4).
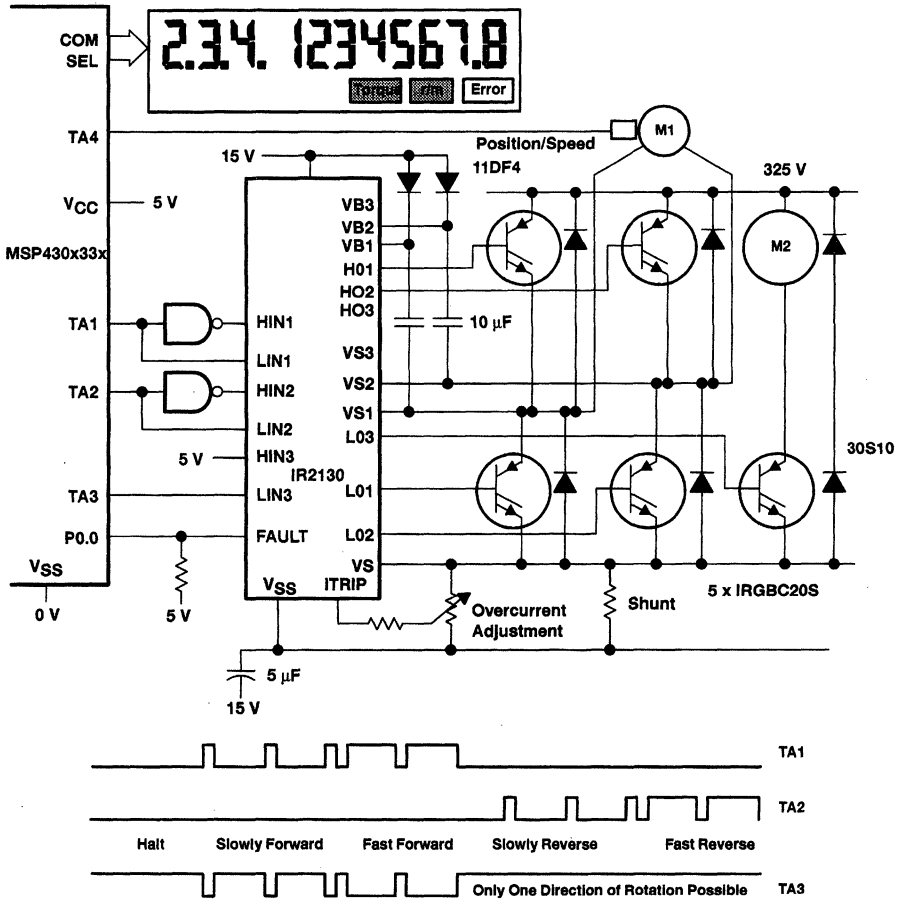
Figure 4–80. PWM Motors Control for High Motor Voltages

The MSP430 software does not need monitor any condition where both transistors of either half of the bridge are switched on simultaneously. Built-in delay times in the IR2130 prevent this state. In case of an overcurrent, the built-in overcurrent detection at the input Itrip switches off the IR2130 completely. An output fault indicates this state and an undervoltage at the Vcc terminal of the IR2130 with a low signal.

The gate voltage of the two upper power transistors, which must be higher than the motor voltage, is generated by the IR2130 with the help of a bootstrap generator. The output signals VS2 and VS1 drive this internal generator. Externally, only two diodes and two storage capacitors are needed. Static operation is not possible in this configuration. The generation of the gate voltage makes dynamic operation necessary. VS1 or VS2 must also be active during motor idle (lower bridge transistors off).

With the unused IR2130 output LO3, a second motor (M2) can be driven with the PWM TA3 output. For the change of rotation direction, a relay is needed. The circuitry for this is shown in Figure 4–74. Motor M2 is driven as desribed in Section 4.11.2.1, *Single Output Stages*. M2 is completely independant from M1.

❑ Applicable for

■ DC motors with direct ac circuit connection, universal motors

❑ Advantages

■ Both rotation directions are possible due to H-bridge

■ Direct ac circuit connection possible

■ Built-in delay times

■ Full PWM resolution for both rotation directions

■ High motor power possible

4.11.2.3 Three-Phase Motor Control

The MSP430C33x is also able to control 3-phase electric motors. This is due to the following hardware features:

❑ Three synchronized PWM outputs (Timer_A)

❑ Table processing capabilities (indirect, indirect with autoincrement, and indexed addressing modes)

❑ Hardware multiplier (16×16 bit) with immediate 32-bit result

❑ Up to 3.8-MHz CPU frequency; 263-ns execution time for single-cycle instructions (register/register mode)

These features together allow the generation of three PWM output signals that are phase shifted 120° relative to the others. The repetition rate should be near 20 kHz to prevent it from being heard. With a PWM frequency of 16 kHz, nearly 8-bit resolution is possible. The system works as follows:

❑ The repetition rate of the PWM pulses is defined by the content of the period register CCR0. This value is always the same.

❑ The pulse width for the three PWM signals is defined by registers CCR1 – CCR3. Each CCR controls one phase.

❑ The nominal pulse widths for the generation of a sine curve are contained in a byte table (nominal 100% values). Dependent on the angle counter, the actual values are read out individually for each phase by indexed addressing.

❑ The frequency of the motor voltage is defined by the modification frequency of the angle counter. The hardware multiplier is used here for the necessary calculations.

❑ The motor voltage is defined by the modified pulse width (table value multiplied by the percentage of the voltage). The hardware multiplier is used also for this task

The complete hardware diagram is shown in Figure 4–82. The interface to the motor is made by an IR2130 motor controller. This chip includes the necessary safety functions for overcurrent detection and generation of the necessary dead times for the output transistors.

The formula for the timer value n_{CCRx} is:

$$Vm = \left(\frac{n_{CCRx}}{n_{CCR0}} - 0.5\right) \times V_{motor} \rightarrow n_{CCRx} = \left(\frac{Vm}{V_{motor}} + 0.5\right) \times n_{CCR0}$$

Where:

Vm	Mean voltage at the motor phases (−Vmotor/2 to +Vmotor/2)	[V]
Vmotor	Voltage of the motor power supply	[V]
n_{CCRx}	Content of Compare Register x	
n_{CCR0}	Content of Compare Register 0 (Period Register)	

If the up mode of the Timer_A is used, then n_{CCR0} must be substituted by $n_{CCR0}+1$.

❑ Applicable for

■ Three-phase motors like induction motors

■ Open loop control method

■ Voltage/frequency method

Figure 4–81 shows some PWM outputs for different phase voltages.
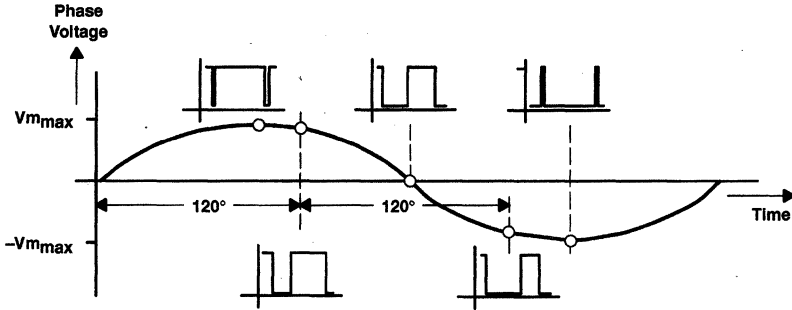
Figure 4–81. PWM Outputs for Different Phase Voltages

Note that zero volt for a motor phase is generated by a pulse width of 0.5 relative to the period.
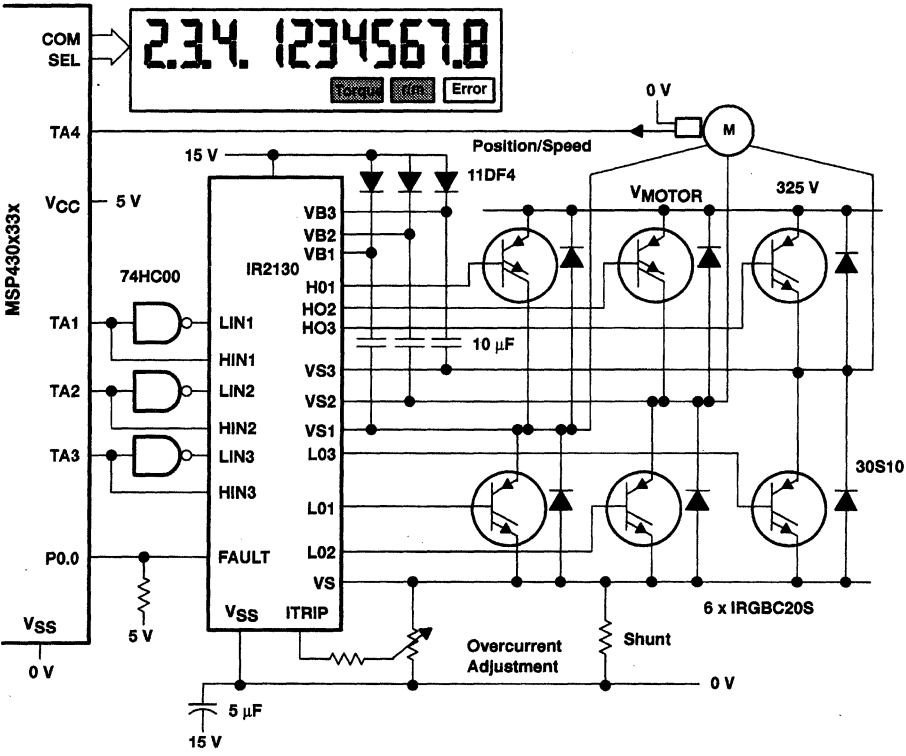
Figure 4–82. PWM Motors Control for High Motor Voltages

4.11.2.4 Low-Frequency Pulse Width Modulation ·

The PWM examples demonstrated in the circuits of this section are primarily thought for high repetition rates (16 kHz and more) but a lot of motor control applications do not need these high repetition rates. For these applications the same hardware proposals can be used with an output controlled by the Universal Timer/Port Module. If fed by the ACLK (32 kHz), it allows for example the following combinations:

❑   128-Hz repetition rate with a resolution of 256 steps or
❑   256-Hz repetition rate with a resolution of 128 steps
❑   512-Hz repetition rate with a resolution of 64 steps

Other combinations are possible too. If the MCLK is used as input frequency then the repetition rates and the resolution can be even higher, but the interrupt latency time plays an increasing role due to the software-based structure of this timer module: the PWM output is controlled by an interrupt handler and not by a hardware module as with the Timer_A. The characteristics of this kind of control are very similar to the TRIAC control due to the low repetition rates.

This way of motor control can substitute the PWM-control solutions realized with relays, as it is implemented in some automotive applications.

More details of the PWM generation are described in Section 3.6.4, *PWM DAC With the Universal Timer/Port Module*.

❑ Applicable for

■ All motors controllable by TRIACs

■ DC motors

4.11.2.5 Bandwidth of the MSP430 Solutions for PWM Control

Figure 4–83 shows the bandwidth of solutions the MSP430 family offers for PWM control systems; starting from a minimum system with a MSP430C312 up to a maximum system using a MSP430C337.

The minimum system with the MSP430C312 gets its information concerning the motor control (reference speed, direction of rotation, on/off) normally from a host via the I/O terminals or the SW/HW UART (RS232 link). It allows relatively slow PWM frequencies ($\approx$1 kHz).

The maximum system with the MSP430C337 is shown in the following. Its capabilities allow complete system control, not just the motor handling.

The PWM control for a single-phase motor normally does not use 100% of the MSP430 CPU. This being known, many functions of the host computer can be taken over by the MSP430 (e.g., when used in a tumbler or a dish washer controller). This is especially true for versions of the MSP430 having large memories and many I/O lines.
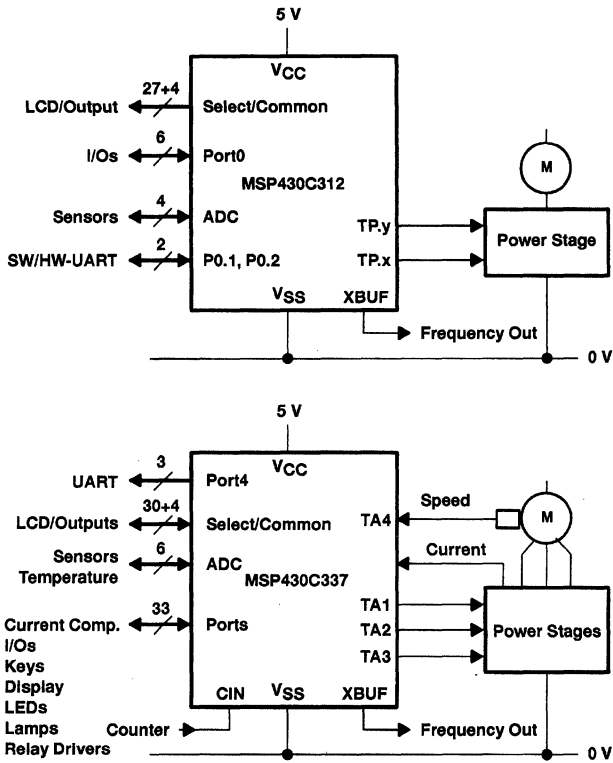
Figure 4–83. Minimum System and Maximum System Using the MSP430 Family

In Figure 4–83 all components not absolutely needed are omitted.

The hardware proposals shown are not only usable for the motor type named in the text, but also for other motor types when the needed hardware changes are made (e.g. the adding of a Hall sensor (position indication sensor) for a brushless dc motor).

If needed the application shown can be completed with one or more of the following features:

❏ Temperature sensors for the measurement of the motor temperature(s)

❏ Temperature sensors for the driver IC

❑ Tachometer for the measurement of the motor speed or the rotor position

❑ Inputs for light sensors (safety, movement, flame observation etc.)

❑ Analog inputs for the measurement of the motor voltage (improvement of control)

❑ Connections to a host via the USART (SPI or SCI), HW/SW–UART or ports

❑ Some of the possibilities shown in Figure 4–83 (keys, LEDs, relays, LCD etc)

Caused by the numerous peripherals of the MSP430 family, all of these previous functions can be implemented easily and cheaply.

4.11.3 Digital Motor Control With TRIACs

With the help of a TRIAC (TRiode for ac) the following electric motor types can be controlled:

❑ Universal motors

❑ DC motors (connected via a bridge rectifier. See Figure 4–84)

❑ Capacitor motors

❑ Single-phase asynchronous motors

❑ Single-phase synchronous motors

The timing for the TRIAC control is possible with the Universal Timer/Port Module and the Timer_A. Both can deliver the timing in the range from 0.5 ms to 20 ms with the needed resolution.

4.11.3.1 Motor Connection and Control

The electric motor can be connected to ac directly or via a bridge rectifier. Both possibilities are shown in Figure 4–84. It is possible to control ac motors as well as dc motors with a TRIAC.
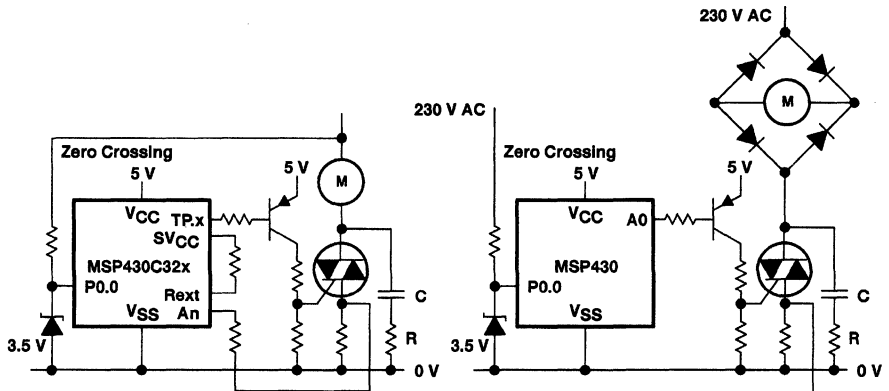
Figure 4–84. TRIAC Control for AC Motors and DC Motors

The RC combination switched in parallel to the TRIAC (see Figure 4–84) prevents the turn-on of the TRIAC in case the voltage changes too fast (large dv/dt). Switching ac transients, therefore, do not cause errors. Otherwise, this RC combination greatly reduces switching noise (EMV).

With the circuitry of the left hand side of Figure 4–84, the current through the TRIAC can be measured in the positive and negative direction. The current source of the MSP430 shifts the signed input voltage of the TRIAC current into the unsigned range of the 14-bit ADC. The zero point of the ADC can be calibrated during periods without TRIAC current.

4.11.3.2 TRIAC Control

A TRIAC normally cannot be controlled directly from a microcomputer. Two factors cause this:

❑ A normal microcomputer output cannot provide the necessary current for the TRIAC gate. The gate current is near 100 mA.

❑ During the triggering of the TRIAC, the TRIAC gate generates a voltage that can pull the microcomputer output above or below the supply voltages, Vcc with respect to Vss. This can lead to destruction of the output, to latch-up, or to a hang up of the software.

Both of the previous mentioned disadvantages are eliminated when a simple transistor stage is added between the microcomputer output and the TRIAC gate.

❑ The current amplification of the transistor provides the necessary gate current using the limited output current of the microcomputer

❑ The voltage range of the transistor collector withstands even strong voltage peaks generated by the TRIAC gate.

Depending on whether a negative or positive gate current is used, an npn- or a pnp-transistor is used. Both possibilities are shown in Figure 4–85. The superior circuit arrangement depends on the gate characteristics of the TRIAC used. The gate current needed is normally lower if a negative-going gate trigger pulse is used.
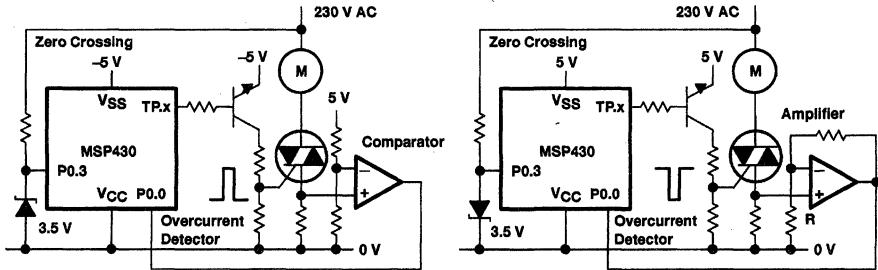
Figure 4–85. Positive and Negative TRIAC Gate Control

The TRIAC gate can be controlled in a static or in a dynamic manner.

❑ Static Gate Control: a long gate pulse switches on the TRIAC safely. The disadvantage of this method is the high gate current that is needed.

❑ Dynamic Gate Control: a sequence of short pulses (duration approximately 10 μs) switches on the TRIAC. If the first pulse does not have enough energy, one of the following pulses switches the TRIAC on safely. This method needs little energy and lessings the load on the power supply. One of the MSP430 timers can be left running in PWM mode or the setting/resetting by software can also do this job.
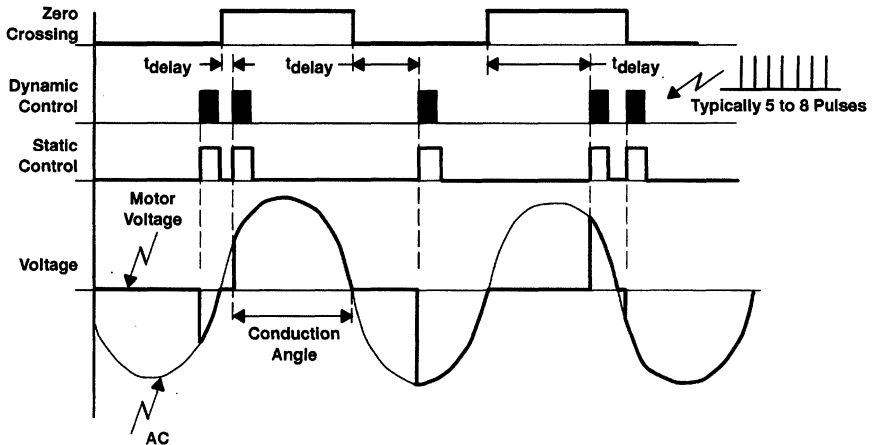
Figure 4–86. Static and Dynamic TRIAC Gate Control

The time t_{delay} in Figure 4–86 represents the time delay measured from the zero crossing of the ac voltage to the triggering of the TRIAC. The conduction angle is defined in this way.

The sequence of software steps is different for the Timer_A and the Universal Timer/Port Module.

Universal Timer/Port Module

☐ The time t_{delay} is calculated by the MSP430 software depending on the control algorithm

☐ The negated number of cycles (MCLK or ACLK) corresponding to the result t_{delay} is loaded into the counter registers TPCNT1 and TPCNT2 after the zero crossing of the ac voltage

☐ The timer requests an interrupt after the elapsed time t_{delay} (TPCNT2 overflows).

☐ The called interrupt handler finally triggers the TRIAC, which switches the ac voltage to the load.

Timer_A (Continuous Mode)

❑ The time t_{delay} is calculated by the MSP430 software depending on the control algorithm

❑ The number of cycles (MCLK or ACLK) corresponding to the result t_{delay} is added to one of the compare registers CCRx after the zero crossing of the ac voltage

❑ The output unit x is programmed to the mode that outputs the desired trigger pulse

❑ The CCRx requests an interrupt after the elapsed time t_{delay} (CCRx equals the timer register).

❑ The output unit x triggers the TRIAC, which switches the ac voltage to the load.

❑ If dynamic gate control is used, the called interrupt handler outputs several trigger pulses by software or by using the PWM capability of Timer_A.

4.11.3.3 Control Algorithms

The value to be controlled (speed/velocity, current consumption, or torque) is influenced with the conduction angle of the TRIAC. This conduction angle (see Figure 4–86) is defined by t_{delay}, the time the TRIAC triggering is delayed with reference to the zero crossing of the ac voltage.

For the control algorithms (that need to run in real time) two different methods are used:

❑ Normal calculation of the algorithm: For this method, the MSP430c33x is well suited very because of its hardware multiplier on-chip. This allows a very-high calculation speed. (10 to 20 times higher than possible with an 8-bit CPU.

❑ Use of tables (especially with very high control speeds): For this method, the MSP430 is also very well suited because of its addressing modes, indirect, indirect autoincrement, and indexed, allow for a very simple and fast access to table values.

With TRIAC controls, normally everything necessary for the controlling is calculated directly. For dc machines, PID control is possible without the use of characteristics because of their linear behavior.

Exception: With asynchronous machines most often a voltage/frequency or a current/frequency characteristic method is used that uses description tables. These are located in the ROM (normalized form) or in an external memory.

4.11.3.4 Cost Reduction

To lower the cost of the complete system, two possibilities exist. They are described in the following two sections.

Nonregulated Voltage for the TRIAC Control

To minimize the cost for the power supply, it is possible to split the parts for the supply of the MSP430 and for the TRIAC control. The TRIAC control does not need a regulated voltage supply, so this voltage can be supplied directly from the charge capacitor Cch. Figure 4–87 illustrates this method. This solution has another advantage; the two supplies are separated completely. The power part interferes with the control part very little. The npn transistor can be replaced by an unused driver on the board.
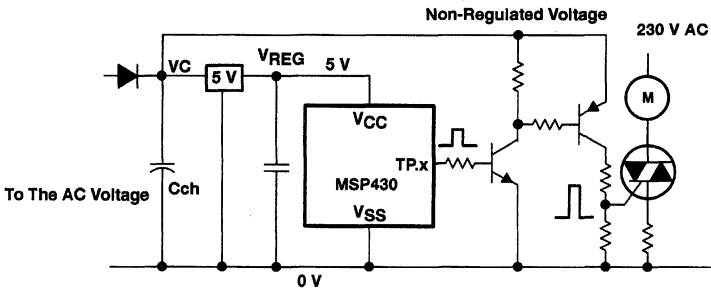
Figure 4–87. Nonregulated Voltage for the TRIAC Control

Use Without a Crystal

Despite the relatively low cost of a 32-kHz crystal, it can be advantageous to leave this component out. The TRIAC control requires the measurement of the ac frequency. This is required to know the exact time of the zero crossing of the ac voltage. If no crystal is used, the DCO frequency can be controlled by the measurement of a full ac period with one of the MSP430 timers. The formula for the calculation of the MCLK frequency fMCLK out of the timer value n and the ac frequency fac is:

$$f_{MCLK} = n \times k \times f_{ac}$$

Where:

fMCLK   Output frequency of the DCO   [Hz]

fac   AC frequency   [Hz]

n   Measurement result in the timer register

k   Predivider constant of the timer used (1, 2, 4, 8)

The measured DCO frequency fMCLK can be adjusted to the desired value by taking measurements in regular time intervals. The calculated value of fMCLK is used as a time base for the TRIAC triggering. More details are given in Section 6.3.8.7, *MSP430 Operation Without Crystal*.

4.11.3.5 Bandwidth of the MSP430 Solutions for TRIAC Control

Figure 4–88 shows the available bandwidth the MSP430 family offers. Starting from a minimum system with an MSP430C312 up to a maximum system using the MSP430C337 a lot of solutions are possible.

For the application of the MSP430 for a TRIAC motor control the same considerations are valid as made before for the PWM applications.

It is possible with an MSP430 to control more than one electric motor. The second motor can also be controlled as shown with a TRIAC—then the TRIAC control circuit is simply doubled—or the TAx output of the MSP430C33x is used for the PWM control of the second motor.
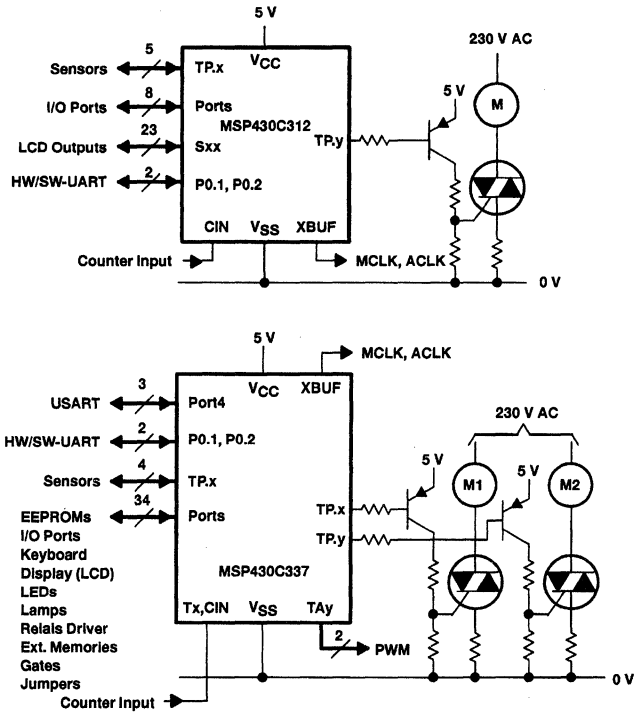
Figure 4–88. Minimum System and Maximum System With the MSP430 Family

In Figure 4–88, all circuitry not needed to demonstrate the application is omitted.

4.11.4 Motor Measurements

The methods shown for the measurement of the needed values like temperature, speed/velocity, etc are valid for PWM and TRIAC controls.

4.11.4.1 Overcurrent Detection

Many applications make it necessary to detect increased motor current and to start provisions when this occurs. An example for this is the blocking of a motor.

Independent, if the high current consumption is detected by a threshold comparison or by a current measurement In any case, the software has to take

steps against the overcurrent with the switch-off of the motor and the preventing of further gate triggering or by switching off the PWM output.

Threshold Detection

MSP430 family members that do not have an ADC on-chip must simplify the overcurrent detection to the detection of a passed-over threshold value. A simple operational amplifier is used, it compares the voltage generated by the motor current over a shunt with the calculated threshold. If this fixed threshold is reached, an interrupt is requested. Figure 4–89 shows this method of overcurrent detection on its upper side. The threshold itself is defined by the two resistors at the inverting input of the operational amplifier. If the voltage at the shunt resistor gets higher than this threshold, the positive edge of the operational amplifier output generates an interrupt signal.

If one threshold is not sufficient because the motor current needs to be better defined, a variable threshold, as shown in Figure 48–9 on the lower side, can be used. The MSP430 defines the desired threshold by the switching of the resistors 2R and 4R. If the outputs use the high, the low, and the high-impedance states, then 9 different thresholds are possible with this circuit.

The NMI input (non-maskable interrupt) can be used also for the overcurrent detection. No disabling is possible and the fastest response is assured.
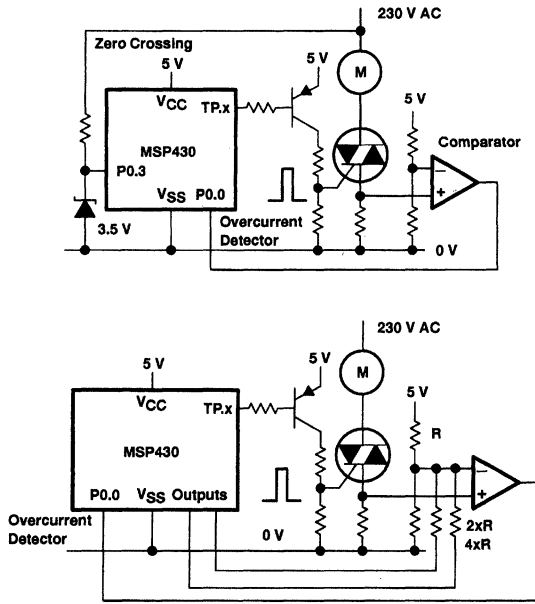
Figure 4–89. Overcurrent Detection With Single and Multiple Thresholds

Current Measurement

MSP430 family members having an on-chip ADC (MSP430C32x), can measure the current of both half-waves of the motor current. This allows a much better judgment of the behavior of the motor system than is possible with a simple threshold comparison. The voltage at the shunt, which is proportional to the motor current, is shifted into the range of the ADC (AVss to SVcc) with the voltage drop of Ics at Rv. Ics is the output current of the MSP430 current source. The voltage at the shunt is measured with one of the ADC inputs A0 to A5. The resolution at these analog inputs is 305 µV for a supply voltage of 5 V. If this is not sufficient, a simple amplifier is used. The zero point can be measured during periods with zero current. Figure 4–90 shows the measurement of the motor current.
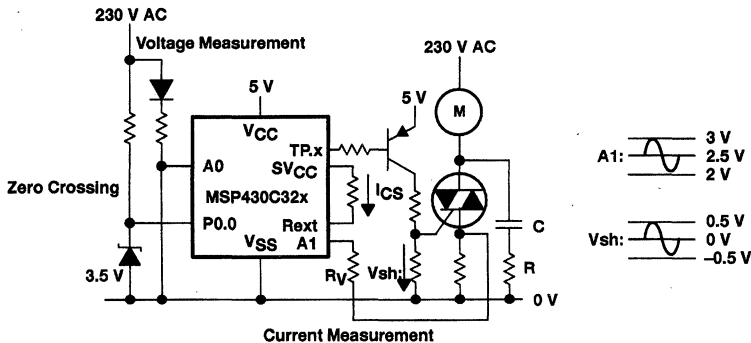
Figure 4–90. Motor Voltage Measurement and Current Measurement

4.11.4.2 Voltage Measurement

Figure 4–90 shows, at the left-hand side, how to measure the ac voltage (or another voltage) if needed. The diode prevents a negative voltage at the analog input A0. In this way, only the positive half wave can be measured. If both half waves are needed, the same way as shown for the motor current path and can be used. The voltage drop of Ics at resistor Rv shifts the signed input voltage into the range of the ADC.

4.11.4.3 Zero Crossing Detection

The detection of the zero crossing time of the ac voltage is very important with the TRIAC control because the zero crossing time represents the reference point for the phase control. The absolutely accurate zero crossing time is not necessary to get because in any case a certain minimum voltage must be reached at the TRIAC to hold it in the on-state. Figure 4–91 shows a simple circuit for this purpose. Via a resistor with a high resistance, the ac is connected to an interrupt input of the MSP430. This interrupt input is protected by a Zener diode (3.5 V), which protects against positive or negative overvoltages. The two edges of the square wave input signal give a very good indication for the positive and the negative zero crossing of the ac voltage. The time error of the zero crossing is due to this circuit arrangement and is approximately 60 µs.
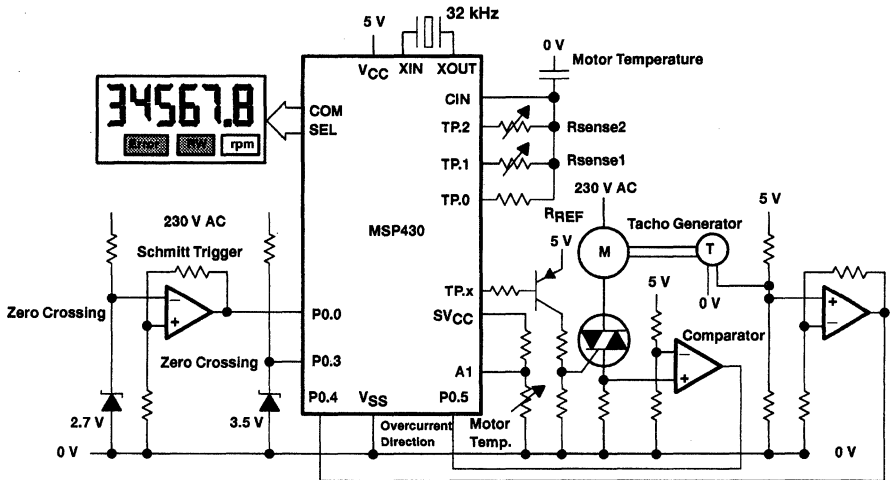
Figure 4–91. Support Functions for the TRIAC Control

A second possibility for the detection of the zero crossing is shown on the left-hand side of Figure 4–91. In case of a heavy disturbed ac voltage, the operational amplifier used as a Schmitt trigger gives an undisturbed zero crossing signal.

4.11.4.4 Measurement of the Motor Speed

If the control of an electric motor's speed is desired, a tachometer or something similar is necessary at the motor's shaft. The output signal of this tachometer is connected directly to an interrupt input of the MSP430 or is amplified with a simple operational amplifier when the output signal is too low. The second method is shown in Figure 4–91. With the capture latches the Timer_A provides, very precise timing measurements are possible.

4.11.4.5 Supervision of the Motor Temperature

To avoid the overheating of the motor, a temperature sensor (e.g. an NTC sensor) can be connected to MSP430 family members that have an ADC on-chip. In Figure 4–91, this possibility is shown for the analog input A1. Other MSP430 members can use the Universal Timer/Port Module as an ADC (see Figure 4–91). The software has to take steps when a high temperature is detected (e.g. turn-off of the motor, turn-on of an error indication, and other things).

4.11.4.6 Change of the Rotation Direction

In Figure 4–92, it is shown how the rotation direction can be changed for a universal motor (single-phase series commutator motor). The field winding is changed with a relay having two change over contacts. The same way the motor winding can be changed over.
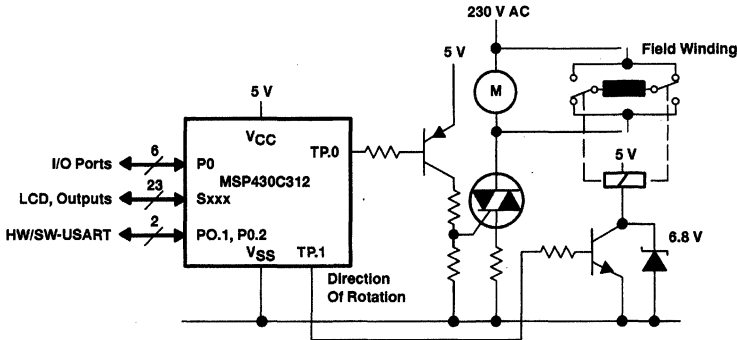
Figure 4–92. Change of the Direction of Rotation for a Universal Motor

4.11.5 Conclusion

The application examples shown for the MSP430 family demonstrate the excellent suitability of this microcontroller for the digital control of electric motors. This is true for PWM control as well as for TRIAC control. The numerous on-chip hardware modules like an ADC, I/O ports, and other helpful peripherals also ease the task. The total software compatibility of the MSP430 family members allows its use in software development. Table 4–15 gives an overview of the capabilities of the MSP430 sub-families:

Table 4–15. Capabilities of the MSP430 Sub-Families

CAPABILITY	MSP430x31x	MSP430x32x	MSP430x33x
20kHz PWM Control	No	No	Yes
Slow PWM Control (< 1kHz)	Yes	Yes	Yes
TRIAC Control	Yes	Yes	Yes
Single Phase PWM Motor Control	Yes	Yes	Yes
Three Phase PWM Motor Control	No	No	Yes
Voltage/Current Measurement	No	Yes	No
Voltage/Current Comparison	Yes	Yes	Yes
Temperature Measurement	Yes	Yes	Yes
Speed Measurement	Yes	Yes	Yes

Software Applications

5.1 Integer Calculation Subroutines

Integer routines have important advantages compared to all other calculation subroutines:

❑   Speed: Highest speed is possible especially when no loops are used

❑   ROM space: Least amount of ROM space is needed for these subroutines

❑   Adaptability: With the following definitions it is very easy to adapt the sub-routines to the actual needs. The necessary calculation registers can be located in the RAM or in registers.

The following definitions are valid for all of the following integer subroutines. They can be changed as needed.

```
; Integer Subroutines Definitions: Software Multiply
;
IRBT     .EQU    R9              ; Bit test register MPY
IROP1    .EQU    R4              ; First operand
IROP2L   .EQU    R5              ; Second operand low word
IROP2M   .EQU    R6              ; Second operand high word
IRACL    .EQU    R7              ; Result low word
IRACM    .EQU    R8              ; Result high word
;
; Hardware Multiplier
;
ResLo    .EQU    013Ah           ; HW_MPYer: Result reg. LSBs
ResHi    .EQU    013Ch           ; Result register MSBs
SumExt   .EQU    013Eh           ; Sum Ext. Register
```

All multiplication subroutines shown in the following section permit two different modes:

❑   The normal multiplication: the result of the multiplication is placed into the result registers

❑   The multiplication and accumulation function (MAC): the result of the multiplication is added to the previous content of the result registers.

5.1.1   Unsigned Multiplication 16 x 16-Bits

The following subroutine performs an unsigned 16 x 16-bit multiplication (label MPYU) or multiplication and accumulation (label MACU). The multiplication

subroutine clears the result registers IRACL and IRACM before the start. The MACU subroutine adds the result of the multiplication to the contents of the result registers.

The multiplication loop starting at label MACU is the same one as the one used for the signed multiplication. This allows the use of this subroutine for signed and unsigned multiplication if both are needed. The registers used are shown in the Figure 5–1:

```
        15                      0
        ┌──────────────────────┐
        │         R9 IRBT      │    Bit Test Register
        └──────────────────────┘
                    │
                    ▼
        ┌──────────────────────┐
        │         R4 IROP1     │    Multiplicand
        └──────────────────────┘

┌──────────────────┬──────────────────┐
│    R6 IROP2M     │    R5 IROP2L     │    Multiplier
└──────────────────┴──────────────────┘

┌──────────────────┬──────────────────┐
│     R8 IRACM     │     R7 IRACL     │    Accumulated Result
└──────────────────┴──────────────────┘
```
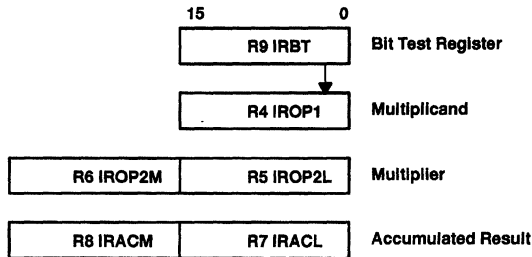
Figure 5–1. 16 x 16 Bit Multiplication – Register Use

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:

; TASK           MACU           MPYU      EXAMPLE

;-----------------------------------------------------------------

; MINIMUM        132            134       00000h x 00000h = 000000000h

; MEDIUM         148            150       0A5A5h x 05A5Ah = 03A763E02h

; MAXIMUM        164            166       0FFFFh x 0FFFFh = 0FFFE0001h

; UNSIGNED MULTIPLY SUBROUTINE: IROP1 x IROP2L -> IRACM/IRACL

;

; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM, IRBT

;

MPYU     CLR     IRACL                    ; 0 -> LSBs RESULT

         CLR     IRACM                    ; 0 -> MSBs RESULT

; UNSIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:

; (IROP1 x IROP2L) + IRACM|IRACL -> IRACM|IRACL

;

MACU     CLR     IROP2M                   ; MSBs MULTIPLIER

         MOV     #1,IRBT                  ; BIT TEST REGISTER

L$002    BIT     IRBT,IROP1               ; TEST ACTUAL BIT
```

```
        JZ      L$01                        ; IF 0: DO NOTHING
        ADD     IROP2L,IRACL                ; IF 1: ADD MULTIPLIER TO RESULT
        ADDC    IROP2M,IRACM
L$01    RLA     IROP2L                      ; MULTIPLIER x 2
        RLC     IROP2M                      ;
;
        RLA     IRBT                        ; NEXT BIT TO TEST
        JNC     L$002                       ; IF BIT IN CARRY: FINISHED
        RET
```

If the hardware multiplier is implemented then the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. The SumExt Register contains the carry after the MAC instruction; 0 (no carry) or 1 (carry occurred).

```
; Macro Definition for the unsigned multiplication 16 x 16 bits
;
MPYU    .MACRO  arg1,arg2                   ; Unsigned MPY 16x16
        MOV     arg1,&0130h
        MOV     arg2,&0138h
        .ENDM                               ; Result in ResHi|ResLo
;
; Multiply the contents of two registers
;
        MPYU    IROP1,IROP2L                ; CALL the MPYU macro
        MOV     ResLo,R6                    ; Fetch LSBs of result
        MOV     ResHi,R7                    ; Fetch MSBs of result
        ...
;
; Multiply the operands located in a table, R6 points to
;
        MOV     #ResLo,R5                   ; Pointer to LSBs of result
        MPYU    @R6+,@R6                    ; CALL the MPYU macro
        NOP                                 ; NOP: allow completion of MPYU
        MOV     @R5+,R7                     ; Fetch LSBs of result
```

```
           MOV      @R5,R8                   ; Fetch MSBs of result
   ;
   ; Macro Definition for the unsigned multiplication and
   ; accumulation 16 x 16 bits
   ;
MACU       .MACRO   arg1,arg2                ; Unsigned MAC 16x16
           MOV      arg1,&0134h              ; Carry in SumExt
           MOV      arg2,&0138h
           .ENDM                             ; Result in SumExt|ResHi|ResLo
   ;
   ; Multiply and accumulate the contents of two registers
   ;
           MPYU     R5,R6                    ; Initialize SumExt|ResHi|ResLo
           MACU     IROP1,IROP2L             ; Add IROP1 x IROP2 to result
           ADC      &SumExt,RAM              ; Add carry to RAM extension
   ;
```

5.1.1.1 Run Time Optimized Unsigned Multiplication 16 x 16-Bits

If the operands of the multiplication subroutine are shorter than 16 bits, the previous multiplication subroutine MPYU can be optimized during run time

The multiplication stops immediately after the operand IROP1 equals zero. This indicates that the operand with leading zeroes should be in IROP1. This run time optimized subroutine can be used instead of the normal subroutine. (The subroutine was developed by Leslie Mable/UK).

```
;
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; TASK            MACU           MPYU      IROP1     IROP2
;----------------------------------------------------------------
; MINIMUM          18             20      00000h x 00000h = 000000000h
; MEDIUM           90             92      000FFh x 0FFFFh = 000FEFF01h
; MAXIMUM          170           172      0FFFFh x 0FFFFh = 0FFFE0001h
; UNSIGNED MULTIPLY SUBROUTINE (Run time optimized):
; IROP1 x IROP2L -> IRACM|IRACL
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM
;
```

```
MPYU      CLR      IRACL                          ; 0 -> LSBs RESULT

          CLR      IRACM                          ; 0 -> MSBs RESULT
; UNSIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 x IROP2L) + IRACM|IRACL -> IRACM|IRACL
;
MACU      CLR      IROP2M                         ; MSBs MULTIPLIER
L$002     BIT      #1,IROP1                       ; TEST ACTUAL BIT (LSB)
          JZ       L$01                           ; IF 0: DO NOTHING
          ADD      IROP2L,IRACL                   ; IF 1: ADD MULTIPLIER TO RESULT
          ADDC     IROP2M,IRACM
L$01      RLA      IROP2L                         ; Double MULTIPLIER IROP2
          RLC      IROP2M                         ;
;
          RRC      IROP1                          ; Next bit of IROP1 to LSB
          JNZ      L$002                          ; If IROP1 = 0: finished
          RET
```

5.1.1.2 Fast Unsigned Square Function

For some applications, a fast square function is necessary. Two different solutions are given:

❑ For 16-bit unsigned numbers without rounding

❑ For 14-bit unsigned numbers with rounding. This version is adapted to the output of the ADC of the MSP430C32x family.

Both use table processing; an offset to a table containing the squared input numbers is built. The given cycles include the move of the operand into R5.

```
; Fast unsigned squaring for a 16 bit number. The upper 16 bits
; of the result are moved to R5. No rounding is used. 7 cycles
;
          MOV.B    DATA+1,R5                      ; MSBs to R5
          RLA      R5                             ; Number x 2 (word table address)
          MOV      SQTAB(R5),R5                   ; MSBs^2 to R5
          ...                                     ; Squared value in R5
;
; Fast unsigned squaring for a 14 bit number. The upper 16 bits of
; the result are added to a buffer SQSUM. Rounding is used.
```

```
; 18 cycles. If registers are used for the sum: 12 cycles
;              .            .
          MOV     &ADAT,R5                   ; ADC result to R5
          ADD     #80h,R5                    ; Round high byte
          SWPB    R5                         ; MSBs to LSBs
          RLA.B   R5                         ; Number x 2 (word table address)
          ADD     SQTAB(R5),SQSUM            ; Add MSBs^2 to SQSUM
          ADC     SQSUM+2                    ; Add carry
          ...                                ; Continue
;
; Table with squared values. Length may be adapted to the maximum
; possible input number.
;
SQTAB     .word   ($-SQTAB)*($-SQTAB)/4      ; 0 x 0 = 0
          .word   ($-SQTAB)*($-SQTAB)/4      ; 1 x 1 = 1
          .word   ($-SQTAB)*($-SQTAB)/4      ; 2 x 2 = 4
          ...
          .word   ($-SQTAB)*($-SQTAB)/4      ; 0FFh x 0FFh = 0FE01h
          .word   0FFFFh                     ; Max. for 0100h x 0100h
```

5.1.2 Signed Multiplication 16 x 16-Bits

The following subroutine performs a signed 16 x 16-bit multiplication (label MPYS) or multiplication and accumulation (label MACS). The multiplication subroutine clears the result registers IRACL and IRACM before the start. The MACS subroutine adds the result of the multiplication to the contents of the result registers. The register used is the same as with the unsigned multiplication. Therefore, Figure 5–1 is also valid.

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; TASK            MACS            MPYS        EXAMPLE
;---------------------------------------------------------------
; MINIMUM         138             140         00000h x 00000h = 000000000h
; MEDIUM          155             157         0A5A5h x 05A5Ah = 0E01C3E02h
; MAXIMUM         172             174         0FFFFh x 0FFFFh = 000000001h
; SIGNED MULTIPLY SUBROUTINE: IROP1 x IROP2L -> IRACM|IRACL
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM, IRBT
```

```
MPYS        CLR     IRACL                       ; 0 -> LSBs RESULT
            CLR     IRACM                       ; 0 -> MSBs RESULT
; SIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 x IROP2L) + IRACM|IRACL -> IRACM|IRACL
;
MACS        TST     IROP1                       ; MULTIPLICAND NEGATIVE ?
            JGE     L$001
            SUB     IROP2L,IRACM                ; YES, CORRECT RESULT REGISTER
L$001       TST     IROP2L                      ; MULTIPLIER NEGATIVE ?
            JGE     MACU
            SUB     IROP1,IRACM                 ; YES, CORRECT RESULT REGISTER
; THE REMAINING PART IS EQUAL TO THE UNSIGNED MULTIPLICATION
MACU        CLR     IROP2M                      ; MSBs MULTIPLIER
            MOV     #1,IRBT                     ; BIT TEST REGISTER
L$002       BIT     IRBT,IROP1                  ; TEST ACTUAL BIT
            JZ      L$01                        ; IF 0: DO NOTHING
            ADD     IROP2L,IRACL                ; IF 1: ADD MULTIPLIER TO RESULT
            ADDC    IROP2M,IRACM
L$01        RLA     IROP2L                      ; MULTIPLIER x 2
            RLC     IROP2M                      ;
;
            RLA     IRBT                        ; NEXT BIT TO TEST
            JNC     L$002                       ; IF BIT IN CARRY: FINISHED
            RET
```

If the hardware multiplier is implemented then the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, then a NOP is necessary after the MACRO call to allow the completion of the multiplication. The SumExt Register contains the sign of the result in ResHi and ResLo; 0000h (positive result) or 0FFFFh (negative result).

```
; Macro Definition for the signed multiplication 16 x 16 bits
;
MPYS        .MACRO  arg1,arg2                   ; Signed MPY 16x16
            MOV     arg1,&0132h
            MOV     arg2,&0138h
```

```
        .ENDM                              ; Result in SumExt|ResHi|ResLo
;
; Multiply the contents of two registers
;
        MPYS    IROP1,IROP2                ; CALL the MPYS macro
        MOV     &ResLo,R6                  ; Fetch LSBs of result
        MOV     &ResHi,R7                  ; Fetch MSBs of result
        MOV     &SumExt,R8                 ; Fetch Sign of result
;
; Multiply the operands located in a table, R6 points to
;
        MOV     #ResLo,R5                  ; Pointer to LSBs of result
        MPYS    @R6+,@R6                   ; CALL the MPYS macro
        NOP                                ; NOP: allow completion of MPYS
        MOV     @R5+,R7                    ; Fetch LSBs of result
        MOV     @R5+,R8                    ; Fetch MSBs of result
        MOV     @R5,R9                     ; Fetch sign of result
;
; Macro Definition for the signed multiplication and
; accumulation 16 x 16 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further RAM extensions (here shown for only one).
;
MACS    .MACRO  arg1,arg2                  ; Signed MAC 16x16
        MOV     arg1,&0132h                ; Signed MPY is used
        MOV     arg2,&0138h
        ADD     &ResLo,MAClo               ; Add LSBs to result
        ADDC    &ResHi,MACmid              ; Add MSBs to result
        ADDC    &SumExt,MAChi              ; Add SumExt to MSBs
        .ENDM                              ;
;
; Multiply and accumulate signed the contents of two tables
;
        MACS    2(R6),@R5+                 ; CALL the MACS macro
        ....                               ; Accumulation is yet made
```

5.1.2.1 Fast Signed Square Function

For some applications, a fast signed square function is necessary (e.g. if the RMS value of an input signal needs to be calculated). Two different solutions are given:

❑ For 16-bit signed numbers without rounding

❑ For 14-bit signed numbers with rounding. This version is adapted to the output of the ADC of the MSP430C32x family.

Both use table processing; an offset to a table containing the squared input numbers is built. The given cycles include the move of the operand into R5.

```
; Fast signed squaring for a 16 bit number. The upper 16 bits
; of the result are moved to R5. No rounding is used. 10-12 cycles
;
        MOV.B   DATA+1,R5           ; MSBs of number to R5
        TST.B   R5                  ; Check sign of input number
        JGE     L$1                 ; Positive sign
        INV.B   R5                  ; Negative sign:
        INC.B   R5                  ; Use absolute value
L$1     RLA     R5                  ; Number x 2 (word table address)
        MOV     SQTAB(R5),R5        ; MSBs^2 from table to R5
        ...                         ; Squared value in R5
;
; Squaring for a signed 14 bit value:
; Change the unsigned ADC value (0 to 3FFFh) to a signed value
; by the subtraction of the measured zero point of the system:
;
        MOV     &ADAT,R5            ; ADC result to R5
        SUB     VAL0,R5             ; Subtract measured 0-point
;
; Fast signed squaring for a 14 bit number. The upper 16 bits of
; the result are added to a buffer SQSUM. Rounding is used.
; If registers are used for the sum: 15-17 cycles
;
        RLA     R5                  ; One bit more resolution
        ADD     #80h,R5             ; Round to high byte
        BIC     #0FFh,R5            ; Delete lower byte
```

```
            JGE      L$1                      ; Sign?

            INV      R5                       ; Absolute value of ADC result

            INC      R5               .        ; Complement + increment

L$1         SWPB     R5                       ; MSBs to LSBs

            RLA.B    R5                       ; Number x 2 (word table address)

            ADD      SQTAB(R5),SQSUM          ; Add MSBs^2 to SQSUM

            ADC      SQSUM+2                  ; Add carry

            ...                               ; Continue

;

; Table with squared values. Length may be adapted to the maximum

; possible input number.

;

SQTAB       .word    ($-SQTAB)*($-SQTAB)/4     ; 0 x 0 = 0

            .word    ($-SQTAB)*($-SQTAB)/4     ; 1 x 1 = 1

            .word    ($-SQTAB)*($-SQTAB)/4     ; 2 x 2 = 4

            ...

            .word    ($-SQTAB)*($-SQTAB)/4     ; 07Fh x 07Fh

            .word    ($-SQTAB)*($-SQTAB)/4     ; 080h x 080h
```

The errors for a single squaring are in the range of 1%. But, if rounding is used and several squared inputs are summed-up, the resulting error gets much smaller. For example, if a sinusoidal input voltage is measured in distances of 15°, then an error of less than 0.24% results.

If the previous method is used for the measurement of RMS values, then for a decision, it usually is not necessary to calculate the square root out of the accumulated squared inputs. It is much faster to use the accumulated value itself .

5.1.3 Unsigned Multiplication 8 x 8-Bits

The following subroutine performs an unsigned 8 x 8-bit multiplication (label MPYU8) or multiplication and accumulation (label MACU8). The multiplication subroutine clears the result register IRACL before the start. The MACU subroutine adds the result of the multiplication to the contents of the result register. The upper bytes of IROP1 and IROP2L must be zero when the subroutine is

called. The MOV.B instruction used for the loading ensures these bits are cleared. The registers used are shown in the Figure 5–2:
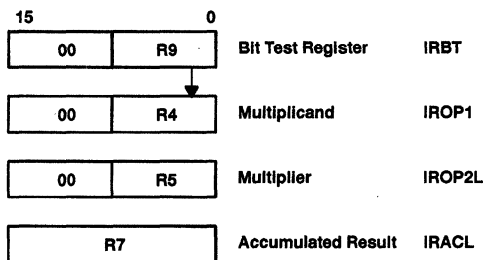
Figure 5–2. 8 x 8 Bit Multiplication – Register use

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; TASK            MACU8             MPYU8   EXAMPLE
;--------------------------------------------------------------
; MINIMUM         58                59      000h x 000h = 00000h
; MEDIUM          62                63      0A5h x 05Ah = 03A02h
; MAXIMUM         66                67      0FFh x 0FFh = 0FE01h
; UNSIGNED BYTE MULTIPLY SUBROUTINE: IROP1 x IROP2L -> IRACL
;
; USED REGISTERS IROP1, IROP2L, IRACL, IRBT
;
MPYU8   CLR    IRACL                       ; 0 -> RESULT
;
; UNSIGNED BYTE MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 x IROP2L) +IRACL -> IRACL
;
MACU8   MOV    #1,IRBT                      ; BIT TEST REGISTER
L$002   BIT    IRBT,IROP1                   ; TEST ACTUAL BIT
        JZ     L$01                         ; IF 0: DO NOTHING
        ADD    IROP2L,IRACL                 ; IF 1: ADD MULTIPLIER TO RESULT
L$01    RLA    IROP2L                       ; MULTIPLIER x 2
        RLA.B  IRBT                         ; NEXT BIT TO TEST
        JNC    L$002                        ; IF BIT IN CARRY: FINISHED
        RET
```

If the hardware multiplier is implemented, the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. If byte instructions are used for loading the multiplier registers, the high byte is cleared like a CPU register.

```
; Macro Definition for the unsigned multiplication 8 x 8 bits
;
MPYU8     .MACRO   arg1,arg2                ; Unsigned MPY 8x8
          MOV.B    arg1,&0130h              ; 00xx to 0130h
          MOV.B    arg2,&0138h              ; 00yy to 0138h
          .ENDM                             ; Result in ResLo. ResHi = 0
;
; Multiply the contents of two registers (low bytes)
;
          MPYU8    IROP1,IROP2L             ; CALL the MPYU8 macro
          MOV      &ResLo,R6                ; Fetch result (16 bits)
          . . .
;
; Macro Definition for the unsigned multiplication and
; accumulation 8 x 8 bits
;
MACU8     .MACRO   arg1,arg2                ; Unsigned MAC 8x8
          MOV.B    arg1,&0134h              ; 00xx
          MOV.B    arg2,&0138h              ; 00yy
          .ENDM                             ; Result in SumExt|ResHi|ResLo
;
; Multiply and accumulate the low bytes of two registers
;
          MACU8    IROP1,IROP2              ; CALL the MACU8 macro
```

5.1.4   Signed Multiplication 8 x 8-Bits

The following subroutine performs a signed 8 x 8-bit multiplication (label MPYS8) or multiplication and accumulation (label MACS8). The multiplication subroutine clears the result register IRACL before the start, the MACS8 subroutine adds the result of the multiplication to the contents of the result register.

The register usage is the same as with the unsigned 8 x 8 multiplication. Therefore, Figure 5–2 is also valid.

The part starting with label MACU8 is the same as used with the unsigned multiplication.

```
; EXECUTION TIMES FOR REGISTER CONTENTS (CYCLES) without CALL:
; TASK              MACS8           MPYS8     EXAMPLE
;--------------------------------------------------------------
; MINIMUM           64              65        000h x 000h = 00000h
; MEDIUM            75              76        0A5h x 05Ah = 0E002h
; MAXIMUM           86              87        0FFh x 0FFh = 00001h
; SIGNED BYTE MULTIPLY SUBROUTINE: IROP1 x IROP2L -> IRACL
;
; USED REGISTERS IROP1, IROP2L, IRACL, IRBT
;
MPYS8     CLR     IRACL                      ; 0 -> RESULT
;
; SIGNED BYTE MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 x IROP2L) +IRACL -> IRACL
;
MACS8     TST.B   IROP1                      ; MULTIPLICAND NEGATIVE ?
          JGE     L$101                      ; NO
          SWPB    IROP2L                     ; YES, CORRECT RESULT
          SUB     IROP2L,IRACL
          SWPB    IROP2L                     ; RESTORE MULTIPLICATOR
;
L$101     TST.B   IROP2L                     ; MULTIPLICATOR NEGATIVE ?
          JGE     MACU8
          SWPB    IROP1                      ; YES, CORRECT RESULT
          SUB     IROP1,IRACL
          SWPB    IROP1
;
; THE REMAINING PART IS THE UNSIGNED MULTIPLICATION
;
MACU8     MOV     #1,IRBT                    ; BIT TEST REGISTER
L$002     BIT     IRBT,IROP1                 ; TEST ACTUAL BIT
```

```
          JZ      L$01                  ; IF 0: DO NOTHING

          ADD     IROP2L,IRACL          ; IF 1: ADD MULTIPLIER TO RESULT

L$01      RLA     IROP2L                ; MULTIPLIER x 2

          RLA.B   IRBT                  ; NEXT BIT TO TEST

          JNC     L$002                 ; IF BIT IN CARRY: FINISHED

          RET
```

If the hardware multiplier is implemented, the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. If byte instructions are used for loading the multiplier registers, the high byte is cleared like a CPU register.

```
; Macro Definition for the signed multiplication 8 x 8 bits
;
MPYS8     .MACRO  arg1,arg2             ; Signed MPY 8x8

          MOV.B   arg1,&0132h           ; 00xx

          SXT     &0132h                ; Extend sign: 00xx or FFxx

          MOV.B   arg2,&0138h           ; 00yy

          SXT     &0138h                ; Extend sign: 00yy or FFyy

          .ENDM                         ; Result in SumExt|ResHi|ResLo
;
; Multiply the contents of two registers signed (low bytes)
;
          MPYS8   IROP1,IROP2           ; CALL the MPYS8 macro

          MOV     &ResLo,R6             ; Fetch result (16 bits)

          MOV     &ResHi,R7             ; Only sign: 0000 or FFFF
;
; Macro Definition for the signed multiplication and
; accumulation 8 x 8 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further RAM extensions
;
MACS8     .MACRO  arg1,arg2             ; Signed MAC 8x8

          MOV.B   arg1,&0132h           ; MPYS is used
```

```
        SXT     &0132h                          ; Extend sign: 00xx or FFxx

        MOV.B   arg2,&0138h                     ; 00yy

        SXT     &0138h                          ; Extend sign

        ADD     &ResLo,MAClo                    ; Accumulate LSBs 16 bits

        ADDC    &ResHi,MACmid

        ADDC    &SumExt,MAChi                   ; Add SumExt to MSBs

        .ENDM                                   ;
;
; Multiply and accumulate signed the contents of two byte tables
;
        MACS8   2(R6),@R5+                      ; CALL the MACS8 macro

        ....                                    ; Accumulation is yet made
```

5.1.5 Unsigned Division 32/16-Bits

The subroutine performs an unsigned 32-bit by 16-bit division. If the result does not fit into 16 bits, the carry is then set after return. If a valid result is obtained, the carry is reset after a return. The register usage is shown in Figure 5–3. The subroutine was developed by Mr. Leipold/L&G.
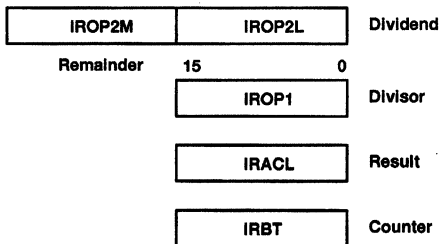
Figure 5–3. Unsigned Division – Register Use

```
; EXECUTION CYCLES FOR REGISTER CONTENTS (without CALL):
;DIVIDE          CYCLES              EXAMPLE
;----------------------------------------------------------------------------------
;               242                  0xxxxxxxxh : 00000h = 0FFFFh       C = 1
;               237                  03A763E02h : 05A5Ah = 0A5A5h       C = 0
;               240                  0FFFE0001h : 0FFFFh = 0FFFFh       C = 0
;
```

```
; USED REGISTERS IROP1, IROP2L, IRACL, IRBT, IROP2M
;
; UNSIGNED DIVISION SUBROUTINE 32-BIT BY 16-BIT
; IROP2M|IROP2L : IROP1 -> IRACL    REMAINDER IN IROP2M
; RETURN: CARRY = 0: OK    CARRY = 1: QUOTIENT > 16 BITS
;
DIVIDE   CLR     IRACL                    ; CLEAR RESULT
         MOV     #17,IRBT                 ; INITIALIZE LOOP COUNTER
DIV1     CMP     IROP1,IROP2M             ;
         JLO     DIV2
         SUB     IROP1,IROP2M
DIV2     RLC     IRACL
         JC      DIV4                     ; Error: result > 16 bits
         DEC     IRBT                     ; Decrement loop counter
         JZ      DIV3                     ; Is 0: terminate w/o error
         RLA     IROP2L
         RLC     IROP2M
         JNC     DIV1
         SUB     IROP1,IROP2M
         SETC
         JMP     DIV2
DIV3     CLRC                             ; No error, C = 0
DIV4     RET                              ; Error indication in C
```

A 32-bit divided by 32-bit numbers (XDIV) is given in the square root section.

5.1.6 Signed Division 32/16-Bits

The subroutine performs a signed 32-bit by 16-bit division. If the result does not fit into 16 bits, the carry is then set after a return. If a valid result is obtained, the carry is reset after a return. The register IRACM contains the extended sign (0000h or 0FFFFh) of the signed result in IRACL. The register usage is shown in the Figure 5–4:

```
 15
┌─┬──────────────┬──────────────┐
│S│   IROP2M     │   IROP2L     │  Dividend
└─┴──────────────┴──────────────┘
        Remainder  15          0
                   ┌──────────────┐
                   │    IROP1     │  Divisor
                   └──────────────┘

    ┌──────────────┬─┬──────────┐
    │    IRACM     │S│   IRACL  │     Result
    └──────────────┴─┴──────────┘
      Extended Sign
                   ┌──────────────┐
                   │    IRBT      │  Counter
                   └──────────────┘
```
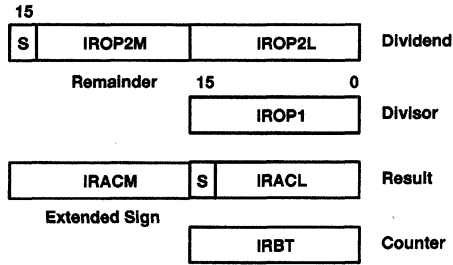
Figure 5–4. Signed Division – Register Use

```
; EXECUTION CYCLES FOR REGISTER CONTENTS (without CALL):
;DIVIDE            CYCLES              EXAMPLE
;------------------------------------------------------------------------------
; MINIMUM          15                  0xxxxxxxxh : 00000h = 0yyyyh    C = 1
;                  268                 0E01C3E02h : 05A5Ah = 0A5A5h    C = 0
;                  258                 000000001h : 0FFFFh = 0FFFFh    C = 0
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRBT
;
; SIGNED DIVISION SUBROUTINE 32-BIT BY 16-BIT
; IROP2M|IROP2L : IROP1 -> IRACL    REMAINDER IN IROP2M
; RETURN: CARRY = 0: OK    CARRY = 1: QUOTIENT > 16 BITS
;
DIVS    CLR     IRACM                       ; Sign of result
        TST     IROP2M                      ; Check sign of dividend
        JGE     DIVS1
        INV     IROP2M                      ; Is neg.: |dividend|
        INV     IROP2L
        INC     IROP2L
        ADC     IROP2M
        INV     IRACM                       ; Invert sign of result
DIVS1   TST     IROP1                       ; Check sign of divisor. C = 1
        JEQ     DIVSERR                     ; Divisor is 0: error. C = 1
        JGE     DIVS2                       ; Sign is neg.: |divisor|
        INV     IROP1
```

```
            INC     IROP1
            INV     IRACM                       ; Invert sign of result
DIVS2       CALL    #DIVIDE                     ; Call unsigned division
            JC      DIVSERR                     ; C = 1: error
            TST     IRACM                       ; Test sign of result
            JZ      DIVS3
            INV     IRACL                       ; Is neg.: negate result
            INC     IRACL
DIVS3       CLRC                                ; No error occured: C = 0
DIVSERR     RET                                 ; Error: C = 1
```

5.1.7 Shift Routines

The results of the previous subroutines (MPY, DIV) accumulated in IRACM/ IRACL have to be adapted to different numbers of bits after the decimal point because they are too large to fit into 32 bits. The following subroutines can do this function. If other types of number shifting is necessary, the subroutines can be constructed as shown for the 6-bit shifts (subroutine SHFTRS6). No tests are made for overflow.

```
; Signed shift right subroutine for IRACM/IRACL
; Definitions see above
;
SHFTRS6     CALL    #SHFTRS3                    ; Shift 6 bits right signed
SHFTRS3     RRA     IRACM                       ; Shift MSBs, bit0 -> carry
            RRC     IRACL                       ; Shift LSBs, carry -> bit15
SHFTRS2     RRA     IRACM
            RRC     IRACL
SHFTRS1     RRA     IRACM
            RRC     IRACL
            RET
;
; Unsigned shift right subroutine for IRACM/IRACL
;
SHFTRU6     CALL    #SHFTRU3                    ; Shift 6 bits right unsigned
SHFTRU3     CLRC                                ; Clear carry
            RRC     IRACM               ; Shift MSBs, bit0 -> carry, 0 -> bit15
            RRC     IRACL               ; Shift LSBs, carry -> bit15
```

```
SHFTRU2   CLRC
          RRC      IRACM
          RRC      IRACL
SHFTRU1   CLRC
          RRC      IRACM
          RRC      IRACL
          RET
;
; Signed/unsigned shift left subroutine for IRACM/IRACL
;
SHFTL6    CALL     #SHFTL3          ; Shift 6 bits left
SHFTL3    RLA      IRACL            ; Shift LSBs, bit0 -> carry
          RLC      IRACM            ; Shift MSBs, carry -> bit15
SHFTL2    RLA      IRACL
          RLC      IRACM
SHFTL1    RLA      IRACL
          RLC      IRACM
          RET
```

5.1.8   Square Root Routines

The square root of a number is often needed in computations. Two different methods are given:

❑   A very fast method for 32-bit integer numbers

❑   A normal method for 32-bit numbers that can have a fractional part

5.1.8.1   Square Root for 32-Bit Integer Numbers

The square root of a 30-bit integer number is calculated. The result contains 15 correct fractional bits. The subroutine uses the method known from the finding of a square root by hand. This method is much faster than the widely known NEWTONIAN method and only 720 cycles are needed. This subroutine was developed by Jürg Müller Software–Art GmbH/Zurich. The C program code needed is also shown:

```
{ unsigned long y, h;
  int  i;
  h = x;
  x = y = 0;
```

```
  for (i = 0; i < 32; i++)
  {                      // x ist eigentlich 2*x
    x <<= 1; x++;        // 4*x + 1
    if (y < x)
    {
      x -= 2;
    } else
      y -= x;
    x++;
    y <<= 1;             // <y, h> <<= 2
    if (h & Minus) y++;
    h <<= 1;
    y <<= 1;
    if (h & Minus) y++;
    h <<= 1;
  }
  return x;
}
; Square Root of a 32-bit number.
;
x_MSB    .equ    R4
x_LSB    .equ    R5
y_MSB    .equ    R6
y_LSB    .equ    R7
h_MSB    .equ    R8
h_LSB    .equ    R9
i        .equ    R10
;
; Call:    32-bit-Integer in x_MSB, x_LSB
; Result:  32-bit-number  in x_MSB  (16 bit integer part)
;                            x_LSB  (16 bit fraction)
;
; Range for x:      0 <= x    <= 40000000h
; Range for result: 0 <= SQRT <= 8000.0000h
; Max. Error:       0000.0002h
; Calculation Time: 720 cycles (t = 720/MCLK)
```

```
;
; Examples: sqrt (10000000h) = 4000.0000h
;           sqrt     (2710h) = 0000.0064h
;           sqrt       (2h) = 0001.6a09h = 92681 = 1.4142 * 2^16
;
Sqrt    Mov       x_MSB,h_MSB
        Mov       x_LSB,h_LSB
        Clr       x_MSB
        Clr       x_LSB
        Clr       y_MSB
        Clr       y_LSB
        Mov       #32,i
Sqrt10  SetC                        ; x <<= 1; x++;
        Rlc       x_LSB
        Rlc       x_MSB
        Sub       x_LSB,y_LSB       ; y.l -= x.l;
        Subc      x_MSB,y_MSB
        Jhs       Sqrt12            ; if (y.l & Minus)
        Add       x_LSB,y_LSB       ; {
        Addc      x_MSB,y_MSB       ;   y.l += x.l;
        Sub       #2,x_LSB          ;   x.l -= 2; }
Sqrt12  Inc       x_LSB             ; x.l++;
;                                     <y.l, HilfsReg> <<= 2
        Rla       h_LSB             ; <y.l, HilfsReg> <<= 1
        Rlc       h_MSB
        Rlc       y_LSB
        Rlc       y_MSB
        Rla       h_LSB             ; <y.l, HilfsReg> <<= 1
        Rlc       h_MSB
        Rlc       y_LSB
        Rlc       y_MSB
        Dec       i
        Jne       Sqrt10
        Ret
```

5.1.8.2 Square Root for 32-Bit Numbers

The following subroutine uses the Newtonian-approximation method for calculating the square root. The number of iterations depends on the length of the

operand. The subroutine was developed by A. Mühlhofer/TID. The general formula is:

$$\sqrt[m]{A} = X$$

$$X_{n+1} = \frac{1}{m}\left((m-1)\times X_n + \frac{A}{X_n^{m-1}}\right)$$

Where m = 2 (square root)

$$\sqrt{A} = X$$

$$X_{n+1} = \frac{1}{2}\times\left(X_n + \frac{A}{X_n}\right)$$

$$X_0 = \frac{A}{2}$$

To calculate A/Xn a division is necessary. This is done with the subroutine XDIV. The result of this division has the same integer format as the divisor Xn. This makes an easy operation possible.

```
Ah       .EQU    R8                      ;High word of A
Al       .EQU    R9                      ;Low word of A
XNh      .EQU    R10                     ;High word of result
XN1      .EQU    R11                     ;Low word of result
; Square Root
; The valid range for the operand is from 0000.0002h to
; 7FFF.ffffh
; EXAMPLE: SQR(2)=1.6a09h
;          SQR(7fff.ffffh) = B5.04f3h
;          SQR(0000.0002h) = 0.016ah
;
SQR      .EQU    $
         MOV     Ah,XNh                  ; set X0 to A/2 for the first
         MOV     Al,XN1                  ; approximation
         RRA     XNh                     ; X0=A/2
         RRC     XN1
SQR_1    CALL    #XDIV                   ; R12xR13=A/Xn
         ADD     R13,XN1                 ; Xn+1=Xn+A/Xn
         ADDC    R12,XNh
```

```
         RRA     XNh                       ; Xn+1=1/2(Xn+A/Xn)
         RRC     XN1
         CMP     XNh,R12                   ; is high word of Xn+1 = Xn
         JNE     SQR_1                     ; no, another approximation
         CMP     XN1,R13                   ; yes, is low word of Xn+1 = Xn
         JNE     SQR_1                     ; no, another approximation
SQR_3    RET                               ; yes, result is XNh.XN1
;
; Extended unsigned division
; R8|R9 / R10|R11 = R12|R13, remainder is in R14|R15
;
XDIV     .EQU    $
         PUSH    R8                        ; Save operands onto the stack
         PUSH    R9
         PUSH    R10
         PUSH    R11
         MOV     #48,R7                    ; Counter=48
         CLR     R15                       ; Clear remainder
         CLR     R14
         CLR     R12                       ; Clear result
         CLR     R13
L$361    RLA     R9                        ; Shift one bit of R8|R9 to R14|R15
         RLC     R8
         RLC     R15
         RLC     R14
         CMP     R10,R14                   ; Is subtraction necessary?
         JLO     L$364                     ; No
         JNE     L$363                     ; Yes
         CMP     R11,R15                   ; R11=R15
         JLO     L$364                     ; No
L$363    SUB     R11,R15                   ; Yes, subtract
         SUBC    R10,R14
L$364    RLC     R13                       ; Shift result to R12|R13
         RLC     R12
         DEC     R7                        ; Are 48 loops over ?
         JNZ     L$361                     ; No
```

```
        POP     R11                          ; Yes, restore operands

        POP     R10

        POP     R9

        POP     R8

        RET
```

5.1.9 Signed and Unsigned 32-Bit Compares

The following examples show optimized routines for the comparison of values longer than 16 bits. They can be enlarged to any length (i.e., 48 bit, 64 bit etc.).

```
; Comparison for unsigned 32-bit numbers: R11|R12 with R13|R14

;

        CMP     R11,R13                      ; Compare MSBs

        JNE     L$1                          ; MSBs are not equal

        CMP     R12,R14                      ; Equality: Compare LSBs too

L$1     JLO     LO                           ; Jumps are used for MSBs and LSBs

        JEQ     EQUAL                        ;

        ...                                  ; R13|R14 > R11|R12

LO      ...                                  ; R13|R14 < R11|R12

EQUAL   ...                                  ; R13|R14 = R11|R12
```

The approach shown can be adapted to any number length, only additional comparisons have to be added:

```
; Comparison for unsigned 48-bit numbers: R10|R11|R12 with

; R13|R14|R15

;

        CMP     R10,R13                      ; Compare MSBs

        JNE     L$1                          ; MSBs are not equal

        CMP     R11,R14                      ; Equality: Compare MSBs-1 too

        JNE     L$1                          ; MSBs-1 are not equal

        CMP     R12,R15                      ; Equality: Compare LSBs too

L$1     JLO     LO                           ; Jumps are used for all words

        JEQ     EQUAL                        ;

        ...                                  ; R13|R14|R15 > R10|R11|R12

LO      ...                                  ; R13|R14|R15 < R10|R11|R12

EQUAL   ...                                  ; R13|R14|R15 = R10|R11|R12
```

```
; Comparison for signed 32-bit numbers: R11|R12 with R13|R14
;
            CMP     R11,R13                 ; Compare MSBs signed
            JLT     LO                      ; R13 < R11
            JNE     HI                      ; Not LO, not EQUAL: only HI rests
            CMP     R12,R14                 ; Equality: Compare LSBs too
            JLO     LO                      ; LSBs use unsigned jumps!
            JEQ     EQUAL                   ; Not LO, not EQUAL: only HI rests
HI          ...                             ; R13|R14 > R11|R12
LO          ...                             ; R13|R14 < R11|R12
EQUAL       ...                             ; R13|R14 = R11|R12
; Comparison for signed 48-bit numbers: R10|R11|R12 with
; R13|R14|R15
;
            CMP     R10,R13                 ; Compare MSBs signed
            JLT     LO
            JNE     HI                      ; Not LO, not EQUAL: only HI rests
            CMP     R11,R14                 ; Equality: Compare MSBs-1 too
            JNE     L$1                     ; MSBs-1 are not equal
            CMP     R12,R15                 ; Equality: Compare LSBs too
L$1         JLO     LO                      ; Used for MSBs-1 and LSBs
            JEQ     EQUAL                   ; Not LO, not EQUAL: only HI rests
HI          ...                             ; R13|R14|R15 > R10|R11|R12
LO          ...                             ; R13|R14|R15 < R10|R11|R12
EQUAL       ...                             ; R13|R14|R15 = R10|R11|R12
```

5.1.10 Random Number Generation

The linear congruential method is used (introduced by D. Lehmer in 1951). The advantages of this method are speed, code simplicity, and ease of use. However, if care is not taken in choosing the multiplier and increment values, the results can quickly degenerate. This algorithm produces 65,536 unique numbers with very good correlation. Therefore, the random numbers repeat in the same sequence every 65,536. Within this sequence, only the LSB exhibits a repeatable pattern every 16 calls.

The linear congruential method has the following form:

$$Rndnum_n = \left(Rndnum_{n-1} \times MULT \right) + INC(modM)$$

Where:
Rndnum$_n$	Current random number
Rndnum$_{n-1}$	Previous random number
MULT	Multiplier (unique constant)
INC	Increment (unique constant)
M	Modulus (word width of MSP430 = 16 bits = 64K)

Many hours of research have been done to identify the optimal choices for the constants MULT and INC. The constant used in this implementation are based on this research. If changes are made to these numbers, extreme care must be taken to avoid degeneration. The following text is a more detailed look at the algorithm and the numbers used:

❑ M: M is the modulus value and is typically defined by the word width of the processor. The linear congruential algorithm returns a random number between 0 and 65,536 and is NOT internally bounded. If the user requires a min/max limit, this must be coded externally to this routine. The result is not actually divided by 65,536. The result register is allowed to overflow, thus implementing the modulus.

❑ SEED: The first random number in the sequence is called the seed value. This is an arbitrary constant between 0 and 64K. Zero can be used. This is OK if the code is allowed 3 calls to *warm up* before the numbers are considered valid. The number 21,845 was used in this implementation because it is 1/3 of the modulus (65,536).

❑ MULT: Based on random number theory, this number should be chosen such that the last three digits are even–2–1(such as xx821, x421, etc.). The number 31,821 was used in this implementation.

The generator is extremely sensitive to the choice of this constant!
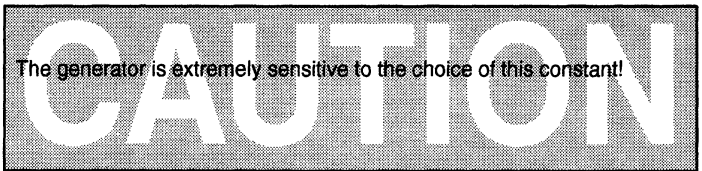
❑ INC: In general, this constant can be any prime number related to M. Two values were actually tested in this implementation: 1 and 13,849. Research shows that INC should be chosen based on the following formula:

$$INC = \left(\frac{1}{2} - \left(\frac{1}{6} \times \sqrt{3} \right) \right) \times M$$

(Using M=65,536 leads to INC=13,849)

The following code describes the first equation. Three subroutines are used to generate random numbers. Furthermore, the initialization of corresponding constants and of a RAM-variable storing the random number is included. The symbol names of the 1st equation are strictly used in the code underneath. The first time, an initialization routine INIRndnum must be called. Then a subroutine Rndum16 is called to calculate the random numbers as often as needed. The code necessary and the description of the subroutine MPYU can be found in Section 5.1.1, *Unsigned Multiplication 16 x 16-bits*.

```
;
; INITIALIZE CONSTANTS FOR RANDOM NUMBER GENERATION
;
SEED       .set    21845                    ; Arbitrary seed value (65536/3)
MULT       .set    31821                    ; Multiplier value (last 3
                                            ; Digits are even-2-1)
INC        .set    13849                    ; 1 and 13849 have been tested
HW_MPY     .set    0                        ; 1: HW-MPYer on chip
;
; ALLOCATION RANDOM NUMBER IN RAM-ADDRESS 200h
;
           .bss    Rndnum,2,0200h
;
; SUBROUTINE: INITIALIZE RANDOM NUMBER GENERATOR:
; Load the SEED value and produce the 1st random number
;
INIRndnum  .equ    $                        ; Uses Rndnum16
           MOV     #SEED,Rndnum             ; Initialize generator
;
; SUBROUTINE: GENERATES NEXT RANDOM NUMBER
; HW_MPY = 0: 169 cycles
; HW_MPY = 1:  26 cycles
;
Rndnum16   .equ    $
           .if     HW_MPY=0                 ; No MPYer
           MOV     Rndnum,IROP2L            ; Prepare multiplication
           MOV     #MULT,IROP1              ; Prepare multiplication
```

```
        CALL    #MPYU                   ; Call unsigned MPY (5.1.1)
        ADD     #INC,IRACL              ; Add INC to low word of product
;
; Overwrite old random number with low word of new product
;
        MOV     IRACL,Rndnum            ; Result to Rndnum and IRACL
        .else                           ; HW MPYer on chip
        MPYU    Rndnum,#MULT            ; Rndnum x MULT
        MOV     &ResLo,Rndnum           ; Low word of product
        ADD     #INC,Rndnum             ; Add INC to low word
        .endif
        RET                             ; Random number in Rndnum
EXAMPLE: Use of the Random Generator (1st call and succeeding calls).
;
; First call: produce the 1st random number
;
        CALL    #INIRndum               ; Initialize generator
        ....
; Second and all other calls to get the next random number
;
        CALL    #Rndnum16               ; Next random number to register
        ....                            ; IRACL and location Rndnum
```

Algorithm from *TMS320DSP Designer's Notebook Number 43 Random Number Generation on a TMS320C5x*. 7/94

5.1.11 Rules for the Integer Subroutines

Despite the fact that the subroutines shown previously can only handle integer numbers, it is possible to use numbers with fractional parts. It is necessary only to define for each number where the virtual decimal point is located. Relatively simple rules define where the decimal point is located for the result.

For calculations with the integer subroutines, it is almost impossible to remember where the virtual decimal point is located. It is good programming practice to indicate in the comment part of the software listing where the decimal point is currently located. The indication can have the following form:

N.M

where
N   Worst-case bit count of integer part (allows additional assessments)
M   Number of bits after the virtual decimal point

The rules for determining the location of the decimal point are simple:

❑ Addition and subtraction: Positions after the decimal point have to be equal. The position is the same for the result.

❑ Multiplication: Positions after the decimal point can be different. The two positions are added to get the result's position after the decimal point.

❑ Division: Positions after the decimal point can be different. The two positions are subtracted to get the result's position. (Dividend – divisor)

Table 5–1. Examples for the Virtual Decimal Point

First Operand	Operation	Second Operand	Result
NNN.MMM	+	NNNN.MMM	NNNN.MMM
NNN.M	×	NN.MMM	NNNNN.MMMM
NNN.MM	–	NN.MM	NNN.MM
NNNN.MMMM	:	NN.MMM	NN.M
NNN.M	+	NNNN.M	NNNN.M
NNN.MM	×	NN.MMM	NNNNN.MMMMM
NNN.M	–	NN.M	NNN.M
NNNN.MMMMM	:	NN.M	NN.MMMM

If two numbers have to be divided and the result needs n digits after the decimal point, the dividend has to be loaded with the number shifted appropriately to the left and zeroes filled into the lower bits. The same procedure can be used if a smaller number is to be divided by a larger one.

EXAMPLES for the division:

Table 5–2. Rules for the Virtual Decimal Point

First Operand (Shifted)	Operation	Second Operand	Result
NNNN.000	:	NN	NN.MMM
NNNN.000	:	NN.M	NN.MM
NNNN.000	:	N.MM	NNN.M
0.MMM000	:	NN.M	0.MMMMM

EXAMPLE for a source using the number indication:

```
MOV     #01234h,IROP2L          ; Constant 12.34h loaded     8.8
MOV     R15,IROP1               ; Operand fetched            2.3
CALL    #MPYS                   ; Signed MPY                 10.11
CALL    #SHFTRS3                ; Remove 3 fraction bits     10.8
ADD     #00678h,IRACL           ; Add Constant 6.78h         10.8
ADC     IRACM                   ; Add carry                  10.8
```

5.2 Table Processing

One of the development targets of the MSP430 was the capability of process-
ing tables because software can be written more legible and more functional
when using tables. The addressing modes, the instruction set, and the word/
byte structure make the MSP430 an excellent table processor. The arrange-
ment of information in tables has several advantages:

- ☐ Good visibility
- ☐ Simplifies changes: enlargements and deletions are made easily
- ☐ Low software overhead: Short programs
- ☐ High speed: Fastest way to access data

Generally, two ways exist of arranging data in tables:

- ☐ Data is arranged in blocks, each block containing the complete informa-
  tion of one item

- ☐ Data is arranged in several tables, each table containing one or two kinds
  of information for all items.
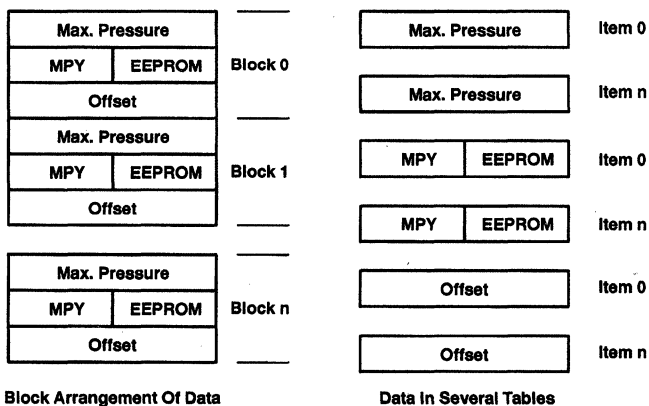
Figure 5–5. Data Arrangement in Tables

EXAMPLE: A table arranged in blocks is shown. Some examples for random
access are given. The addressed tables refer to Figure 5–5

```
;Block Arrangement of data
;
TABLE    .WORD    2095                    ; Maximum pressure item 0
```

```
TEEPR    .BYTE    16                    ; EEPROM start address
TMPY     .BYTE    3                     ; Multiply constant
TOFFS    .WORD    01456h                ; Offset correction value
;
TABN     .WORD    3084                  ; Maximum pressure item 1
         . . .
         .WORD    2010                  ; Maximum pressure item N
         .BYTE    37                    ; EEPROM start address
         .BYTE    3                     ; Multiply constant
         .WORD    00456h                ; Offset correction value
;
; Access examples for the above block arrangement:
; R5 points to the 1st word of a block (max. pressure)
; Examples how to access the other values are given:
;
         MOV      @R5,R6                ; Copy max. pressure to R6
         MOV.B    TEEPR-TABLE(R5),R7    ; EEPROM start to R7
         CMP.B    TMPY-TABLE(R5),R8     ; Same constant as in R8?
         MOV      &ADAT,R9              ; ADC result to R9
         ADD      TOFFS-TABLE(R5),R9    ; Correct ADC result
         ADD      #TABN-TABLE,R5        ; Address next item's block
;
; Copying of block arranged data to registers
;
         MOV      @R5+,R6               ; Copy max. pressure to R6
         MOV.B    @R5+,R7               ; EEPROM start to R7
         MOV.B    @R5+,R8               ; MPY constant to R8
         MOV      @R5+,R9               ; Offset to R9
;
; R5 points to next item's block now
```

EXAMPLE: A table arranged in several tables is shown. Some examples for random access are given. The addressed tables refer to Figure 5–5

```
; Arrangement of data in several tables
;
TMAXPR   .WORD    2095                  ; Maximum pressure item 0
```

```
            WORD     3084                    ; Maximum pressure item 1
            ...
            .WORD    2010                    ; Maximum pressure item N
    ;
    TEEMPY  .BYTE    16,3                    ; EEPROM start, MPY constant
            .BYTE    37,3                    ; item 1
            ...
            .BYTE    37,114                  ; item N
    ;
    TOFFS   .WORD    01456h                  ; Offset correction value
            ...
            .WORD    00456h                  ; item N
    ;
    ; Access examples for the above arrangement:
    ; R5 contains the item number x 2: (word offset)
    ; Examples with identical functions as for the block arrangement
    ; shown in the example before
    ;
            MOV      TMAXPR(R5),R6           ; Copy max. pressure to R6
            MOV.B    TEEMPY(R5),R7           ; EEPROM start to R7
            CMP.B    TMPY+1(R5),R8           ; Same constant as in R8?
            MOV      &ADAT,R9                ; ADC result to R9
            ADD      TOFFS(R5),R9            ; Correct ADC result
            INCD     R5                      ; Address next item
```

5.2.1 Two Dimensional Tables

The output value of a function often depends on two (or more) input values. If there is no algorithm for such a function, then a two (or more) dimensional table is needed. Examples of such functions are:

❏ The entropy of water depends on the inlet temperature and the outlet temperature. An approximation equation of the twelfth order is needed for this problem if no table is used.

❏ The ignition angle of an Otto-motor depends on the throttle opening and the motor revolutions per minute.

Figure 5–6 shows a function like the one described. The output value T depends on the input values X and Y.
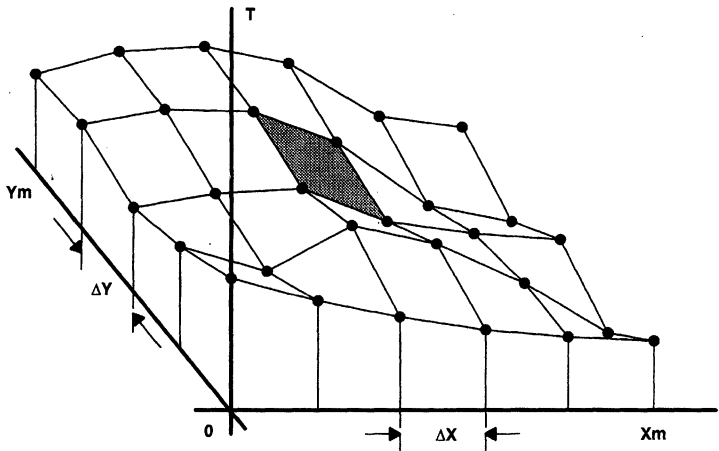
Figure 5–6. Two-Dimensional Function

A table contains the output values T for all crossing points of X and Y that have distances of ΔX and ΔY respectively. For every point in between these table points, the output value can be calculated.
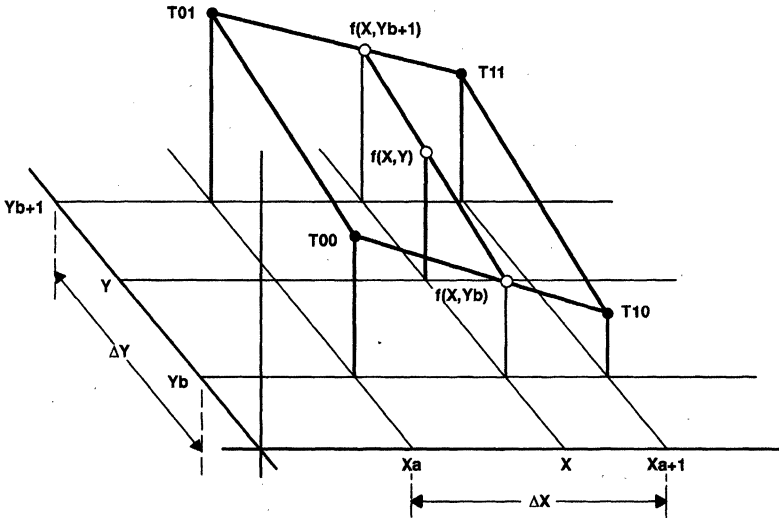
Figure 5–7. Algorithm for Two-Dimensional Tables

The calculation formulas are:

$$f(X, Yb) = \frac{X - Xa}{Xa_{+1} - Xa} \times (T_{10} - T_{00}) + T_{00} = \frac{X - Xa}{\Delta X} \times (T_{10} - T_{00}) + T_{00}$$

$$f(X, Yb_{+1}) = \frac{X - Xa}{\Delta X} \times (T_{11} - T_{01}) + T_{01}$$

$$f(X, Y) = \frac{Y - Yb}{\Delta Y} \times \left(f(X, Yb_{+1}) - (f(X, Yb)) \right) + f(X, Yb)$$

These formulas need division. There are two possible ways to avoid the division:

❏ To choose the values for ΔX and ΔY in such a way that simple shifts can do the divisions (ΔX = 0.25, 0.5, 1, 2, 4 etc.)

❏ To use adapted output values T' within the table

$$T'xy = \frac{Txy}{\Delta X \Delta Y}$$

This adaptation leads to:

$$\frac{f(X, Yb)}{\Delta Y} = (X - Xa) \times (T'10 - T'00) + T'00$$

$$\frac{f(X, Yb+1)}{\Delta Y} = (X - Xa) \times (T'11 - T'01) + T'01$$

$$f(X, Y) = (Y - Yb) \times \left(\frac{f(X, Yb+1)}{\Delta Y} - \frac{f(X, Yb)}{\Delta Y} \right) + \frac{f(X, Yb)}{\Delta Y} \times \Delta Y$$

The output value f(X,Y) is now calculated with multiplications only.

EXAMPLE: A 2-dimensional table is given. ΔX and ΔY are chosen as multiples of 2. The integer subroutines are used for the calculations

Note:

The software shown is not a generic example. It is tailored to the input values given. If other ΔX and ΔY values are used, the adaptation parts and masks have to be changed.

ITEM	X	Y	COMMENT
Delta	2	4	ΔX and ΔY
Input value format	8.2	7.1	Bits before/after decimal point
Starting value	0	0	X0 resp. Y0
End value	42	56	XM resp. YN
Input value (RAM, reg)	XIN	YIN	Assembler mnemonic

```
; Two dimensional table processing
;
XIN      .EQU    R15                    ; unsigned X value, register or RAM
YIN      .EQU    R14                    ; unsigned Y value, register or RAM
XM       .EQU    42                     ; Number of X rows
YN       .EQU    56                     ; Number of Y columns
XCL      .EQU    7                      ; Mask for fraction and dX
YCL      .EQU    7                      ; Mask for fraction and dY
XAYB     .EQU    R13                    ; Rel. address of (XA,YB), register
ZCFLG    .EQU    0                      ; Flag: 0: 2-dim    1: 3-dimensional
;
; Address definitions for the 4 table points:
```

```
;
T00       .EQU    TABLE                        ; (XA,YB)      TABLE(XAYB)
T01       .EQU    TABLE+2                      ; (XA,YB+1)    TABLE+2(XAYB)
T10       .EQU    TABLE+(YN*2)                 ; (XA+1,YB)    TABLE+(YN*2)(XAYB)
T11       .EQU    TABLE+(YN*2)+2               ; (XA+1,YB+1)  TABLE+(YM*2)+2(XAYB)
;
; Table for two dimensional processing. Contents are signed
; numbers.
;
TABLE     .WORD   01015h,...073A7h ; (X0,Y0) (X0,Y1)...(X0,YN)
          .WORD   02222h,...08E21h ; (X1,Y0) (X1,Y1)...(X1,YN)
          . . .
          .WORD   0A730h,...068D1h ; (XM,Y0) (XM,Y1)...(XM,YN)
;
; Table calculation software 2-dimensional. Approx. 700 cycles
;
; Input value X in XIN, Input value Y in YIN
; Result T in IRACL, same format as TABLE contents
;
; Calculation of YB out of YIN. One less adaption due to
; word table. Relative address of (X0,YB) to IRACL
;
TABCAL2   CLR     IRACM                ; 0 -> Hi result register
          MOV     YIN,IRACL            ; Y -> Lo result register     7.1
          RRA     IRACL                ; Shift out fraction part     7.0
          RRA     IRACL                ; Adapt to dY = 4             6.0
          BIC     #1,IRACL             ; Word address needed
;
; Calculation of XA out of XIN. One less adaption due to
; word table. Relative address of (XA,YB) to IRACL (T00)
;
          MOV     XIN,IROP1            ; X -> Multiplicand           8.2
          RRA     IROP1                ; Shift out fraction part     8.1
          RRA     IROP1                ; Adapt to dX = 2             8.0
          BIC     #1,IROP1             ; Word address needed
          MOV     #YN,IROP2L           ; Max. Y (YN) to multipl.     5.0
```

```
        CALL    #MACS               ; Rel address (XA,YB)         13.0
        MOV     IRACL,XAYB          ; to storage register         13.0
;
        .IF     ZCFLG               ; If 3-dimensional calculation
        ADD     OFFZC,XAYB          ; Add offset for actual table
        .ENDIF                      ; Rel. address of ZC
;
; Calculation of f(X,YB) = (XIN-XA)/dX x (T10-T00) + T00
;
        MOV     XIN,IROP1           ; build (XIN - XA)            8.2
        AND     #XCL,IROP1          ; Fraction and dX rests       1.2
        MOV     T10(XAYB),IROP2L    ; T10 -> IROP2L              16.0
        SUB     T00(XAYB),IROP2L    ; T10 - T00                  16.0
        CALL    #MPYS               ; (XIN - XA)(T10 - T00)      17.2
        CALL    #SHFTRS3            ; :dX, to integer            15.0
        ADD     T00(XAYB),IRACL     ; (XIN-XA)(T10-T00)+T00      15.0
        PUSH    IRACL               ; Result on stack
;
; Calculation of f(X,YB+1) = (XIN-XA)/dX x (T11-T01) + T01
; (XIN-XA) still in IROP1
;
        MOV     T11(XAYB),IROP2L    ; T11 -> IROP2L              16.0
        SUB     T01(XAYB),IROP2L    ; T11 - T01                  16.0
        CALL    #MPYS               ; (XIN - XA)(T11 - T01)      17.2
        CALL    #SHFTRS3            ; :dX, to integer            15.0
        ADD     T01(XAYB),IRACL   ; (XIN-XA)(T11-T01)+T01        15.0
;
; Calculation of f(X,Y) = (YIN-YB)/dY x (f(X,YB)-f(X,YB+1) +
; f(X,YB)
;
        MOV     YIN,IROP1           ; build (YIN - XB            7.1
        AND     #YCL,IROP1          ; Fraction and dX rests       2.1
        SUB     @SP,IRACL           ; f(X,YB+1)-f(X,YB)          16.0
        MOV     IRACL,IROP2L        ; Result to multiplier
        CALL    #MPYS               ; (YIN-YB)(f..-f..)          18.1
        CALL    #SHFTRS3            ; :dY, to integer            16.0
```

```
        ADD      @SP+,IRACL                    ; (YIN-YB)(f..-f..)+f..      15.0
        RET                                    ; Result T in IRACL          16.0
```

The table used with the previous example uses unsigned values for X and Y (the upper left hand table of Figure 5–8 shows this). If X or Y or both are signed values, the structure of the table and its entry point have to be changed. The following examples in Figure 5–8 show how to do that.
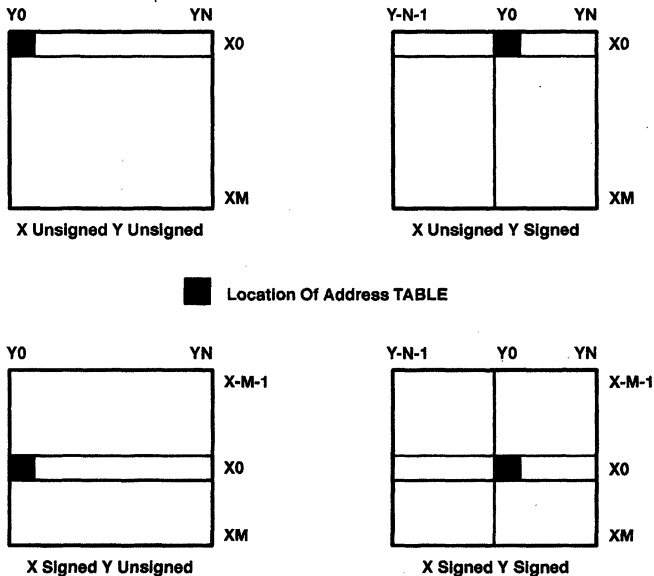
Figure 5–8. Table Configuration for Signed X and Y

The previous tables are shown in assembler code:

```
; X unsigned, Y unsigned
;
TABLE    .WORD    01015h,...073A7h             ; (X0,Y0)...(X0,YN)
         .WORD    02222h,...08E21h             ; (X1,Y0)...(X1,YN)
         ...
         .WORD    0A73h,...068D1h              ; (XM,Y0)...(XM,YN)
;
; X unsigned, Y signed
```

```
;
            .WORD    03017h,...093A2h        ; (X0,Y-N-1)...(X0,Y-1)
TABLE       .WORD    02233h,...08721h        ; (X0,Y0)...(X0,YN)
            .WORD    03017h,...093A2h        ; (X1,Y-N-1)...(X1,YN)
            ...
            .WORD    00173h,...07851h        ; (XM,Y-N-1)...(XM,YN)
;
; X signed, Y unsigned
;
            .WORD    03017h,...093A2h        ; (X-M-1,Y0)...(X-M-1,YN)
            .WORD    08012h,...0B3C1h        ; (X-M,Y0).....(X-M,YN)
            ...
            .WORD    04019h,...0D3A3h        ; (X-1,Y0)...(X-1,YN)
TABLE       .WORD    02233h,...08721h        ; (X0,Y0)....(X0,YN)
            .WORD    03017h,...093A2h        ; (X1,Y0)....(X1,YN)
            ...
            .WORD    00173h,...07851h        ; (XM,Y0)....(XM,YN)
;
; X signed, Y signed
;
            .WORD    03017h,...093A2h        ; (X-M-1,Y-N-1)(X-M-1,YN)
            .WORD    08012h,...0B3C1h        ; (X-M,Y-N-1) ...(X-M,YN)
            ...
            .WORD    04019h,...0D3A3h        ; (X-1,Y-N-1)...(X-1,YN)
            .WORD    02233h,...08721h        ; (X0,Y-N-1)....(X0,Y-1)
TABLE       .WORD    02233h,...08721h        ; (X0,Y0).......(X0,YN)
            .WORD    03017h,...093A2h        ; (X1,Y-N-1)....(X1,YN)
            ...
            .WORD    00173h,...07851h        ; (XM,Y-N-1)....(XM,YN)
```

The entry label TABLE always points to the word or byte with the coordinates (X0,Y0).

5.2.2 Three-Dimensional Tables

If the output value T depends on three input variables X, Y and Z, a three-dimensional table is necessary for the crossing points. Eight values T000 to T111 are used for the calculation of the output value T.

The simplest way is to compute these figures is to calculate the output values for both two-dimensional tables f(X,Y,Zc) and f(X,Y,Zc+1) with the subroutine TABCAL2. The two results are used for the final calculation:

$$f(X, Y, Z) = \frac{Z - Zc}{Zc + 1 - Zc} \times \left(f(X, Y, Zc + 1) - (f(X, Y, Zc)) \right) + f(X, Y, Zc)$$

The following figure shows this method. The output values Txxx are calculated for Zc and for Zc+1. Out of these two output values, the final value f(X,Y,Z), is calculated.
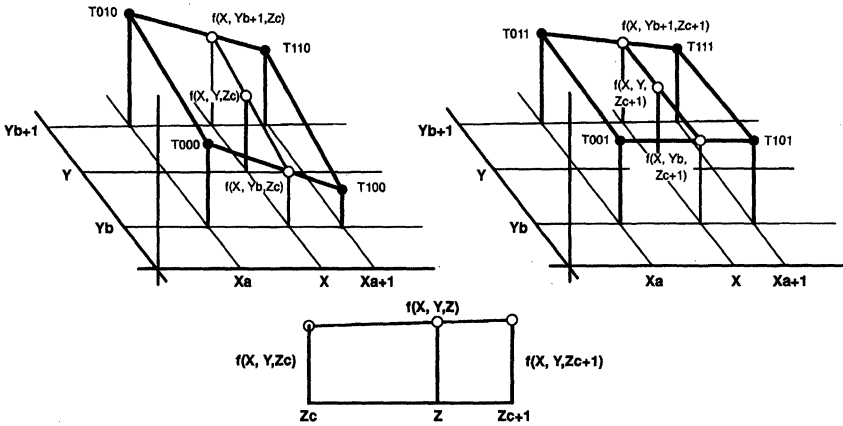
Figure 5–9. Algorithm for a Three-Dimensional Table

EXAMPLE: A three-dimensional table is given. ΔX and ΔY and ΔZ are chosen as multiples of 2. The integer subroutines are used for calculations.

ITEM	X	Y	Z	COMMENT
Delta	2	4	256	$\Delta X, \Delta Y, \Delta Z$
Input value format	8.2	7.1	0	Bits after decimal point
Starting value	0	0	0	X0, Y0, Z0
End value	42	56	$2^{14}-1$	XM, YN, ZP
Input value (RAM, register)	XIN	YIN	ZIN	Assembler mnemonic

```
;
XIN      .EQU    R15              ; unsigned X value, register or RAM
YIN      .EQU    R14              ; unsigned Y value, register or RAM
```

```
ZIN      .EQU    R13              ; unsigned Z value, register or RAM
XM       .EQU    42               ; Number of X rows
YN       .EQU    56               ; Number of Y columns
XCL      .EQU    7                ; Mask for fraction and dX
YCL      .EQU    7                ; Mask for fraction and dY
ZCL      .EQU    0FFh             ; Mask for deltaZ
XAYB     .EQU    R12              ; Rel. address of (XA,YB), register
ZCFLG    .EQU    1                ; Flag: 0: 2-dim.   1: 3-dim. Table
OFFZC    .EQU    R11              ; Relative offset to actual (X0,Y0,ZC)
;
; Three dimensional table
;
TABL3D   .WORD   01015h,...073A7h ; (X0,Y0,Z0)...(X0,YN,Z0)
         ...
         .WORD   02222h,...08E21h ; (XM,Y0,Z0)...(XM,YN,Z0)
;
         .WORD   0A730h,...068D1h ; (X0,Y0,Z1)...(X0,YN,Z1)
         ...
         .WORD   010A5h,...09BA7h ; (XM,Y0,Z1)...(XM,YN,Z1)
;
         .WORD   02BC2h,...08E41h ; (X0,Y0,ZP)...(X0,YN,ZP)
         ...
         .WORD   0A980h,...023D1h ; (XM,Y0,ZP)...(XM,YN,ZP)
; Table calculation software 3-dimensional
; Input values: X in XIN, Y in YIN, Z in ZIN
; Result is located in IRACL, same format as TABLE content
;
; Calculation of ZC out of ZIN. One less adaption due to
; word table.
;
TABCAL3  MOV     ZIN,IROP1        ; Z -> Operand register        14.0
         SWPB    IROP1            ; Use only upper byte (dZ =256)
         MOV.B   IROP1,IROP1      ; Adapt to dZ = 256             6.0
;
; Calculation of relative address of (X0,Y0,ZC) to IRACL
; Corrected for word table
```

```
;
        MOV     #YN*2*XM,IROP2L ; Table length for dZ
        CALL    #MPYU                   ; Rel address (X0,Y0,ZC)        13.0
        MOV     IRACL,OFFZC             ; to storage register          13.0
;
; Calculation of f(X,Y,ZC): The table block for ZC is used
;
        CALL    #TABCAL2                ; f(X,Y,ZC) -> IRACL           16.0
        PUSH    IRACL                   ; Save f(X,Y,ZC)
; Calculation of f(X,Y,ZC+1): The table block for ZC+1 is used
;
        ADD     #YN*2*XM,OFFZC          ; Rel. adress (X0,Y0,ZC+1)
        CALL    #TABCAL2                ; f(X,Y,ZC+1) -> IRACL         16.0
;
; Calculation of f(X,Y,Z)
;
        MOV     ZIN,IROP1               ; build (YIN - XB               6.8
        AND     #ZCL,IROP1              ; Fraction and dZ rests         0.8
        SUB     @SP,IRACL               : f(X,Y,ZC+1)-f(X,Y,ZC)        16.0
        MOV     IRACL,IROP2L            ; Result to multiplier
        CALL    #MPYS                   ; (ZIN-ZC)(f..-f..)            16.8
        CALL    #SHFTRS6        ;       :dZ, to integer               16.2
        CALL    #SHFTRS2                ;                              16.0
        ADD     @SP+,IRACL              ; (ZIN-ZC)(f..-f..)+f...       15.0
        RET                             ; Result in IRACL
```

5.3 Signal Averaging and Noise Cancellation

If the measured signals contain noise, spikes, and other unwanted signal components, it may be necessary to average the ADC results. Six different methods are mentioned here:

1) Oversampling: Several measurements are added-up and the accumulated sum is used for the calculations.

2) Continuous Averaging: A circular buffer is used for the measured samples. With every new sample a new average value can be calculated.

3) Weighted summation: The old value and the new one are added together and divided by two afterwards.

4) Wave Digital Filtering: Complex filter algorithms, which need only small amounts of calculation power, are used for the signal conditioning.

5) Rejection of Extremes: the largest and the smallest samples are rejected from the measured values and the remaining samples are added-up and averaged.

6) Synchronization of the measurements to hum

The advantages and disadvantages of the different methods are shown in the following sections.

5.3.1 Oversampling

Oversampling is the simplest method for the averaging of measurement results. N samples are added-up and the accumulated sum is divided by N afterwards or is used as it is with the following algorithm steps. It is only necessary to remember that the accumulated value is N-times too large. For example, the following formula, used for a single measurement, needs to be modified when N samples are summed-up as shown:

$$V_{normal} = Slope \times ADC + Offset \quad \rightarrow \quad V_{oversample} = \frac{\Sigma\, (\, Slope \times ADC + Offset\,)}{N}$$

EXAMPLE: N measurements have to be summed-up in SUM and SUM+2. The number N is defined in R6

```
SUMLO    .EQU    R4                      ; LSBs of sum
SUMHI    .EQU    R5                      ; MSBs of sum
;
```

```
          CLR     SUMLO                    ; Init of registers
          CLR     SUMHI
          MOV     #16,R6                   ; Sum-up 16 samples of the ADC
OVSLOP    CALL    #MEASURE                 ; Result in ADAT
          ADD     &ADAT,SUMLO              ; LSD of accumulated sum
          ADC     SUMHI                    ; MSD
          DEC     R6                       ; Decr. N counter: 0 reached?
          JNZ     OVSLOP
          ...                              ; Yes, 16 samples in SUMHI|SUMLO
```

❏ Advantages

■ Simple programming

❏ Disadvantages

■ High current consumption due to number of ADC conversions

■ Low suppression of spikes etc. (by N)

5.3.2 Continuous Averaging

A very simple and fast way for averaging digital signals is continuous averaging. A circular buffer is fed at one end with the newest sample and the oldest sample and is deleted at the other end (both items share the same RAM location). To reduce the calculation time, the oldest sample is subtracted from the actual sum and the new sample is added to the sum. The actual sum (a 32-bit value containing N samples) is used by the background. For calculations, it is only necessary to remember that it contains the accumulated sum of N samples. The same rule is valid for oversampling.

The characteristic of this averaging is similar to a comb filter with relatively good suppression of frequencies that are integral multiples of the scanning frequency. The frequency behavior is shown in the following figure.
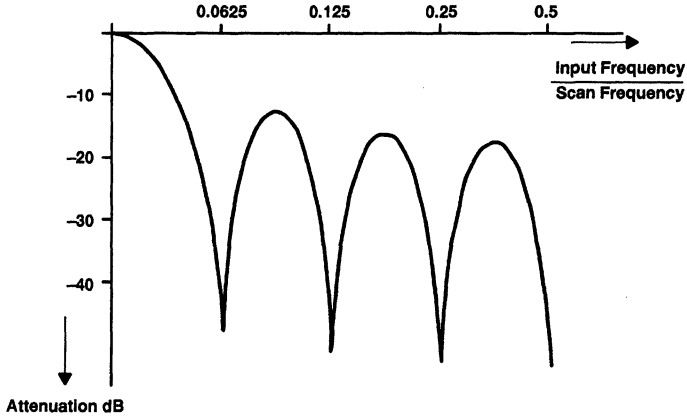
Figure 5–10. Frequency Response of the Continuous Averaging Filter

❑ Advantages

 ■ Low current consumption with only one measurement

 ■ Fast update of buffer

 ■ Good suppression of certain frequencies (multiples of scan frequency)

 ■ Low-pass filter characteristic

❑ Disadvantages

 ■ RAM allocation: N words are needed for the circular buffer

EXAMPLE: An interrupt driven routine (e.g. from the ADC, which is started by the basic timer) that updates a circular buffer with N items is shown. The actual sum CFSUM is calculated by subtracting of the oldest sample and adding of the newest one. CFSUM and CFSUM+2 contain the sum of the latest N samples.

```
N       .EQU    16                      ; Circular buffer with N items
        .BSS    CFSTRT,N*2              ; Address of 1st item
        .BSS    CFSUM,4                 ; Accumulated sum 32 bits
```

```
         .BSS    CFPOI,2                 ; Points to next (= oldest) item
;
CFHND    PUSH    R5                      ; Save R5
         MOV     CFPOI,R5                ; Actual address to R5
         CMP     #CFSTRT+(N*2),R5        ; Outside circ. buffer?
         JLO     L$300                   ; No
         MOV     #CFSTRT,R5              ; Yes, reset pointer
;
; The oldest item is subtracted from the sum. The newest item
; overwrites the oldest one and is added to the sum
;
L$300    SUB     @R5,CFSUM               ; Subtract oldest item from CFSUM
         SBC     CFSUM+2
         MOV     &ADAT,0(R5)             ; Move actual item to buffer
         ADD     @R5+,CFSUM              ; Add latest ADC result to CFSUM
         ADC     CFSUM+2
         MOV     R5,CFPOI                ; Update pointer
         POP     R5                      ; Restore R5
         RETI
```

5.3.3   Weighted Summation

The weighted sum of the measurements before and the current measurement result are added and then divided by two. This gives every measurement result a certain weight.

Table 5–3. Sample Weight

MEASUREMENT TIME	WEIGHT	COMMENT
t0	0.5	Actual measurement
t0 − Δt	0.25	Last measurement
t0 − 2Δt	0.125	
t0 − 3Δt	0.0625	
t0 − 4Δt	0.03125	
t0 − nΔt	$2^{-(n+1)}$	

❑ Advantages

■ Low current consumption due to one measurement only

■ Low pass filter characteristic

■ Very short code

■ Only one RAM word needed

❏ Disadvantages

■ Suppression of spikes not sufficient (factor 2 only for actual sample)

EXAMPLE: The update of the actual sum WSSUM is shown.

```
       .BSS    WSSUM,2                    ; Accumulated weighted sum
;
WSHND  ADD     &ADAT,WSSUM                ; Add actual measurement to sum
       RRA     WSSUM                      ; New sum divided by 2
       ...                                ; Continue with value in WSSUM
```

5.3.4 Wave Digital Filtering

Wave digital filters (WDFs) have notable advantages:

❏ Excellent stability even under nonlinear operating conditions resulting from overflow and roundoff effects

❏ Low coefficient word length requirements

❏ Inherently good dynamic range

❏ Stability under looped conditions

Compared with the often used averaging of measured sensor data, the digital filtering has advantages: Lowpass filtering with sharp cut-off region, notch filtering of noise.

For the design of WDF algorithms specialized CAD programs have been designed to speed-up the top-down design from filter specification to the machine program for the processor:

❏ LWDF_DESIGN allows the design of Lattice-WDFs

❏ LWDF_COMP transforms a Lattice-WDF structure into an assembler program for the MSP430

❏ DSP430 allows fast transient simulations of the filter algorithms on a model of the MSP430, analysis of frequency response, check of accuracy and stability proof.

The programs enable the users of the MSP430 to solve special measurement problems by means of robust digital filter algorithms.

A complete description of the WDF algorithms and development tools is given in the "*TEXAS INSTRUMENTS Technical Journal*" November/December 1994.

❑ Disadvantages

■ Complex algorithm. Support software needed for finding algorithm

❑ Advantages

■ Low current consumption because only one measurement per time slice needed

■ Good attenuation inside stopband

■ Good dynamic stability

5.3.5 Rejection of Extremes

This averaging method measures (N+2) ADC-samples and rejects the largest and the smallest values. The remaining N samples are added-up and the accumulated sum is divided by N afterwards or is used as it is with the next algorithm steps. It is only necessary to remember that the added-up value is N-times too large.

❑ Disadvantages

■ Current consumption high due to (N+2) ADC conversions

❑ Advantages

■ Simple programming

■ Very good suppression of spikes (extremes are rejected)

■ Small amount of RAM needs (4 words)

The following software example adds six ADC samples, subtracts the two extremes, and returns with the sum of the four medium samples. The constant N can be changed to any number, but the summing-up buffer SESUM needs two words if N exceeds two. It is an advantage to use powers of two for N due to the simple divisions needed (right shifts only). Register use is possible for SESUM, SEHI and SELO.

```
N       .EQU    4                       ; Sample count used -2
        .IF     N>2
        .BSS    SESUM,4                 ; Summing-up buffer
        .ELSE                           ;
        .BSS    SESUM,2                 ; N<=2
```

```
          .ENDIF
          .BSS    SEHI,2                    ; Largest ADC result
          .BSS    SELO,2                    ; Smallest ADC result
          .BSS    SECNT,1                   ; Counter for N+2
;
SEHND     CLR     SESUM                     ; Initialize buffers
          .IF     N>2
          CLR     SESUM+2
          .ENDIF
          MOV.B   #N+2,SECNT                ; Sample count +2 to counter
          MOV     #0FFFFh,SELO              ; ADCmax -> SELO
          CLR     SEHI                      ; ADCmin -> SEHI
;
; N+2 measurements are made and summed-up in SESUM
;
SELOOP    CALL    #MEASURE                  ; ADC result to &ADAT
          MOV     &ADAT,R5                  ; Copy ADC result to R5
          ADD     R5,SESUM
          .IF     N>2                       ; Use 2nd sum buffer if N>2
          ADC     SESUM+2
          .ENDIF
          CMP     R5,SEHI                   ; Result > SEHI?
          JHS     L$1                       ; No
          MOV     R5,SEHI                   ; Yes, actualize SEHI
L$1       CMP     R5,SELO                   ; Result < SELO?
          JLO     L$2                       ; No
          MOV     R5,SELO
L$2       DEC.B   SECNT                     ; Counter - 1
          JNZ     SELOOP                    ; N+2 not yet reached
;
; N+2 measurements are made, extremes are subtracted now
; from summed-up result. Return with N-times value in SESUM
;
          SUB     SELO,SESUM                ; Subtract lowest result
          .IF     N>2                       ; Necessary if N>2
          SBC     SESUM+2
```

```
.ENDIF
SUB      SEHI,SESUM                ; Subtract highest result
.IF      N>2                       ; Necessary if N>2
SBC      SESUM+2
.ENDIF
RET
```

5.3.6 Synchronization of the Measurement to Hum

If hum plays a role during measurements then a synchronization to the ac frequency can help to overcome this problem. Figure 5–11 shows the influence of the ac voltage during the measurement of a single sensor. The necessary number of measurements (here 10 are needed) is split into two equal parts, the second part is measured after exactly one half of the period T_{ac} of the ac frequency. The hum introduced to the two parts is equal but has different signs. Therefore the accumulated influence (the sum) is nearly zero.
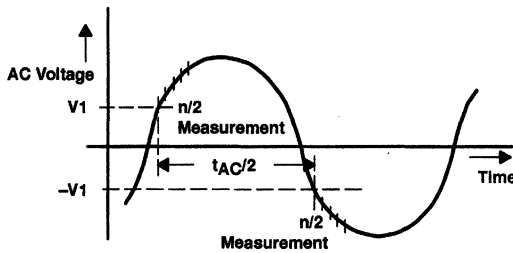
Figure 5–11. Reduction of Hum by Synchronizing to the AC Frequency. Single Measurement

If the basic timer is used for the timing then the following numbers of basic timer interrupts can be used.

Table 5–4. Basic Timer Frequencies for Hum Suppression

AC Frequency f_{ac}	Basic Timer Frequency f_{BT}	Number of BT Interrupts k	Time Error e_t max.	Residual Error e_r max
50Hz	4096Hz	41	0.097%	0.61%
60Hz	2048Hz	17	−0.39%	−2.45%

The formulas to get the above errors are:

$$e_t = \left(\frac{T_{BT}}{T_{AC}} \times 2k - 1 \right) \times 100$$

$$e_r = sin\left(\frac{T_{BT}}{T_{AC}} \times 2k \times 2\pi \right) \times 100$$

where:

e_t Maximum time error due to fixed basic timer frequency (in %)

e_r Maximum remaining influence of the hum (in %) compared to a measurement without hum cancellation

T_{BT} Period of basic timer frequency ($1/f_{BT}$)

T_{ac} Period of ac ($1/f_{ac}$)

k   Number of basic timer interrupts to reach $T_{ac}/2$ respective of T_{ac}

If difference measurements are used, the two measurements to be subtracted should be made with a delay of exactly one ac period. Both measurements have the same influence from the hum and the result, the difference of both measurements, does not show the error. This measurement method is used with heat meters, where the temperature difference of the water inlet and the water outlet is used for calculations.
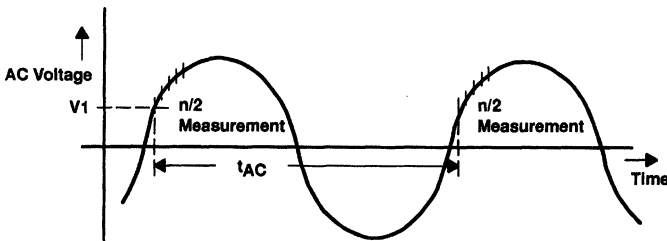
Figure 5–12. Reduction of Hum by Synchronizing to the AC Frequency. Differential Measurement

If the basic timer is used for the timing then the following numbers of basic timer interrupts can be used.

Table 5–5. Basic Timer Frequencies for Hum Suppression

AC Frequency f_{ac}	Basic Timer Frequency f_{BT}	Number of Interrupts k	Time Error e_t max.	Residual Error e_r max.
50Hz	2048Hz	41	0.097%	0.61%
60Hz	1024Hz	17	-0.39%	-2.45%

The formulas to get the previous results are:

$$e_t = \left(\frac{T_{BT}}{T_{AC}} \times k - 1 \right) \times 100$$

$$e_r = \sin\left(\frac{T_{BT}}{T_{AC}} \times k \times 2\pi \right) \times 100$$

The software needed for the modification of the Basic Timer frequency without the loss of the exact time base is shown in Section 6.1.1, *Change of the Basic Timer Frequency.*

5.4 Real-Time Applications

Real-time applications for microprocessors are defined often as follows:

❑ The controlling processor is able, under worst case conditions, to finish the necessary control algorithms before the next sample of the control input arrives.

The architecture of the MSP430 is ideally suited for real time applications due to its system clock generation. The system clock MCLK of the CPU is not generated by a second crystal, which needs a lot of time until it is oscillating with the nominal frequency. But, by the multiplication of the frequency of the 32-kHz crystal that is oscillating continuously.

5.4.1 Active Mode

The active mode shows the fastest response to interrupts because all of the internal clocks are operating at their nominal frequencies. The active mode is recommended when the speed of the MSP430 is the critical factor of an application.

5.4.2 Normal Mode is Low-Power Mode 3 (LPM3)

This mode is used for battery-driven systems where the power consumption plays an overwhelming role. Battery lifetimes over ten years are only possible when the CPU is switched off whenever its processing capability is not needed.

Despite the switched-off CPU, the MSP430 is at the start address of the interrupt handler within eight MCLK cycles; the system clock oscillator is then working at the correct frequency. This means true real-time capability, no delay due to the slow coming-up of the main oscillator crystal (up to 400 ms) is slowing down the system behavior.

See Section 6.5, *The System Clock Generator*, for the details of the programming.

5.4.3 Normal Mode is Low-Power Mode 4 (LPM4)

The low power mode 4 is used if there are relatively long time elapses between two interrupt events. The power consumption goes below 0.1mA if this mode is used All oscillators are switched off and only the RAM and the interrupt hardware are powered.

Despite this inactivity the MSP430 CPU is at the start of the interrupt handler within eight cycles of the programmed DCO tap. See Section 6.5, *The System Clock Generator* for the details of the speed-up of the CPU.

5.4.4 Recommendations for Real-Time Applications

☐ Switch on the GIE bit (SR.3) as soon as possible. Within the interrupt handlers, only the tasks that do not allow interruption should be completed first. This allows nested interrupts and avoids the blocking of other interrupts.

☐ Interrupt handlers (foreground) should be as short as possible. All calculations should be made in the background part of the program. The communication between these two software parts is made by status bytes. See Section 9.2.5, *Flag Replacement by Status Usage*.

☐ Use status bytes and calculated branches.

☐ The interrupt capability of the I/O ports makes input polling superfluous. Any change of an input is seen immediately. Use of the ports this way is recommended.

☐ Disabling and enabling of the peripheral interrupts during the software run is not recommended. Additional interrupt requests can result from these manipulations. The use of status bytes is recommended instead. They inform the software if an interrupt is valid or not. If not, it is neglected.

5.5 General-Purpose Subroutines

The following, tested software examples can be of help during the software development phase. The examples can not fit into any application, but they can be modified easily to the user's needs.

5.5.1 Initialization

For the first power-on, it is necessary to clear the internal RAM to get a defined basis. If the MSP430 is battery powered and contains calibration factors or other important data in its RAM, it is necessary to distinguish between a cold start and a warm start. The reason for this is the possibility of initializations caused by electromagnetic interference (EMI). If such an erroneous initialization is not checked for legality, EMI influence could destroy the RAM content by clearing the RAM with the initialization software routine. Testing can be done by comparing RAM bytes with known content to their nominal value. These RAM bytes could be identification codes or non-critical test patterns (e.g. A5h, F0h). If the tested RAM locations contain the correct pattern, a spurious signal caused the initialization and the normal program can continue. If the tested RAM bytes differ from the nominal value, the RAM content is destroyed (e.g. by a power loss) and the initialization routine is invoked. The RAM is cleared and the peripherals are initialized.

The cold start software contains the waiting loop for the DCO, which is needed to set it to the correct frequency. See Section 6.5, *The System Clock Generator*, and Section 6.6, *The RESET Function*.

```
; Initialization part: Check if Cold Start or Warm Start:
; RAM location 0200h decides kind of initialization:
; Cold Start: content differs from 0A5F0h
; Warm Start: content is 0A5F0h
;
INIT    CMP     #0A5F0h,&0200h          ; Test content of &200h
        JEQ     EMIINI                  ; Correct content: No reset
;
; Control RAM content differs from 0A5F0: RAM needs to be
; cleared, peripherals needs to be initialized
;
        MOV     #0300h,SP               ; Init. Stack Pointer
        CALL    #RAMCLR                 ; Clear complete RAM
        MOV     #0A5F0h,&0200h          ; Insert test word
;
```

```
; System frequency MCLK is set to 2.048MHz
;
        MOV.B   #64-1,&SCFQCTL          ; 64 x 32kHz = 2.048MHz
        MOV.B   #FN_2,&SCFI0            ; DCO current for 2MHz
;
; Waiting loop for the DCO of the FLL to settle: 130ms
;
        CLR     R5                      ; 3 x 65536us = 186ms
L$1     INC     R5
        JNZ     L$1
        ...
;
; EMI caused initialization: Periphery needs to be initialized:
; Interrupts need to be enabled again
;
EMINI   ...
```

5.5.2   RAM clearing Routine

The RAM is cleared starting at label RAMSTRT up to label RAMEND (inclusive).

```
;
; Definitions for the RAM block (depends on MSP430 type)
;
RAMSTRT .EQU    0200h                   ; Start of RAM
RAMEND  .EQU    02FEh                   ; Last RAM address (return address)
; Subroutine for the clearing of the RAM block
RAMCLR  CLR     R5                      ; Prepare index register
RCL     CLR.B   RAMSTRT(R5)             ; 1st RAM address
        INC     R5                      ; Next address
        CMP     #RAMEND-RAMSTRT+1,R5    ; RAM cleared?
        JLO     RCL                     ; No, once more
        RET                             ; Yes, return
;
```

5.5.3   Binary to BCD Conversion

The conversion of binary to BCD and vice versa is normally a time consuming task. Five divisions by ten are necessary to convert a 16-bit binary number to BCD. The DADD instruction reduces this to a loop with five instructions.

```
; THE BINARY NUMBER IN R12 IS CONVERTED TO A 5-DIGIT BCD
; NUMBER CONTAINED IN R14 AND R13: R14|R13
;
BINDEC    MOV      #16,R15                    ; LOOP COUNTER
          CLR      R14                        ; 0 -> RESULT MSD
          CLR      R13                        ; 0 -> RESULT LSD
L$1       RLA      R12                        ; Binary MSB to carry
          DADD     R13,R13                    ; RESULT x2 LSD
          DADD     R14,R14                    ;            MSD
          DEC      R15                        ; THROUGH?
          JNZ      L$1
          RET                                 ; YES, RESULT IN R14|R13
```

The previous subroutine can be enlarged to any length of the binary part simply by the adding of registers for the storage of the BCD number (a binary number with n bits needs approximately 1.2 x n bits for BCD format).

If numbers containing fractions have to be converted to BCD, the following algorithm can be used:

1) Multiply the binary number as often with 5 as there are fractional bits. For example if the number looks like MMM.NN, then multiply it with 25. Ensure that no overflow will take place.

2) Convert the result of step 1 to BCD with the (eventually enlarged) subroutine BINDEC. The BCD result is a number with the same number of fractional digits as the binary number has fractional bits.

EXAMPLE: The hexadecimal number 0A8Bh has the binary format MMM.NNN. The decimal value is therefore 337,375. The steps to get the BCD number are:

3) 0A8Bh is to be multiplied by 5^3 or 125_{10} due to its 3 fractional bits. 0A8Bh x 125_{10} =0525DFh

4) 0525DFh has the decimal equivalent 337,375. The correct BCD number with 3 fractional digits.

To convert the previous example, the basic subroutine BINDEC needs to be enlarged. Two binary registers are necessary to hold the input number.

```
; THE BINARY NUMBER IN R12|R11 IS CONVERTED TO AN 8-DIGIT BCD
; NUMBER CONTAINED IN R14 AND R13: R14|R13
; Max. hex number in R12|R11: 05F5E0FFh (999999999)
```

```
;
BINDEC    MOV     #32,R15           ; LOOP COUNTER
          CLR     R14               ; 0 -> RESULT MSD
          CLR     R13               ; 0 -> RESULT LSD
L$1       RLA     R11               ; MSB of LSBs to carry
          RLC     R12               ; Binary MSB to carry
          DADD    R13,R13           ; RESULT x2 LSD
          DADD    R14,R14           ;          MSD
          DEC     R15               ; THROUGH?
          JNZ     L$1
          RET                       ; YES, RESULT IN R14|R13
```

5.5.4 BCD to Binary

This subroutine converts a packed 16-bit BCD word to a 16-bit binary word by multiplying each digit with its decimal value (10^0, 10^1, . . .). To reduce code length, the HORNER scheme is used as follows:

$$R5 \;=\; X0 + 10(X1 + 10(X2 + 10X3)$$

```
; The packed BCD number contained in R4 is converted to a binary
; number contained in R5
;
BCDBIN    MOV     #4,R8             ; LOOP COUNTER ( 4 DIGITS )
          CLR     R5
          CLR     R6
SHFT4     RLA     R4                ; SHIFT LEFT DIGIT INTO R6
          RLC     R6                ; THROUGH CARRY
          RLA     R4
          RLC     R6
          RLA     R4
          RLC     R6
          RLA     R4
          RLC     R6
          ADD     R6,R5             ; X_N+10X_N+1
          CLR     R6
          DEC     R8                ; THROUGH ?
          JZ      END               ; YES
```

```
MPY10    RLA     R5                          ; NO, MULTIPLICATION WITH 10
         MOV     R5,R7                       ; DOUBLED VALUE
         RLA     R5
         RLA     R5
         ADD     R7,R5                       ; VALUE X 8
         JMP     SHFT4                       ; NEXT DIGIT
END      RET                                 ; RESULT IS IN R5
```

5.5.5 Keyboard Scan

A lot of possibilities exist for the scanning of a keyboard, which also includes jumpers and digital input signals. If more input signals exist than free inputs, scanning is necessary. The scanning outputs can be I/O-ports and unused segment outputs, On. The scanning input can be I/O ports and analog inputs, An, switched to the function of digital inputs. If I/O ports are used for inputs, wake-up by input changes is possible. The select line(s) of the interesting inputs (keys, gates etc.) are set high and the interrupt(s) are enabled for the desired signal edges. If one of the desired input signal changes occurs, an interrupt is given and wake-up takes place.

Figure 5–13 shows a keyboard with 16 keys.
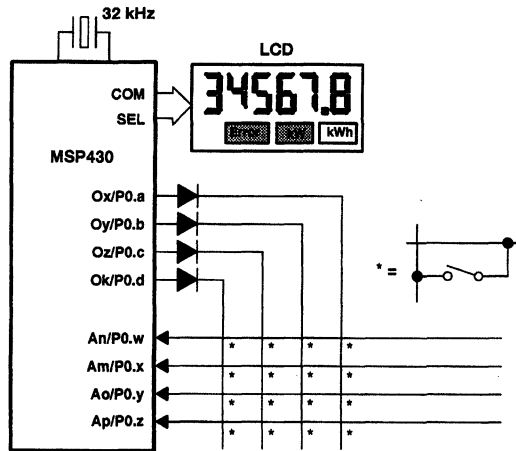
Figure 5–13. Keyboard Connection to MSP430

Figure 5–14 shows some possibilities for connecting external signals to the MSP430:

❑ The first row contains keys. The decoupling diode in the row selection line prevents a pressed key from shorten other signals. If more than one key can be activated simultaneously, then all keys need to have a decoupling diode.

❑ The second row contains diodes. This is a simple way to identify the version used to a system.

❑ The third row selects digital signals coming from peripherals with outputs that can be switched to high-impedance mode.

❑ The fourth row uses an analog switch to connect digital signals to the MSP430. The output of a CMOS gate and the output of a comparator are shown.

The rows containing keys need to be debounced. If a change is seen at these inputs, the information is read in and stored. A second read is made after 10 ms to 100 ms, and the information read is compared to the first one. If both reads are equal, the information is used. Otherwise, the procedure is repeated. The basic timer can be used for this purpose.
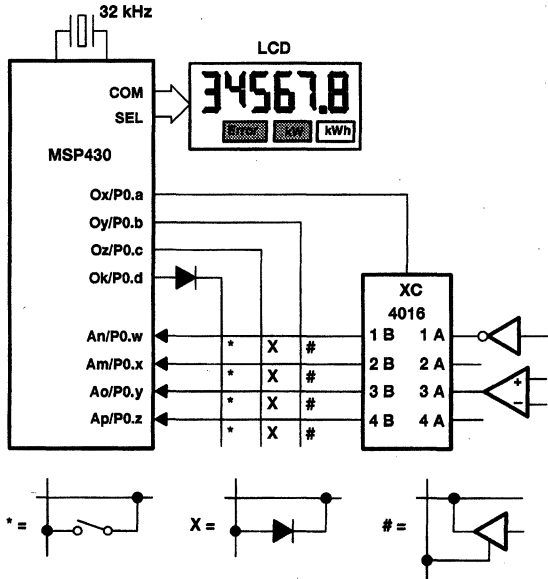
Figure 5–14. Connection of Different Input Signals

5.5.6 Temperature Calculations for Sensors

Several sensors can be connected to the MSP430. Chapter 2, *The Analog-to-Digital Converters*, describes the different possible ways of doing this. Independent of the ADC or sensor type used, a binary number N is finally delivered from the ADC that represents the measured value K:

$$K = f(N)$$

where:

K   Measured value (temperature, pressure etc.)

N   Result of ADC

The function f(N) is normally non-linear for sensors, and, therefore, a calculation is needed to get the measured value K. The linearization of sensors by resistors is described in Section 4.7.1, *Sensor Connection and Linearization*.

Two methods of how to represent the function, f(N), are described:

❑ Table processing

❑ Algorithms (linear, quadratic, cubic or hyperbolic equations)

5.5.6.1 Table Processing for Sensor Calculations

The ADC measurement range used is divided into parts, each of them having a length of 2^M bits. For any multiple of 2^M the output value K is calculated and stored in a one-dimensional table.

This table is used for linear interpolation to get the values for ADC results between two table values. Figure 5–15 shows such a non-linear sensor characteristic.
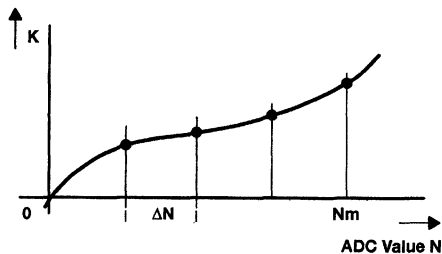
Figure 5–15. Nonlinear Function K = f(N)

Steps for the development of a sensor table:

1) Definition of the external circuitry used at the ADC inputs (see Chapter 2, *The Analog-to-Digital Converters*)

2) Definition of the output format of the table contents (bits after decimal point, M.N)

3) Calculation of the voltage at the analog input Ax for equally spaced (ΔN) ADC values n

4) Calculation of the sensor resistances for the previous calculated analog input voltages

5) Calculation of the input values K (temperature, pressure etc.) that cause these sensor resistances

6) Insertion of the calculated input values K in the format defined with 2. into the table

EXAMPLE: A sensor characteristic is described in a table TABLE. The ADC results are divided in distances $\Delta N = 128$ starting at value N0 = 256. The output value K is content of this table. The ADC result is corrected with offset and slope coming from the calibration procedure.

```
;
        .BSS    OFFSET,2                ; Offset from calibration 10.0
        .BSS    SLOPE,2                 ; Slope from calibration   1.10
DN      .EQU    128                     ; Delta N
;
; Table contains signed values. The decimal point may be anywhere
;
TABLE   .WORD   02345h, ...,00F3h ; K0, K1, ...KM
;
TABCAL1 MOV     &ADAT,IROP1             ; ADC result N to IROP1        14.0
        ADD     OFFSET,IROP1            ; Correct offset              10.0
        MOV     SLOPE,IROP2L            ; Slope                       1.10
        CALL    #MPYS                   ; (ADC+OFFSET)xSLOPE          15.10
;
; Corrected ADC value in IRACM|IRACL.
;
        CALL    #SHFTLS6                ; Result to IRACM             15.0
        MOV     IRACM,XIN               ; Copy it
```

```
;
; Calculation of NA address. One less adaptation due to
; word table (2 bytes/item).
;
        MOV     XIN,IROP1           ; N -> Multiplicand          15.0
        SWPB    IROP1               ; Adapt to deltaN = 128      14.0
        BIC.B   #1,IROP1            ; Even word address needed    8.0
        SUB     #2,IROP1            ; Adapt to N0 = 256 (2 x deltaN)
        MOV     TABLE(IROP1),R15    ; KA from table
        MOV     TABLE+2(IROP1),R14  ; KA+1 from table
;
; K = ((XIN-KA)/deltaN) x (KA+1 - KA) + KA
;
        SUB     R15,XIN             ; XIN - KA
        MOV     R14,IROP2L          ; KA+1
        SUB     R15,IROP2L          ; KA+1 - KA)
        MOV     XIN,IROP1           ; XIN - KA
        CALL    #MPYS               ; (XIN - KA) x (KA+1 - KA)
        CALL    #SHFTRS6            ; /deltaN
        CALL    #SHFTRS1            ; deltaN = 2^7
        ADD     R15,IRACL           ; + KA, result in IRACL
        RET
```

5.5.6.2 Algorithms for Sensor Calculations

If the sensor characteristic can be described by a function, K = f(N), then no table processing is necessary. The value K can be calculated out of the ADC result N. The coefficients a_n and b_n can be found with PC computer software (e.g. MATHCAD), with formulas by hand, or by the MSP430 itself. These categories are for example:

Linear Equation

$$K \;=\; a_1 \times N + a_0$$

Quadratic Equation

$$K \;=\; a_2 \times N^2 + a_1 \times N + a_0$$

Cubic Equation

$$K = a_3 \times N^3 + a_2 \times N^2 + a_1 \times N + a_0$$

Root Equation

$$K = a_0 \pm \sqrt{b_1 \times N + b_0}$$

Hyperbolic Equation

$$K = \frac{b_1}{N + b_0} + a_0$$

Steps for the development of a sensor algorithm:

1) Definition of the hardware circuitry used at the ADC inputs (See Chapter 2, *The ADC*, for the different possibilities)

2) Definition of the format of the algorithm (floating point: 2- or 3-word package, integer software: bits after decimal point M.N)

3) Definition of a value for K to be measured (temperature, pressure etc.)

4) Calculation of the nominal sensor resistance for the previous chosen value of K

5) Calculation of the voltage at the analog input Ax for this sensor resistance (See Chapter 2, *The ADC*, for the formulas used with the different circuits)

6) Calculation of the ADC result N for this input voltage at analog input Ax

7) Repetition of steps 3 to 6 depending on the algorithm used: twice for linear equations, three times for quadratic, hyperbolic and root equations, four times for cubic equations.

8) Decision of the sensor characteristic: look for best suited equation.

9) Calculation of the coefficients a_n and b_n out of the calculated pairs of values K_n and the ADC result N_n. See Section 5.5.6.3, *Coefficient Calculation for the Equations*.

EXAMPLE: A quadratic behavior is given for a sensor characteristic:

$$K = a_2 \times N^2 + a_1 \times N + a_0$$

with N representing the ADC result. The corrected ADC result (see the previous text) is stored in XIN. The three terms are stored in the ROM locations A2, A1 and A0.

```
;
A2          .WORD       07FE3h                ; Quadratic term              +-0.14
A1          .WORD       00346h                ; Linear term                 +-0.14
A0          .WORD       01234h                ; Constant term               +-15.0
;
QUADR       MOV         XIN,IROP1         ; Corrected ADC result        14.0
            MOV         A2,IROP2L         ; Factor A2                   +-0.14
            CALL        #MPY                  ; XIN x A2                14.14
            ADD         A1,IRACL          ; (XIN x A2) + A1             +-0.14
            ADC         IRACM                 ; Carry to HI reg
            CALL        #SHFTL3               ; To IRACM                14.1
            MOV         IRACM,IROP2L      ; (XIN x A2) + A1 -> IROP2L   14.1
            CALL        #MPYS                 ; (XIN x A2) + A1) x XIN  28.1
            CALL        #SHFTL2               ; Result to IRACM         15.0
            ADD         A0,IRACM          ; Add A0                      15.0
;
; The signed 16-bit result is located in IRACM.
;
            RET
```

The Horner-scheme used above can be expanded to any level. It is only necessary to shift the multiplication results to the right to ensure that the numbers always fit into the 32-bit result buffer IRACM and IRACL. The terms A2, A1, A0 can also be located in RAM.

If lots of calculations need to be done, then the use of the floating point package should be considered. See Section 5.6, *The Floating Point Package*, for details.

5.5.6.3 Coefficient Calculation for the Equations

With two pairs (linear equation), three pairs (quadratic, hyperbolic and root equations), or four pairs (cubic equations) of K_n and N_n, the coefficients a_n and b_n can be calculated. The formulas are shown in the following.

where:

K_n Calculation result for the ADC result N_n
(e.g. temperature, pressure)

N_n Input value for the calculation e.g. ADC result

a_n Coefficient for the N^n value of the polynoms

b_n Coefficients for the hyperbolic and root equations

Linear equation:

$$K \;=\; a_1 \times N + a_0$$

$$a_1 \;=\; \frac{K_2 - K_1}{N_2 - N_1} \qquad a_0 \;=\; \frac{K_1 \times N_2 - K_2 \times N_1}{N_2 - N_1}$$

Quadratic Equation:

$$K \;=\; a_2 \times N^2 + a_1 \times N + a_0$$

$$a_2 \;=\; \frac{\left(K_2 - K_1\right) - a_1 \times \left(N_2 - N_1\right)}{N_2^2 - N_1^2} \qquad a_0 \;=\; K_1 - a_2 \times N_1^2 - a_1 \times N_1$$

$$a_1 \;=\; \frac{\left(K_2 - K_1\right)\times\left(N_3^2 - N_2^2\right) - \left(K_3 - K_2\right)\times\left(N_2^2 - N_1^2\right)}{\left(N_2 - N_1\right)\times\left(N_3^2 - N_2^2\right) - \left(N_3 - N_2\right)\times\left(N_2^2 - N_1^2\right)}$$

Cubic Equation:

$$K \;=\; a_3 \times N^3 + a_2 \times N^2 + a_1 \times N + a_0$$

The equations for the four coefficients a_n are too complex. Shift the calculation task to the calibration PC and use MATHCAD or something similar to it.

Root Equation:

$$K \;=\; a_0 \pm \sqrt{b_1 \times N + b_0}$$

$$a_0 \;=\; 0.5 \times \frac{\left(N_2 - N_1\right)\times\left(K_3^2 - K_1^2\right) - \left(N_3 - N_1\right)\times\left(K_2^2 - K_1^2\right)}{\left(N_2 - N_1\right)\times\left(K_3 - K_1\right) - \left(N_3 - N_1\right)\times\left(K_2 - K_1\right)}$$

$$b_1 \;=\; \frac{\left(K_2^2 - K_1^2\right) - 2a_0 \times\left(K_2 - K_1\right)}{N_2 - N_1} \qquad b_0 \;=\; \left(K_1 - a_0\right)^2 - b_1 \times N_1$$

Hyperbolic Equation:

$$K = \frac{b_1}{N + b_0} + a_0$$

$$b_0 = \frac{(N_3 \times K_3 - N_1 K_1) \times (N_2 - N_1) - (N_2 \times K_2 - N_1 \times K_1) \times (N_3 - N_1)}{(N_3 - N_1) \times (K_2 - K_1) - (N_2 - N_1) \times (K_3 - K_1)}$$

$$b_1 = (K_1 - a_0) \times (N_1 + b_0)$$

$$a_0 = \frac{(N_3 \times K_3 - N_1 K_1) + b_0 \times (K_3 - K_1)}{N_3 - N_1}$$

EXAMPLE: the sensor used has a quadratic characteristic $R = d_2 \times K^2 + d_1 \times K + d_0$. This means, the value K is described best by the root equation (inverse to the quadratic characteristic of the sensor):

$$K = a_0 \pm \sqrt{b_1 \times N + b_0}$$

where the sensor resistance R is replaced by the ADC result N. During the calibration with the values for K_n 0, 200, and 400, the following ADC Results, N_n, were measured:

Calc.Value K_n (°C, hP, V)		ADC Value N_n	
K_1	0	N_1	4196
K_2	200	N_2	4430
K_3	400	N_3	4652

With the previous numbers the coefficients a_0, b_0 and b_1 can be calculated:

$$a_0 = 0.5 \times \frac{(4430 - 4196) \times (400^2 - 0^2) - (4652 - 4196) \times (200^2 - 0^2)}{(4430 - 4196) \times (400 - 0) - (4652 - 4196) \times (200 - 0)} = 4000$$

$$b_1 = \frac{(200^2 - 0^2) - 2 \times 4000 \times (200 - 0)}{4430 - 4196} = -6666.6667$$

$$b_0 = (0 - 4000)^2 - (-6666.6667) \times 4196 = 43.97371E6$$

With the above calculated coefficients, the negative root value is to be used:

$$K = a_0 - \sqrt{b_1 \times N + b_0}$$

5.5.7 Data Security Applications

If consumption data is transmitted via telephone lines or sent by RF then it is normally necessary to encrypt this data to make it completely unreadable. For these purposes the DES (Data Encryption Standard) is used more and more, and is becoming the standard in Europe also. The next two sections show how to implement the algorithms of this standard and how the encrypted data can be sent by the MSP430.

5.5.7.1 Data Encryption Standard (DES) Routines

The DES works on blocks of 64 bits. These blocks are modified in several steps and the output is also a block with 64 totally-scrambled bits. It is not the intention of this section to show the complete DES algorithm. Instead, a subroutine is shown that is able to do all of the necessary permutations in a very short time. The subroutine mentioned can do the following permutations (the tables mentioned refer to the booklet *Data Encryption Algorithm* from ANSI):

❏ Initial Permutation: 64-bits plain text to 64-bit encrypted text via table IP

❏ 32-bit to 48-bit permutation via table E

❏ 48-bit to 32-bit permutation via tables S1 to S8

❏ 32-bit to 32-bit permutation via table P

❏ Inverse initial Permutation: 64-bits to 64-bit via table IP^{-1}

The permutation subroutine is written in a code and time optimized manner to get the highest data throughput with the lowest ROM space requirements.

For each kind of permutation a description table is necessary that contains the following information for every bit to be permuted:

7		5	3	2	0
Rep. Bit	EOT	Byte Index		Bit Position	

Where:

Rep. Bit	Repetition Bit: The actual bit is contained twice in the output table. The next byte (with Rep. = 0) contains the address for the second insertion. This bit is only used during the 32-bit to 48-bit permutation
EOT	End of Table Bit: This bit is set in the last byte of a permutation table

Byte Index   The byte address 0 to 7 inside the output block
Bit Position   The bit address 0 to 7 inside the output byte

The following figure shows the permutation of bit i. The description table contains at address i the information:

Repetition Bit = 0: The bit i is to be inserted into the output table only once
EOT = 0   Bit i is not the last bit in the description table
Byte Index = 3:   The relative byte address inside the output table is 3 (PTOUT+3)
Bit Position = 5:   The bit position inside the output byte is 5 (020h)

Description Table   **Input Table**   **Output Table**
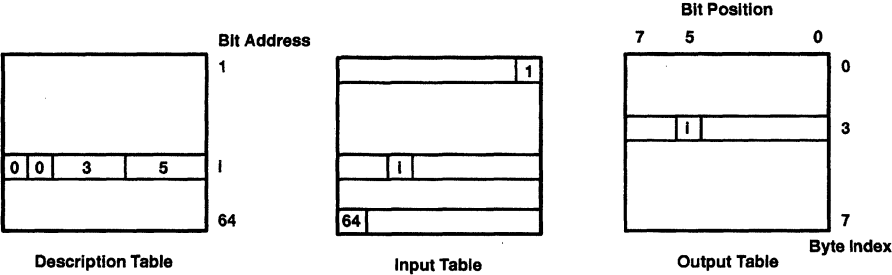
Figure 5–16. DES Encryption Subroutine

Note:

The bit numbers used in the DES specification range from 1 to 64. The MSP430 subroutines use addresses from 0 to 63 due to the computer architecture.

The software subroutines for the previously described permutations follow. The subroutines PERMUT and PERM_BIT are used for all necessary permutations (see previous). The subroutines shown have the following needs:

❏ The initialization of the subroutine PERMUT decides which permutation takes place. The address of the actual description table is written to pointer register PTPOI.

❏ Permutations are always made from table PTIN (input table) to table PTOUT (output table).

❏ Only 1s are processed during the permutation. This saves 50% of processing time. The output buffer is therefore cleared initially by the PERMUT subroutine.

      ❏ The output buffer must start with an even address (word instructions are used for clearing)

```
; Main loop for a permutation run. Tables with up to 64 bits are
; permuted to other tables.
;
; Definitions for the permutation software
;
PTPOI    .EQU    R6                      ; Pointer to description table
PTBYTP   .EQU    R7                      ; Byte index input table
PTBITC   .EQU    R8                      ; Bit counter inside input byte
         .BSS    PTIN,8                  ; Input table 64 bits
         .BSS    PTOUT,8                 ; Output table 64 bits
EOT      .EQU    040h                    ; End of table indication bit
REP      .EQU    080h                    ; Repetition bit
;
; Call for the "Initial Permutation". Description table is
; starting at label IP (64 bytes for 64 bits).
;        ...
         MOV     #IP,PTPOI               ; Load description table pointer
         CALL    #PERMUT                 ; Process Initial Permutation
;        ...
;
; Permutation subroutine. Table PTIN is permuted to table PTOUT
;
PERMUT   CLR     PTBYTP                  ; Clear byte index input table
         CLR     PTOUT                   ; Clear output table 8 bytes
         CLR     PTOUT+2
         CLR     PTOUT+4
         CLR     PTOUT+6
PERML    CLR     PTBITC                  ; Bit counter (bits inside byte)
L$502    RRA.B   PTIN(PTBYTP)            ; Next input bit to Carry
         JNC     L$500                   ; If bit = 0: No activity nec.
L$501    CALL    #PERM_BIT               ; Bit = 1: Insert bit to output
L$500    INC     PTPOI                   ; Incr. description table pointer
         TST.B   -1(PTPOI)               ; REP bit set for last bit?
         JN      L$501                   ; Yes, process 2nd output bit
```

```
;
; One input table bit is processed. Check if byte limit reached
;
        INC.B   PTBITC                  ; Incr. bit counter
        CMP.B   #8,PTBITC               ; Bit 8 (outside byte) reached?
        JLO     L$502
        INC.B   PTBYTP                  ; Yes, address next byte
        BIT.B   #EOT,-1(PTPOI)          ; End of desc. table reached?
        JZ      PERML                   ; No, proceed with next byte
        RET
;
; Permutation subroutine for one bit: A set bit of the input is
; set in the output depending on the information of a
; description table pointed too by pointer PTPOI
; 20 cycles + CALL (5 cycles)
;
PERM_BIT .EQU   $
        MOV.B   @PTPOI,R4               ; Fetch description word
        MOV     R4,R5                   ; Copy it
        BIC.B   #REP+EOT,R4             ; Clear Repetition bit and EOT
        RRA.B   R4                      ; Move Index Bits to LSBs
        RRA.B   R4                      ; to form byte index to PTBIT
        RRA.B   R4
        AND.B   #07h,R5                 ; Mask out index for output table
        BIS.B   PTBIT(R5),PTOUT(R4)     ; Set bit in output table
        RET
;
PTBIT   .BYTE   1,2,4,8,10h,20h,40h,80h ; Bit table
;
; Description Table for the Initial Permutation. 64 bits of
; the input table are permuted to 64 bits in the output table
; (IP-1 table contains these numbers)
;
IP      .BYTE   40-1                    ; Bit 1 -> position 40
        .BYTE   8-1                     ; Bit 2 -> position 8
;       ...
```

```
            .BYTE     EOT+25-1          ; Bit 64 -> pos. 25, End of table
;
; Description Table for the Expansion Function E. 32 bits of
; the input table are permuted to 48 bits in the output table
;
E           .BYTE     REP+2-1           ; Bit 1 -> position 2 and 48
            .BYTE     48-1              ; Bit 1 -> position 48
            .BYTE     3-1               ; Bit 2 -> pos. 3
;           . . .
            .BYTE     REP+1-1           ; Bit 32 -> position 1 and 47
            .BYTE     EOT+47-1          ; Bit 32 -> pos. 47, End of table
```

Processing time for a 64-bit block: The most time consuming parts for the encryption are the permutations. All other operations are simple moves or exclusive ORs (XOR). This means that the number of permutations multiplied with the number of cycles per bit gives an estimation of the processing time needed. Every bit needs 43 cycles to be permuted.

The necessary number of permutations is:

1)	Initial Permutation	64
2)	32-bit to 48-bit permutation	16×48
3)	48-bit to 32-bit permutation	16×32
4)	32-bit to 32-bit permutation	16×32
5)	Inverse initial Permutation:	64
6)	Key permutations choice 1	56
7)	Key permutations choice 2	16×48

Sum of permutations   2744

Number of cycles typically $(2744 \times 43 \times 0.5)$ 58996 cycles 32 ones in block

maximum (2744×43) 117992 cycles 64 ones in block

For a block with 64 bits approximately 59 ms are needed with an MCLK of 1 MHz.

ROM space: The needed ROM space can be divided into the following parts:

1)	Main program (approx.)	400 bytes
2)	Subroutines	100 bytes
3)	Tables for permutations	570 bytes

Sum of bytes   1070 bytes

The complete DES encryption software fits into 1K bytes.

5.5.7.2   Output Sequence for 19.2-kHz Biphase Space Code

The encrypted information is normally output with a Biphase Code. Figure 5–17 shows such a modulation. At the beginning of a bit, a level change occurs. A zero bit has an additional level change in the middle of the bit, a one bit has the same information during the whole bit.
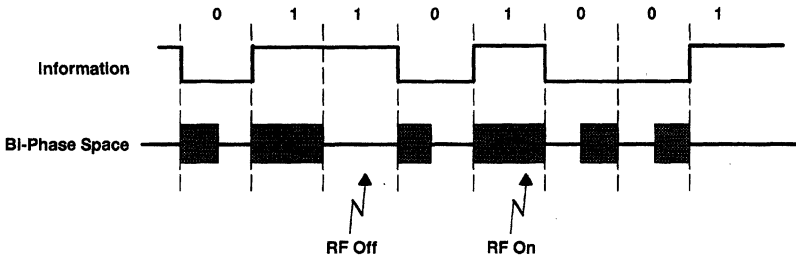
Figure 5–17. Biphase Space Code

The output sequence is written for P0.4 (as shown in Section 4.4, *Heat Allocation Counter*). This means that no constant of the constant generator can be used. If P0.0, P0.1, P0.2, or P0.3 are used, the instructions that address the ports are one cycle shorter and the delay subroutines have to be adapted.

The following output sequence is written with counted instructions per bit due to the normal use of batteries (Vcc = 3V) for these applications. This means the maximum MCLK is 2.2 MHz. If the supply voltage is 5 V, MCLK frequencies up to 3.3 MHz are possible. These high operating frequencies allow the use of interrupt driven output sequences.

The interrupt approach makes strict real time programming necessary. Any interrupt handler must be interruptible (EINT is one of the first instructions of any interrupt handler). Hardware examples are shown in Section 4.8, *RF Readout*.

```
; OUT192 OUTPUTS THE RAM STARTING AT "RAMSTART" BITWISE
; IN BI-PHASE-CODE. EVERY 040h ADDRESSES A SCAN IS MADE
; TO READ P0.1 WHERE THE WATER FLOW COUNTER IS LOCATED. THE
; 4 SCAN RESULTS ARE ON THE STACK AFTER RETURN FOR CHECKS
; NOPs ARE INCLUDED TO ENSURE EQUAL LENGTH OF EACH BRANCH.
```

```
; All interrupts must be disabled during this output subroutine!
; CALL #NOPx MEANS x CYCLES OF DELAY. MCLK = 1MHz
;
OUTPUT   .EQU    010h                    ; P0.4:
PORT     .EQU    011h                    ; PORT0
RAMSTART .EQU    0200h                   ; Start of output info
RAMEND   .EQU    0300h                   ; End of output info
SCAND    .EQU    040h                    ; Scan delta (addresses)
Rw       .EQU    R15                     ; Register allocation
Rx       .EQU    R14
Ry       .EQU    R13
Rz       .EQU    R12
;
OUT192   BIC.B   #OUTPUT,&PORT           ; Reset output port
         MOV     #RAMSTART,Ry            ; WORD POINTER
         MOV     #RAMSTART+SCAND,Rw      ; NEXT SCAN ADDRESS
; FETCH NEXT WORD AND OUTPUT IT                                          CYCLES
WORDLP   MOV     #16,Rz                  ; BIT COUNTER            2
         MOV     @Ry,Rx                  ; FETCH WORD             5
; OUTPUT NEXT BIT: Change output state
BITLOP   XOR.B   #OUTPUT,&PORT           ; CHANGE OUTPUT PORT     5
;
; CHECK IF NEXT SCAN OF WATER FLOW IS NECESSARY: Ry >= Rw
;
         CMP     Rw,Ry                   ;                        1
         JHS     SCAN                    ; YES                    2
         NOP                             ; NO                     5
         NOP
         NOP
         NOP
         NOP
         JMP     BITT                    ;                        2
SCAN     ADD     #SCAND,Rw               ; NEXT SCAN ADDRESS      2
         PUSH    &PORT                   ; PUSH INFO OF PORT      5
;
BITT     RRC     Rx                      ; NEXT BIT TO CARRY      1
```

```
        JNC     OUT0                    ; BIT = 0                    2
;
; BIT = 1: OUTPUT PORT IS CHANGED IN THE MIDDLE OF BIT
;
        CALL    #NOP9                   ;                            9
        XOR.B   #OUTPUT,&PORT           ; CHANGE OUTPUT PORT         5
        JMP     CHECK                   ;                            2
;
; BIT = 0: OUTPUT PORT STAYS DURING COMPLETE BIT
;
OUT0    CALL    #NOP16                  ; OUTPUT STAYS HI            16
;
; END OF LOOP: CHECK IF COMPLETE WORD OR END OF INFO
;
CHECK   DEC     Rz                      ; 16 BITS OUTPUT?            1
        JZ      L$1                     ; YES                        2
        CALL    #NOP15                  ; NO, NEXT BIT               15
        JMP     BITLOP                  ;                            2
;
; COMPLETE WORD OUTPUT: ADDRESS NEXT WORD
L$1     ADD     #2,Ry                   ; POINTER TO NEXT WORD       2
        CMP     #RAMEND,Ry              ; RAM OUTPUT?                2
        JEQ     COMPLET         ; YES  ;                            2
        NOP                             ; NO, NEXT WORD              2
        NOP
        JMP     WORDLP                  ;                            2
;
COMPLET ....                            ; 4 SCANS ON STACK
; NOP Subroutines: The Subroutine inserts defined numbers of
; cycles when called. The number xx of the called label defines
; the number of cycles including CALL (5 cycles) and RET
;
NOP16   NOP                             ; CALL #NOPxx needs 5 cycles
NOP15   NOP
NOP14   NOP
NOP13   NOP
```

```
NOP12    NOP
NOP11    NOP
NOP10    NOP
NOP9     NOP
NOP8     RET                        ; RET needs 3 cycles
```

5.5.8 Status/Input Matrix

A few subroutines are described that handle the inputs coming from keys, signals etc. They check, if the inputs are valid, for the given status of the program.

5.5.8.1 Matrix with Few Valid Combinations

The following subroutine checks if for a given program status an input (e.g., via the keyboard) is valid or not; and, if valid, which response is necessary. This solution is recommended if only few valid combinations exist out of a large possible number (see Figure 5–18).
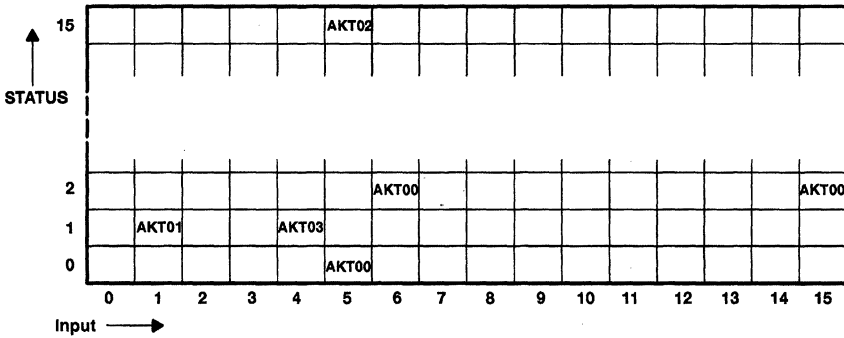
Figure 5–18. Matrix for Few Valid Combinations

❑ Call

■ Input number in R5

■ Status in R4

❑ Return

■ R4 = 0: Input not valid (not included in the table)

■ R4 # 0: task number in R4

```
CALL      MOV     STATUS,R4              ; Program status to R4
          MOV     INPUT,R5              ; New input to R5
          CALL    #STIMTRX             ; Check validity
          ...                           ; R4 contains info
;
STIMTRX   MOV.B   STTAB(R4),R4         ; Start of table for status
          ADD     #STTAB,R4            ; Table address to R4
L$3       CMP.B   @R4+,R5             ; New input included in table?
          JEQ     L$2                  ; Yes, output it
          INC     R4                   ; No, skip response byte
          TST.B   0(R4)               ; End of status table? (0)
          JNE     L$3                  ; No, try next input
          CLR     R4                   ; Yes, end of table reached
          RET                          ; Input invalid, return with R4 = 0
L$2       MOV.B   @R4,R4              ; Input valid, return with task
          RET                          ; number in R4
;
; Table with relative start addresses for the status tables
;
STTAB     .BYTE   ST0-STTAB,ST1-STTAB,ST2-STTAB,...ST15-STTAB
;
; Status tables: valid inputs,response,..,0  (up to 15 inputs)
;
ST0       .BYTE   IN5,AKT00,0          ; Status 0 table
ST1       .BYTE   IN1,AKT01,IN4,AKT03,0 ; Status 1 table
ST2       .BYTE   IN15,AKT00,IN6,AKT06,0 ; Status 2 table
          ....                          ; Status 3 to 14
ST15      .BYTE   IN5,AKT02,0          ; Status 15 table
```

With a small change, the task to do is also executed within the subroutine:

```
CALL      MOV     STATUS,R4              ; Program status to R4
          MOV     INPUT,R5              ; New input to R5
          CALL    #STIMTRX             ; Check validity and execute task
          ...                           ; R4 = 0: invalid input
;
STIMTRX   MOV.B   STTAB(R4),R4         ; Start of table for status
```

```
           ADD       #STTAB,R4                    ; to R4
L$3        CMP.B     @R4+,R5                      ; New input included?
           JEQ       L$2                          ; Yes, proceed
           INC       R4                           ; No, skip task address
           TST.B     0(R4)                        ; End of status table? (0)
           JNE       L$3                          ; No, try next input
           CLR       R4                           ; End of table reached
           RET                                    ; Input invalid, return with R4 = 0
L$2        MOV.B     @R4,R4                        ; Input valid, go to task
           ADD       R4,PC                        ; offset to AKT00 in R4
AKT00      ...                                    ; Task 00
           RET
AKT01      ...                                    ; Task 01
           RET
AKT06      ...                                    ; Task 06
           RET
AKT03      ...                                    ; Task 03
           RET
;
; Table with relative start addresses for the status tables
;
STTAB      .BYTE     ST1-STTAB,ST2-STTAB,...ST15-STTAB
;
; Status tables: valid inputs,task-table_start,0
;
ST1        .BYTE     IN5,AKT00-AKT00,0            ; Status 1 table
ST2        .BYTE     IN1,AKT01-AKT00,IN4, AKT03-AKT00,0
ST3        .BYTE     IN15,AKT00-AKT00,IN6,AKT06-AKT00,0
           ....                                   ; Status 4 to 14
ST15       .BYTE     IN5,AKT02-AKT00,0            ; Status 15 table
```

5.5.8.2 Matrix With Valid Combinations Only

The following subroutine executes the tasks belonging to the 16 possible STA-TUS/INPUT combinations. The handler start addresses must be within 254 bytes relative to the label STTAB. The number of combinations can be enlarged to any value.

❑ Call

■ Input number in RAM byte INPUT (four possibilities 0 to 3)

■ Program status in RAM byte STATUS (four possibilities 0 to 3)

❑ Return

■ No information returned

```
CALL      CALL    #STIMTRX                  ; Execute task for input
;
STIMTRX   MOV.B   STATUS,R4                 ; Program status 00xx
          MOV.B   INPUT,R5                  ; Input (key, Intrpt,) 0yy
          RLA     R4                        ; STATUS x 4: 00xx -> 0xx0
          RLA     R4                        ; 0xx0 -> 0xx00
          ADD     R5,R4                     ; Build table offset: 0xxyy
          MOV.B   STTAB(R4),R4              ; Offset of Start of table
          ADD     R4,PC                     ; Handler start to PC
STTAB     .BYTE   AKT00-STTAB               ; Action STATUS = 0, INPUT = 0
          .BYTE   AKT01-STTAB               ; Action STATUS = 0, INPUT = 1
          .BYTE   AKT02-STTAB,AKT03-STTAB,AKT04-STTAB,AKT05-STTAB
          ...
          .BYTE   AKT12-STTAB,AKT13-STTAB,AKT14-STTAB,AKT15-STTAB
;
; Action handlers for the 16 STATUS/INPUT xy combinations
;
AKT00     ...                               ; Handler for task 0,0
          RET
AKT01     ...                               ; Handler for task 0,1
          RET
          ...                               ; Tasks 02 to 31
AKT32     ...                               ; Handler for task 3,2
          RET
AKT33     ...                               ; Handler for task 3,3
          RET
```

The next subroutine also executes the tasks belonging to the 16 possible STATUS/INPUT combinations. Here the handler start addresses can be located in the complete 64K-byte address space. The number of STATUS/INPUT combinations can be enlarged to any value.

❑ Call

■ Input number in RAM byte INPUT (five possibilities 0 to 3)

■ Program status in RAM byte STATUS (four possibilities 0 to 3)

❑ Return

■ No information returned

```
CALL      CALL      #STIMTRX                    ; Execute task for input
;
STIMTRX   MOV       STATUS,R4                   ; Program status 00xx
          MOV       INPUT,R5                    ; Input (key, Intrpt) 0yy
          RLA       R4                          ; 00xx -> 0xx0
          RLA       R4                          ; 0xx0 -> 0xx00
          ADD       R5,R4                       ; 0xxyy table offset
          RLA       R4                          ; To word addresses
          MOV       STTAB(R4),PC                ; Offset of Start of table
;
STTAB     .WORD     AKT00                       ; Action STATUS = 0, INPUT = 0
          .WORD     AKT01                       ; Action STATUS = 0, INPUT = 1
          ...                                   ; Action handlers AKT02 to AKT32
          .WORD     AKT33                       ; Action STATUS = 3, INPUT = 3
```

5.6 The Floating-Point Package

Floating-point arithmetic is necessary if the range of the numbers used is very large. When using a floating-point package, it is normally not necessary to take care if the limits of the number range are exceeded. This is due to a number ratio of about 10^{78} if comparing the largest to the smallest possible number (remember: the number of smallest particles in the whole universe is estimated to 10^{84}). The disadvantages are the slower calculation speed and the ROM space needed.

A floating-point package with 24-bit and 40-bit mantissa exists for the MSP430. The number range, resolution, and error indication are explained as well as the conversion subroutines used as the interface to binary and binary-coded-decimal (BCD) numbers. Examples are given for many subroutines and applications, like the square root, are included in the software example chapter.

The floating-point package makes use of the RISC architecture of the MSP430 family. During the initialization of the subroutines, the arguments are copied into registers R4 to R15 and the complete calculations take place there. After the completion of the calculation, the result is placed on top of the stack.

5.6.1 General

The floating-point package (FPP) consists of 3 files supporting the .FLOAT format (32 bits) and the .DOUBLE format (48 bits):

❏ FPPDEF4.ASM: the definitions used with the other two files

❏ FPP04.ASM: the basic arithmetic operations add, subtract, multiply, divide and compare

❏ CNV04.ASM: the conversions from and to the binary and the BCD format

---

Notes:

The file FPP04.ASM can be used without the conversions, but the conversion subroutines CNV04.ASM need the FPP04.ASM file. This is due to the common completion parts contained in FPP04.ASM.

The explanations given for the FPP version 04 are valid also for the FPP version 03. The only difference between the two versions is the hardware multiplier that is included in the version 04. Other differences are mentioned in the ajoining sections. FPP4 is upward compatible to FPP3.

The assembly time variable DOUBLE defines which format is to be used:

DOUBLE = 0:   Two word format .FLOAT with 24-bit mantissa

DOUBLE = 1:   Three word format .DOUBLE with 40-bit mantissa

The assembly time variable SW_UFLOW defines the reaction after a software underflow:

SW_UFLOW = 0: Software underflow (result is zero) is not treated as an error

SW_UFLOW = 1: Software underflow is treated as an error (N is set)

The assembly time variable HW_MPY defines if the hardware multiplier is used or not during the multiplication subroutine:

HW_MPY = 0:   No use, the multiplication is made by a software loop

HW_MPY = 1:   The 16×16 bit hardware multiplier is used

The FPP supports the four basic arithmetic operations, comparison, conversion subroutines and two register save/restore functions:

FLT_ADD	Addition
FLT_SUB	Subtraction
FLT_MUL	Multiplication
FLT_DIV	Division
FLT_CMP	Comparison
FLT_SAV	Saving of all used registers on the stack
FLT_REC	Restoring of all used registers from the stack
CNV_BINxxx	Binary to floating point conversions
CNV_BCD_FP	BCD to floating point conversion
CNV_FP_BIN	Floating point to binary conversion
CNV_FP_BCD	Floating point to BCD conversion

5.6.2   Common Conventions

The use of registers containing the addresses of the arguments saves time and memory space. The arguments are not affected by the operations and can be located either in ROM or RAM. Before the call for an operation, the two pointers RPARG and RPRES are loaded with the address(es) of the most significant word MSW of the argument(s). After the return from the call, both pointers and the stack pointer, SP, point to the result (on the stack) for an easy continuation of arithmetical expressions.

Note:

The result of a floating point operation is always written to the address the stack pointer (SP) points to when the subroutine is called. The address contained in register RPRES is used only for the addressing of Argument 1.

The results of the basic arithmetic operations (add, subtract, multiply and divide) are also contained in the RAM address @SP or 0(SP), and the registers RESULT_MID and RESULT_LSB after the return from these subroutines. Using these registers for data transfers saves program space and execution time.

Between FPP subroutine calls, all registers can be used freely. The result of the last operation is stored on the stack. See previous note.

If, at an intermediate stage of the basic arithmetic operations, a renormalization shift of one or more bit positions to the left is required, then valid bits are available for the shift into the low-order bit positions during renormalization. These bits are named guard bits. With some other FPPs having no guard bits, zeroes are shifted in, which means a loss of accuracy.

The registers that hold the pointers are called:

☐ RPRES   Pointer to Argument 1 and Result

☐ RPARG   Pointer to Argument 2 and Result

The following choices can be used to address the two operands:

1) $RESULT_{NEW} = @(RPRES)$ <operator> $@(RPARG)$
2) $RESULT_{NEW} = @(RPRES)$ <operator> $RESULT_{OLD}$
3) $RESULT_{NEW} = RESULT_{OLD}$ <operator> $@(RPARG)$

☐ To 1: RPRES and RPARG both point to the arguments for the next operation. This is the default and is independent of the address pointed to either a new argument or a result. The result of the operation is written to the address in the SP.

☐ To 2: RPRES points to the argument 1, RPARG still points to the result of the last operation residing on the top of the stack (TOS). This calling form allows the operations (argument 2 – result) and (argument 2 / result).

☐ To 3: RPARG points to argument 2, RPRES still points to the result of the last operation residing on the top of the stack. This calling form allows the operations (result – argument 2) and (result / argument 2).

> **Note:**
>
> Formulas 2 and 3 are not equal, they allow use of the result on the TOS in two ways with division and subtraction. No time is needed and no ROM-consuming moves are necessary if the result is the divisor or the subtrahend for the next operation.

Common to these subroutines is:

1) The pointers RPARG and RPRES point to the addresses of the input numbers. They always point to the MSBs of these numbers.

2) The input numbers are not modified, except the last result on the stack, if it is used as an operand.

3) The result is located on the top of the stack (TOS), the stack pointer, RPARG, and RPRES point to the most significant word of the result

4) Every floating point number represents a valid value. No invalid combinations like *Not a Number, Denormalized Number,* or *Infinity* exist. In this way, the MSP430 FPP has a larger range than other FPPs and allows a higher speed with less memory used. This is because no unnecessary checks for invalid numbers are made.

5) Every floating point operation outputs a valid floating point number that can be used immediately by other operations.

6) If a result is too large (exceeds the number range), the signed maximum number is output. An error indication is given in this case (see Table 5–6, *Error Indication*).

7) The CPU registers used are modified within the FPP subroutines, but do not contain valid data after a return from the subroutine. This means, they can be used freely between the FPP subroutines for other purposes.

5.6.3 The Basic Arithmetic Operations

The FPP is designed for fast and memory saving calculations. So register instructions are ideally suited for this operation. A common save and recall routine for the registers used at the beginning and the end of an arithmetical expression is an additional option. The subroutines FLT_SAV and FLT_REC should be applied as shown in the following examples.

5.6.3.1 Addition

❑ FLT_ADD: The floating point number pointed to by the register RPARG is added to the floating point number pointed to by the register RPRES. The

25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding. It is added to the result.

RESULT on TOS = @(RPRES) + @(RPARG)

❑ Errors: Normal error handling. See Section 5.6.3.5, *Error Handling*, for a detailed description.

❑ Output: The floating point sum of the two arguments is placed on the top of the stack. The stack pointer points to the same location as it did before the subroutine call.

The stack pointer, RPRES, and RPARG point to the MSBs of the floating point sum. If an error occurred (N = 1 after return), the result is the number that best represents the correct result: 0 resp. $\pm 3.4 \times 10^{38}$.

❑ EXAMPLE: The floating point number (.FLOAT format) contained in the ROM starting at address NUMBER is added to the RAM location pointed to by R5. The result is written to the RAM addresses RES and RES+2 (LSBs).

```
DOUBLE   .EQU     0
         MOV      R5,RPRES          ; Address of Argument 1 in R5
         MOV      #NUMBER,RPARG     ; Address of Argument 2
         CALL     #FLT_ADD          ; Call add subroutine
         JN       ERR_HND           ; Error occurred, check reason
         MOV      @RPRES+,RES       ; Store FPP result (MSBs)
         MOV      @RPRES+,RES+2     ; LSBs
         ...                        ; Continue with program
```

5.6.3.2 Subtraction

❑ FLT_SUB: The floating point number pointed to by register RPARG is subtracted from the floating point number pointed to by register RPRES. With proper loading of the two input pointers, it is possible to calculate (Argument1 − Argument2) and (Argument2 − Argument1). The 25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding and is subtracted from the result.

RESULT on TOS = @(RPRES) − @(RPARG)

❑ Errors: Normal error handling. See Section 5.6.3.5, *Error Handling*, for a detailed description.

❑ Output: The floating point difference of the two arguments is placed on top of the stack. The stack pointer points to the same location as it did before the subroutine call.

The stack pointer, RPRES, and RPARG point to the MSBs of the floating point difference. If an error occurred (N = 1 after return), the result is the number that best represents the correct result; 0 resp. ±3.4 × 10^{38}.

❏ EXAMPLE: The floating point number (.DOUBLE format) contained in the ROM locations starting at address NUMBER is subtracted from RAM locations pointed to by R5. The result is written to the RAM addresses pointed to by R5.

```
DOUBLE    .EQU    1
          MOV     R5,RPRES            ; Address of Argument1 in R5
          MOV     #NUMBER,RPARG       ; Address of Argument2
          CALL    #FLT_SUB            ; ((R5)) - (NUMBER) -> TOS
          JN      ERR_HND             ; Error occurred, check reason
          MOV     @RPRES+,0(R5)       ; Store FPP result (MSBs)
          MOV     @RPRES+,2(R5)
          MOV     @RPRES,4(R5)        ; LSBs
          ...                         ; Continue with program
```

5.6.3.3 Multiplication

❏ FLT_MUL: The floating point number pointed to by the register RPARG is multiplied by the floating point number pointed to by the register RPRES. The 25th and 26th bit (41st and 42nd bit in case of DOUBLE format) of the calculated mantissa are used for rounding.

If a shift is necessary to get the MSB of the mantissa set then the LSB−1 is shifted into the mantissa and the LSB−2 is added to the result.

If the MSB of the mantissa is set, only the LSB−1 is added to the result. The multiplication subroutine returns the same result regardless of whether the hardware multiplier is used (HW_MPY = 1) or not (HW_MPY = 0).

RESULT on TOS = @(RPRES) × @(RPARG)

❏ Errors: Normal error handling. See Section 5.6.3.5, *Error Handling*, for a detailed description.

❏ Output: The floating point product of the two arguments is placed on the top of the stack. The stack pointer points to the same location as it did before the subroutine call.

The stack pointer, RPRES, and RPARG point to the MSBs of the floating point product. If an error occurred (N = 1 after return), the result is the number that best represents the correct result; 0 resp. ±3.4 × 10^{38}.

❏ Special Cases: 0 × 0 = 0   0 × X = 0   X × 0 = 0

☐ EXAMPLE: The result of the last operation, a floating point number (.FLOAT format) on the top of the stack, is multiplied by the constant π.

```
DOUBLE   .EQU    0

         MOV     #PI,RPARG       ; Address of constant PI

         CALL    #FLT_MUL        ; ((RPRES)) x (PI) -> TOS

         JN      ERR_HND         ; Error occurred, check reason

         ...                     ; Continue with program
PI       .FLOAT  3.1415926535    ; Constant PI
```

5.6.3.4 Division

☐ FLT_DIV: The floating point number pointed to by the register RPRES is divided by the floating point number pointed to by the register RPARG. With proper loading of the two input pointers, it is possible to calculate (Argument1 / Argument2) and (Argument2 / Argument1). The 25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding and is added to the result.

$$\text{RESULT on TOS} = \frac{@\,(\text{RPRES})}{@\,(\text{RPARG})}$$

☐ Errors: Normal error handling. See Section 5.6.3.5, *Error Handling,* for a detailed description. Division by zero is indicated also.

☐ Output: The floating point quotient of the two arguments is placed on the top of the stack. The stack pointer points to the same location as it did before the subroutine call.

The stack pointer, RPRES, and RPARG point to the MSBs of the floating point quotient. If an error occurred (N = 1 after return), the result is the number that best represents the correct result. For example, the largest number that can be represented if a division by zero was made.

☐ Special Cases: $0/0 = 0$ $0/X = 0$ $-X/0 =$ max. neg. number

$+X/0 =$ max. pos. number

☐ EXAMPLE: The floating point number (.DOUBLE format) contained in the ROM locations starting at address NUMBER is divided by the RAM locations pointed to by R5. The result is written to the RAM addresses pointed to by R5.

```
DOUBLE    .EQU    1
          MOV     R5,RPARG        ; Address of dividend
          MOV     #NUMBER,RPRES   ; Address of divisor
          CALL    #FLT_DIV        ; (NUMBER) / ((R5)) -> TOS
          JN      ERR_HND         ; Error occurred, check reason
          MOV     @RPRES+,0(R5)   ; Store FPP result (MSBs)
          MOV     @RPRES+,2(R5)
          MOV     @RPRES,4(R5)    ; LSBs
          ...                     ; Continue with program
```

Examples for the Basic Arithmetic Operations

The following example shows the following program steps for the .FLOAT format:

1) The registers used R5 to R12 are saved on the stack.

2) Four bytes are allocated on the stack to hold the results of the operations.

3) The address to a 12-digit BCD-buffer is loaded into pointer RPARG and the BCD-to-floating point conversion is called. The resulting floating point number is written to the result space previously allocated.

4) The resulting floating point number is multiplied with a number residing in the memory address VAL3. RPARG points to this address.

5) To the last result, a floating point number contained in the memory address VAL4 is added

6) The final result is converted back to BCD format (6 bytes) that can be displayed in the LCD.

7) The final result is copied to the RAM addresses BCDMSD, BCDMID and BCDLSB. The three necessary POP instructions correct the stack pointer to the value after the save register subroutine.

8) The registers used, R5 to R12, are restored from the stack. The system environment is exactly the same now as before the floating point calculations.

```
DOUBLE    .EQU    0               ; Use .FLOAT format
;
          ......                  ; Normal program
          CALL    #FLT_SAV        ; Save registers R5 to R12
```

```
        SUB     #4,SP               ; Allocate stack for result
        MOV     #BCDB,RPARG         ; Load address of BCD-buffer
        CALL    #CNV_BCD_FP         ; Convert BCD number to FP
;
; Calculate (BCD-number x VAL3) + VAL4
;
        MOV     #VAL3,RPARG         ; Load address of slope
        CALL    #FLT_MUL            ; Calculate next result
        MOV     #VAL4,RPARG         ; Load address of offset
        CALL    #FLT_ADD            ; Calculate next result
        CALL    #CNV_FP_BCD         ; Convert final FP result to BCD
        JN      CNVERR              ; Result too big for BCD buffer
        POP     BCDMSD              ; BCD number MSDs and sign
        POP     BCDMID              ; BCD digits MSD-4 to LSD+4
        POP     BCDLSD              ; BCD digits LSD+3 to LSD
                                    ; Stack is corrected by POPs
        CALL    #FLT_REC            ; Restore registers R5 to R12
                                    ; Continue with program
VAL3    .FLOAT  -1.2345             ; Slope
VAL4    .FLOAT  14.4567             ; Offset
CNVERR  ...                         ; Start error handler
```

The next example shows the following program steps for the .DOUBLE format:

1) The registers used, R5 to R15, are saved on the stack.

2) Six bytes are allocated on the stack to hold the results of the operations.

3) The ADC buffer address of the MSP430C32x (14-bit result) is written to RPARG and the last ADC result converted into a floating point number. The resulting floating point number is written to the result space previously allocated.

4) The resulting floating point number is multiplied with a number located at the memory address VAL3. RPARG points to this address.

5) To the last result, a floating point number contained in the memory address VAL4 is added.

6) The final result is converted back to binary format (6 bytes) and can be used for integer calculations.

7) The resulting binary number is copied to the RAM addresses BINMSD, BINMID, and BINLSB. The three necessary POP instructions correct the stack pointer to the value after the save register subroutine.

8) The registers used, R5 to R15, are restored from the stack. The system environment is now exactly the same as it was before the floating point calculations.

```
DOUBLE    .EQU    1                ; Use .DOUBLE format
;
          . . . . . .              ; Normal program
          CALL    #FLT_SAV         ; Save registers R5 to R15
          SUB     #6,SP            ; Allocate stack for result
          MOV     #ADAT,RPARG      ; Load address of ADC data buffer
          CALL    #CNV_BIN16U      ; Convert unsigned result to FP
;
; Calculate (ADC-Result x VAL3) + VAL4
;
          MOV     #VAL3,RPARG      ; Load address of slope
          CALL    #FLT_MUL         ; Calculate next result
          MOV     #VAL4,RPARG      ; Load address of offset
          CALL    #FLT_ADD         ; Calculate next result
          CALL    #CNV_FP_BIN      ; Convert final FP result to binary
          POP     BINMSD           ; Store MSBs of result and sign
          POP     BINMID           ; Store MIDs and LSBs
          POP     BINLSD           ; Stack is corrected by POPs
          CALL    #FLT_REC         ; restore registers R5 to R15
                                   ; Continue with program
VAL3      .DOUBLE 1.2E-3           ; Slope 0.0012
VAL4      .DOUBLE 1.44567E1        ; Offset 14.4567
```

5.6.3.5 Error Handling

Errors during the operation affect the status bits in the status register SR. If the N-bit contained in the status register is reset to zero, no error occurred. If the N-bit is set to one, an error occurred. The kind of error can be seen in Table 5–6. The columns .FLOAT and .DOUBLE show the returned results for each error.

Table 5–6. Error Indication Table

Error	Status	.FLOAT	.DOUBLE
No error	N=0	xxxx,xxxx	xxxx,xxxx,xxxx
Overflow positive	N=1, C=1, Z=1	FF7F,FFFF	FF7F,FFFF,FFFF
Overflow negative	N=1, C=1, Z=0	FFFF,FFFF	FFFF,FFFF,FFFF
Underflow	N=1, C=0, Z=0	0000,0000	0000,0000,0000
Divide by zero Dividend positive Dividend negative	N=1, C=0, Z=1	FF7F,FFFF or FFFF,FFFF	FF7F,FFFF,FFFF or FFFF,FFFF,FFFF

Software underflow is only treated as an error if the variable SW_UFLOW is set to one during assembly.

5.6.3.6 Stack Allocation

Before calling an operation 4 (resp. 6) bytes on the stack have to be reserved for the result. The following return address of the operation occupies another 2 bytes. The subroutines need one subroutine level during the calculations for the common initialization subroutine. The allocation in Figure 5–19 is shown for the use of FLT_SAV.
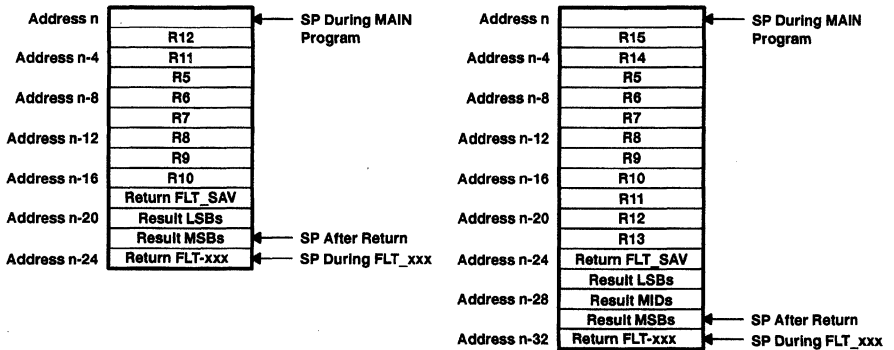
Figure 5–19. Stack Allocation for .FLOAT and .DOUBLE Formats

The FPP-subroutines correctly work only when the previous allocation is provided. This means the SP points to the return address on the stack. If the FPP-subroutines are called inside of a subroutine, a new result area must be allocated because the return address of the calling subroutine is now at the location the SP points to. The return address is overwritten in this case. The following example shows the correct procedure:

```
SUBR     SUB      #(ML/8)+1,SP        ; Allocate new result area
         MOV      @RPARG+,0(SP)       ; Fetch argument 2 to new
         MOV      @RPARG+,2(SP)       ; result area
         .if      DOUBLE=1
         MOV      @RPARG,4(SP)
         .endif
         MOV      SP,RPARG            ; Point again to argument 2
         MOV      #xx,RPRES           ; Point to argument 1
         CALL     #FLT_xxx            ; Use new result area for calc.
         ...                          ; Continue with calculations
         MOV      @SP+,result         ; Free allocated stack
         MOV      @SP+,result+2       ; Store result, correct SP
         .if      DOUBLE=1
         MOV      @SP+,result+4
         .endif
         RET
```

Note that it is strongly recommended that conscientious housekeeping be provided for SP to avoid stack overflow.

5.6.3.7 Number Range and Resolution

E = exponent of the floating point number. See Section 5.6.5, *Internal Data Representation* for more information.

.Float Format

Most positive number	FF7F,FFFF	$2^{127} \times (2 - 2^{-23})$	$= 3.402823 \times 10^{38}$
Least positive number	0000,0001	$2^{-128} \times (1 + 2^{-23})$	$= 2.938736 \times 10^{-39}$
Zero	0000,0000	0	$= 0.0$
Least negative number	0080,0000	-2^{-128}	$= -2.938736 \times 10^{-39}$
Most negative number	FFFF,FFFF	$-2^{127} \times (2 - 2^{-23})$	$= -3.402823 \times 10^{38}$
Resolution		$2^{-23} \times 2^{E}$	$= 119.2093 \times 10^{-9}\ 2^{E}$

.DOUBLE Format

Most positive number	FF7F,FFFF,FFFF	$2^{127} \times (2 - 2^{-39})$	$= 3.402824 \times 10^{38}$

Least positive number	0000,0000,0001	$2^{-128} \times (1 + 2^{-39})$	$= 2.938736 \times 10^{-39}$
Zero	0000,0000,0000	0	$= 0.0$
Least negative number	0080,0000,0000	-2^{-128}	$= -2.938736 \times 10^{-39}$
Most negative number	FFFF,FFFF,FFFF	$-2^{127} \times (2 - 2^{-39})$	$= -3.402824 \times 10^{38}$
Resolution		$2^{-39} \times 2^E$	$= 1.818989 \times 10^{-12} \times 2^E$

5.6.4   Calling Conventions for the Comparison

The comparison subroutine works much faster than a floating subtraction. Only the signs are compared in a first step to find out the relation of the two arguments. When the signs of the two operands are equal, the mantissas are compared. After the comparison, the status bits of the status register (SR) hold the result: The registers RPRES and RPARG point to the same location the SP points to (for the FPP version 3 they were not defined).

Table 5–7. Comparison Results

Relations	Status
Argument 1 > Argument 2	C=1, Z=0
Argument 1 < Argument 2	C=0, Z=0
Argument 1 = Argument 2	C=1, Z=1

The calling and use of the returned status bits is shown in the next example:

```
        . . .
        MOV      #ARG1,RPRES      ; Point to Argument 1 MSBs
        MOV      #ARG2,RPARG      ; Point to Argument 2 MSBs
        CALL     #FLT_CMP         ; Comparison: result to SR
        JEQ      EQUAL            ; Condition for program flow
        JHS      ARG1_GT_ARG2     ; ARG1 is greater than ARG2
        . . . . . .               ; ARG1 is less than ARG2
EQUAL   . . . . . .               ; ARG1 and ARG2 are equal
ARG1_GT_ARG2  . .                 ; ARG1 is greater than ARG2
;
; Other possibilities after the return
;
        CALL     #FLT_CMP         ; Comparison: result to SR
```

```
        JHS     ARG1_GE_ARG2     ; ARG1 is greater/equal ARG2
        ......                   ; ARG1 is less than ARG2
;

        CALL    #FLT_CMP         ; Comparison: result to SR
        JNE     ARG1_NE_ARG2     ; @RPRES not equal to @RPARG
        ......                   ; ARG1 is equal to ARG2
;

        CALL    #FLT_CMP         ; Comparison: result to SR
        JLO     ARG1_LT_ARG2     ; ARG1 is less than ARG2
        ......                   ; ARG1 is greater/equal ARG2
```

5.6.5 Internal Data Representation

The following description explains both the FLOAT and the DOUBLE formats. The two floating point formats consist of a floating point number whose:

❑ 8 most significant bits represent the exponent

❑ 24 (or 40 in the case of DOUBLE format) least significant bits hold the sign and the mantissa.
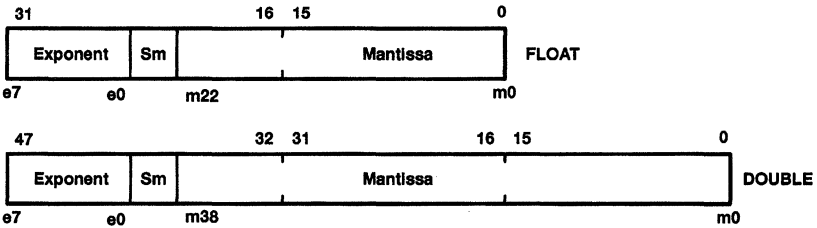
Figure 5–20. Floating Point Formats for the MSP430 FPP

Where:
Sm	Sign of floating point number (sign of mantissa)
mx	Mantissa bit x
ex	Exponent bit x
x	Valence of bit

The value N of a floating point number is

$$N = (-1)^{Sm} \times M \times 2^{E}$$

Note:

The only exception to the previous equation is the floating zero. It is represented by all zeroes (32 if FLOAT format or 48 if DOUBLE format). No negative zero exists, the corresponding number (0080,0000) is a valid non-zero number and is the smallest negative number.

A frequently asked question is why the MSP430 floating point format does not conform to the widely used IEEE format. There are two main reasons why this is not the case:

1) The MSP430 is often used in a real time environment where calculations need to be completed before the next input data are present.

2) Battery-supplied applications make calculations quickly to produce longer battery lifes (up to 10 years for example).

These two main reasons make a run-time optimized floating point package necessary. The format of the floating-point number plays an important role in reaching this target.

❏ With the MSP430-format, every floating-point number represents a valid value. No invalid combinations like Not a Number, Denormalized Number, or Infinity exist. This way the MSP430 FPP has a larger range than other FPPs. This allows a higher speed with the smallest memory usage. This eliminates the need for unnecessary checks for invalid numbers.

❏ The exponent of the IEEE-format is located in two bytes because of the location of the sign in the MSB of the floating point number. With the MSP430-format, the exponent resides completely within the high byte of the most significant word and can, therefore, use the advantages of the byte-oriented architecture of the MSP430. No shifts and no bit handling are necessary to manipulate the exponent.

5.6.5.1 Computation of the Mantissa M

$$M = 1 + \sum_{i=0}^{22} (m_i \times 2^{i-23}) \qquad . FLOAT \quad Format$$

$$M = 1 + \sum_{i=0}^{38} (m_i \times 2^{i-39}) \qquad . DOUBLE \quad Format$$

The result of the previous calculation is always:
$2 > M \geq 1$

Because the MSB of the normalized mantissa is always 1, a most significant non-sign bit is implied providing an additional bit of precision. This bit is hidden and called hidden bit. The sign bit is located at this place instead:

Sm = 0: positive Mantissa

Sm = 1: negative Mantissa

Note:

The mantissa of a negative floating point number is NOT represented as a 2's-complement number, only the sign bit (Sm) decides if the floating-point number is positive or negative.

5.6.5.2 Computation of the Exponent E

$$E = \sum_{i=0}^{7}(e_i \times 2^i) - 128$$

The MSB of the exponent indicates whether the exponent is positive or negative.

MSB of exponent = 0: The exponent is negative

MSB of exponent = 1: The exponent is positive

The reason for this convention is the representation of the number zero. This number is represented by all zeroes.

5.6.6 Execution Cycles

In the following evaluation the variables

```
X        .float   3.1416             ; Resp. .double 3.1416
Y        .float   3.1416*100         ; Resp. .double 3.1416*100
```

are the base for the calculations. The shown cycles include the addressing of one operand and the subroutine call itself:

```
MOV      #X,RPRES           ; Address 1st operand
MOV      #Y,RPARG           ; Address 2nd operand
CALL     #FLT_xxx           ; X <op> Y
....                        ; Result on TOS
```

Table 5–8 shows the number of cycles needed for the previously shown calculations:

Table 5–8. CPU Cycles needed for Calculations

Operation		.FLOAT	.DOUBLE	Comment
Addition	X + Y	184	207	
Subtraction	X – Y	177	199	
Multiplication	X × Y	395	692	Software Loop
Multiplication	X × Y	153	213	Hardware MPYer
Division	X / Y	405	756	
Comparison	X – Y	37	41	

5.6.7   Conversion Routines

5.6.7.1   General

To allow the conversion of integer numbers to floating point numbers and vice versa, the following subroutines are provided (both for .FLOAT and .DOUBLE format):

CNV_BINxxx   Convert 16-bit, 32-bit, or 40-bit signed and unsigned integer binary numbers to the floating point format. See Section 5.6.7.2, Binary to Floating Point Conversions.

CNV_BCD_FP   Convert a signed 12-digit BCD number to the floating point format

CNV_FP_BIN   Convert a floating point number to a signed 5 byte integer (40 bits)

CNV_FP_BCD   Convert a floating point number to a signed 12-digit BCD number

Common to these subroutines is:

1) The pointer RPARG points to the address of the input number

2) The input number is not modified, except when it is the result of the previous operation on the TOS

3) The result is located on the top of the stack (TOS), SP, RPARG, and RPRES point to the most significant word of the result

4) Only integers are converted. See Section 5.6.7.3, *Handling of Noninteger Numbers*, for the handling of non-integer numbers

5) The result is normally calculated using truncation, except when rounding is specified. The assembly-time variable SW_RND defines which mode is to be used.

SW_RND = 0:   Truncation is used, the trailing bits are cut off

SW_RND = 1:   Rounding is used, the first unused bit is added to the number

See Section 5.6.7.4, Rounding and Truncation, for details.

6) The subroutines can be used for 2-word (.FLOAT format) and 3-word (.DOUBLE format) floating point numbers. The assembly time variable DOUBLE defines which mode is to be used:

DOUBLE = 0:   Two word format .FLOAT

DOUBLE = 1:   Three word format .DOUBLE

7) All conversion subroutines need two (three) allocated words on the top of the stack. These words contain the result after the completed operation. A simple instruction is used for this allocation. It is the same allocation that is necessary anyway for the basic arithmetic operations. The possible instructions follow:

```
ML      .equ    24                  ; For .FLOAT. ML = 40 for .DOUBLE
FPL     .equ    (ML/8)+1            ; Length of one FP number
        SUB     #4,SP               ; .FLOAT format allocation
        SUB     #6,SP               ; .DOUBLE format allocation
or      SUB     #(ML/8)+1,SP        ; For both formats
or      SUB     #FPL,SP             ; For both formats
```

8) The FPP04.ASM package is needed. The completion routines of this file are used too

5.6.7.2 Conversions

The possible conversions are described in detail in the following sections. Input and output formats, error handling and number range are given for each conversion.

Binary to Floating Point Conversions

Binary numbers, 16-bit, 32-bit, and 40-bit long, are converted to floating point numbers. The subroutine call used defines if the binary number is treated as a signed or an unsigned number. No errors are possible, the N-bit of SR is always cleared on return. Six different conversion calls are provided:

CNV_BIN16 The 16-bit number, RPARG points to, is treated as a 16-bit signed number (see Figure 5–21.

Range: –32768 to + 32767   (08000h to 07FFFh)
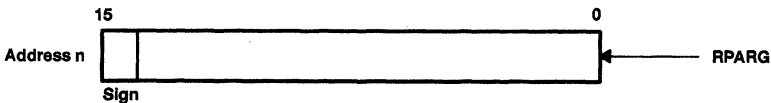
Figure 5–21. Signed Binary Input Buffer Format 16 Bits

CNV_BIN16U   The 16-bit number, RPARG points to, is treated as a 16-bit unsigned number (see Figure 5–22).

Range: 0 to + 65535   (00000h to 0FFFFh)
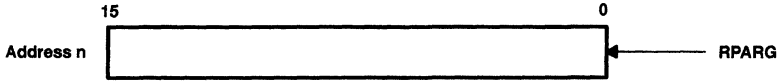
Figure 5–22. Unsigned Binary Input Buffer Format 16 Bits

CNV_BIN32 The 32-bit number, RPARG points to, is treated as a 32-bit signed number (see Figure 5–23).

Range: -2^{31} to $+2^{31} - 1$ (08000,0000h to 07FFF,FFFFh)
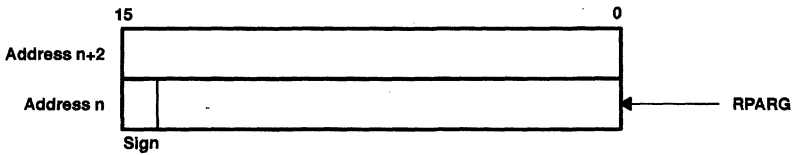
Figure 5–23. Signed Binary Input Buffer Format 32 Bits

CNV_BIN32U The 32-bit number, RPARG points to, is treated as a 32-bit unsigned number (see Figure 5–24).

Range: 0 to $2^{32} - 1$ (00000,0000h to 0FFFF,FFFFh)
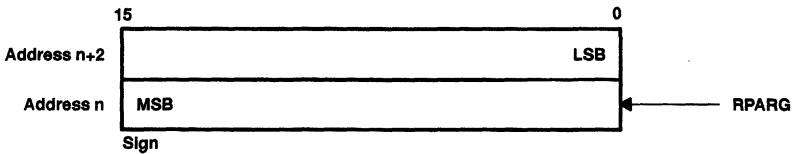
Figure 5–24. Unsigned Binary Input Buffer Format 32 Bits

CNV_BIN40 The 48-bit number, RPARG points to, is treated as a 40-bit signed (unsigned number) (see Figure 5–25).

Range signed: $-2^{40} + 1$ to $+2^{40} - 1$
(0FF00,0000,0001h to 000FF,FFFF,FFFFh)

Range unsigned: 0 to $+2^{40} - 1$
(00000,0000,0000h to 000FF,FFFF,FFFFh)

Address n+4		LSBs
Address n+2		
Address n	Sign Byte	MSBs

(15 ... 0) ◀——— RPARG
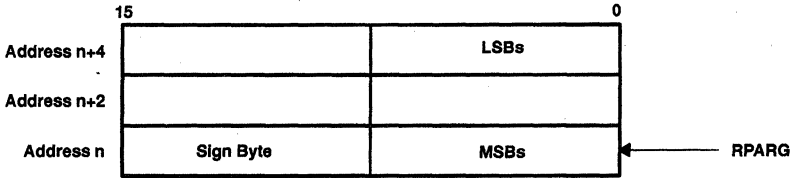
Figure 5–25. Binary Number Format 48 Bit

The previous conversion subroutines convert the 16-bit, 32-bit, or 48-bit numbers to a sign extended 48-bit number contained in the registers BIN_MSB, BIN_MID, and BIN_LSB. Depending on the call (signed or unsigned) used, the leading bits are sign extended or cleared. The resulting 48-bit number is converted afterwards. This allows an additional subroutine call:

CNV_BIN   The 48-bit signed number contained in the registers
                 BIN_MSB to BIN_LSB (3 words) is converted to a floating
                 point number (see Figure 5–26).

Range signed: -2^{40} +1 to $+2^{40} - 1$
(0FF00,0000,0001h to 000FF,FFFF,FFFFh)

Range unsigned: 0 to $+2^{40} - 1$
(00000,0000,0000h to 000FF,FFFF,FFFFh)

BIN_LSB		LSBs
BIN_MID		
BIN_MSB	Sign Byte	MSBs

(15 ... 0)
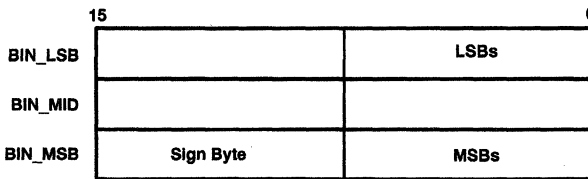
Figure 5–26. Binary Number Format 48 Bit

Note:

Input values outside of the 40-bit range, shown previously, do not generate error messages. The leading bits are truncated and only the trailing 40-bits are converted to the floating point format.

Errors: No error is possible, the N-bit of SR is always cleared on return.

Output: The output depends on the floating point format chosen. The format is selected with the assembly time variable DOUBLE.

.FLOAT   The two-word floating point result is written to the top of the stack. The SP, RPRES, and RPARG point to the MSBs of the floating point number.

.DOUBLE   The three-word floating point result is written to the top of the stack. The SP, RPRES, and RPARG point to the MSBs of the floating point number.

EXAMPLE: The 32-bit signed binary number contained in RAM locations BIN-LO and BINHI (MSBs) is converted to a three-word floating point number. The result is written to the RAM addresses RES, RES+2 and RES+4 (LSBs).

```
DOUBLE    .EQU    1                  ; Define .DOUBLE format
          MOV     #BINHI,RPARG       ; Address of binary MSBs
          CALL    #CNV_BIN32         ; Call conversion subroutine
          MOV     @RPRES+,RES        ; Store MSBs of result
          MOV     @RPRES+,RES+2      ;
          MOV     @RPRES,RES+4       ; Store LSBs of result
          . . .
```

Binary Coded Decimal to Floating Point Conversion

Binary coded decimal numbers (BCD numbers), 12 digits in length, are converted to floating point numbers. The MSB of the MSD word contains the sign of the BCD number:

MSB = 0: positive BCD number

MSB = 1: negative BCD number
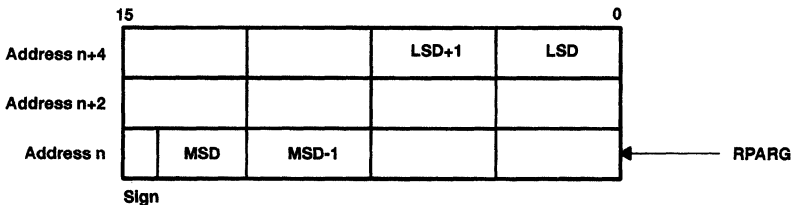
Figure 5–27. BCD Buffer Format

CNV_BCD_FP The 12-digit number (contained in 3 words, see Figure 5–27), RPARG points to, is converted to a floating point number.

Range: $-8 \times 10^{11} +1$ to $+8 \times 10^{11} -1$

Errors: No error is possible, the N-bit of the Status Register is always cleared on return. If non-BCD numbers are contained in the BCD-buffer, the result will be erroneous. If the MSD of the input number is greater than 7, then the input number is treated as a negative number.

Output: A floating point number on the top of the stack:

.FLOAT   The two-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

.DOUBLE   The three-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

EXAMPLE: The signed BCD number contained in the RAM locations starting at label BCDHI (MSDs) is to be converted to a two word floating point number. The result is to be written to the RAM addresses RES, and RES+2 (LSBs).

```
DOUBLE   .EQU    0                ; Define .FLOAT format
         MOV     #BCDHI,RPARG     ; Address of BCD MSDs
         CALL    #CNV_BCD_FP      ; Call conversion subroutine
         MOV     @RPRES+,RES      ; Store FP result (MSBs)
         MOV     @RPRES,RES+2     ; LSBs
         ...                      ; Continue with program
```

Floating Point to Binary Conversion

The floating point number pointed to by register RPARG is converted to a 40-bit signed binary number located on the top of the stack after conversion (see Figure 5–28).
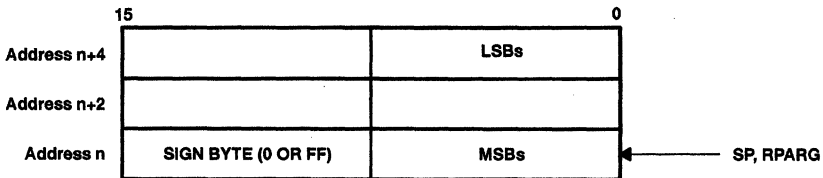
Figure 5–28. Binary Number Format

CNV_FP_BIN   The floating point number at the address in RPARG is converted to a 40-bit signed binary number.

Range signed: $-2^{40}+1$ to $+2^{40}-1$
                       (0FF00,0000,0001h to 000FF,FFFF,FFFFh)

Errors: If the absolute value of the floating point number is greater than $2^{40}-1$, then the N bit in the status register is set to one. Otherwise, the N bit is cleared.
            The result, put on top of the stack, is the largest signed binary number (saturation mode).

Output: A 40-bit signed, binary number at the top of the stack. The sign uses a full byte.

.FLOAT   SP, RPRES, and RPARG point to the MSBs of the three-word binary result. An additional word is inserted. It is the responsibility of the calling software to correct the stack by one level upwards after the result is read.

.DOUBLE   SP, RPRES, and RPARG point to the MSBs of the three-word binary result.

EXAMPLE: The floating point number (.DOUBLE format) contained in the RAM locations starting at label FPHI (MSBs) is converted to a 40-bit signed binary number. The result is written to the RAM addresses RES, RES+2, and RES+4 (LSBs).

```
DOUBLE    .EQU    1
          MOV     #FPHI,RPARG      ; Address of FP MSBs
          CALL    #CNV_FP_BIN      ; Call conversion subroutine
          JN      ERR_HND          ; |FP number| is too big
          MOV     @RPRES+,RES      ; Store binary result (MSBs)
          MOV     @RPRES+,RES+2    ;
          MOV     @RPRES,RES+4     ; LSBs
          ...                      ; Continue with program
```

Floating Point to Binary-Coded Decimal Conversion

The floating point number at the address in RPARG is converted to a signed 12-digit BCD number located on the top of the stack after conversion (see Figure 5–27). The MSD of the result has a maximum value of 7 because the sign bit uses the MSB position.

CNV_FP_BCD   The floating point number at the address in RPARG is converted to a 12-digit signed BCD number.

Range: $-8 \times 10^{11} +1$ to $+8 \times 10^{11} -1$

Errors: Three errors, at different stages of the conversion, are possible. These errors set the N-bit in the status register:

■ The exponent value of the floating point number is greater than 39, which represents an absolute value greater than 1.0995×10^{12}

■ The absolute value of the floating point number is greater than $8 \times 10^{11} -1$

■ The absolute value is greater than 1×10^{12}

Otherwise, the N bit is cleared.

The result, on the top of the stack, is the largest signed BCD number in case of an error.

Output: A 12-digit signed BCD number at the top of the stack (see Figure 5–27).

.FLOAT SP, RPRES, and RPARG point to the MSDs of the three-word BCD result. An additional word is inserted. It is the responsibility of the calling software to correct the stack by one level upwards after the reading of the result.

.DOUBLE SP, RPRES and RPARG point to the MSDs of the three-word BCD result.

EXAMPLE: The floating point number (.FLOAT format) contained in RAM locations starting at label FPHI (MSBs) is converted to a 12-digit BCD number. The result is written to RAM addresses RES, RES+2, and RES+4 (LSDs).

```
DOUBLE   .EQU    0
         MOV     #FPHI,RPARG      ; Address of FP MSBs
         CALL    #CNV_FP_BCD      ; Call conversion subroutine
         JN      ERR_HND          ; |FP number| is too big
         MOV     @SP+,RES         ; Store BCD result (MSDs)
         MOV     @SP,RES+2        ; SP is corrected
         MOV     2(SP),RES+4      ; LSDs
         ...                      ; Continue with program
ERR_HND  ...                      ; Correct error here
```

5.6.7.3 Handling of Non-Integer Numbers

The conversion subroutines handle only integer numbers when converting to or from floating point numbers. The reasons for this restriction are:

1) The stack grows if non-integer handling is included

2) The necessary program code of the conversion software grows larger

3) The integration of non-integer numbers is easier outside of the conversion subroutines

4) The execution time grows longer due to the necessary successive divisions or multiplies by 10. This cannot be tolerated in real time environments.

Binary to Floating-Point Conversion

If the location of the decimal point in the binary or hexadecimal number is known, the correction of the result is as follows:

The resulting floating point number is divided by the constant 2^n for binary numbers or 16^m for hexadecimal numbers (with m = 0.25 n). This is made simply by subtracting n from the exponent of the floating-point number. Overflow or underflow is not possible due to the restricted range of the binary input (-2^{40} +1 to $+2^{40} -1$) compared to the range of the floating-point numbers (-10^{32} to $+10^{32}$).

EXAMPLE: The binary 32-bit signed number contained in the RAM locations starting at label BINHI (MSBs) is converted to a floating-point number (.DOUBLE format). The virtual decimal point of the binary input number is 5 bits left to the LSB. This means the integer input number is 32-times too large. For example, the binary buffer contains 1011000 (88_{10}) but the real number is 10.11000 (2.75_{10}: 88 / 32 = 2.75)

```
MOV      #BINHI,RPARG     ; Address of binary buffer MSBs
CALL     #CNV_BIN32       ; Call conversion subroutine
SUB.B    #5,1(SP)         ; Correct result's exp. by 2^5
...                       ; Continue with corrected number
```

Binary-Coded Decimal (BCD) to Floating-Point Conversion

If the location of the decimal point in the BCD number is known, the correction of the result is as follows:

The resulting floating-point number is divided by the constant 10^n after the conversion. Overflow or underflow is not possible due to the restricted range of the

BCD input number $(-8 \times 10^{11} +1$ to $+8 \times 10^{11} -1)$ compared to the range of the floating-point numbers $(-10^{32}$ to $+10^{32})$.

EXAMPLE: The BCD number contained in the RAM locations starting at label BCDHI (MSDs) is converted to a floating-point number (.FLOAT format). The virtual decimal point of the BCD input number is 3 digits left to the LSD. This means the integer input number is 1000-times too large. For example, the BCD buffer contains 123456 and represents the number 123.456

```
DOUBLE   .EQU     0
         MOV      #BCDHI,RPARG     ; Address of BCD buffer MSDs
         CALL     #CNV_BCD_FP      ; Call conversion subroutine
         MOV      #FLT1000,RPARG   ; Address of constant 1000
         CALL     #FLT_DIV         ; Correct result by 1000
         ...                       ; Continue with corrected input
FLT1000  .FLOAT   1000             ; Correction constant 1000
```

If the location of the decimal point relative to the number's end is contained in a byte DPL (content > 0) the following code can be used.

```
DOUBLE   .EQU     1
         MOV      #BCDHI,RPARG     ; Address of BCD buffer MSDs
         CALL     #CNV_BCD_FP      ; Call conversion subroutine
LOOP     MOV      #DBL10,RPARG     ; Divide result by 10 as often -
         CALL     #FLT_DIV         ; as DPL defines
         DEC.B    DPL              ; DPL - 1
         JNZ      LOOP             ; Repeat as often as necessary
         ...                       ; Continue with corrected input
DBL10    .DOUBLE 10                ; Correction constant 10
```

Floating Point to Binary Conversion

If the binary result should contain n binary digits after the decimal point then the following procedure may be used.

The floating-point number is multiplied by the constant 2^n before the conversion call. This is made simply by adding of n to the exponent of the floating-point number. Overflow can occur if the floating-point number is very large. A very large floating-point number cannot be converted to binary format.

EXAMPLE: The floating-point number contained in the RAM locations starting at label FPHI (MSBs) is to be converted to a binary number (.FLOAT format). Four fractional bits of the resulting binary number should be included in the re-

sult. This means the result needs to be 16-times larger. For example, the floating-point number is 12.125 and the resulting binary number is 11000010_2 ($C2_{16}$) not only 1100_2 (C_{16}) .

```
DOUBLE   .EQU    0
         MOV     FPHI,0(SP)        ; MSBs of FP number to TOS
         MOV     FPHI+2,2(SP)      ; LSBs to TOS+2
         ADD.B   #4,1(SP)          ; Correct exponent by 2^4
         MOV     SP,RPARG          ; Act. pointer (if not yet done)
         CALL    #CNV_FP_BIN       ; Call conversion subroutine
         ...                       ; Result includes 4 add. bits
```

If the floating point number to be converted can be modified then a simplified code can be used.

```
         MOV     #FPHI,RPARG       ; Address of FP number MSBs
         ADD.B   #4,1(RPARG)       ; Correct exponent by 2^4
         CALL    #CNV_FP_BIN       ; Call conversion subroutine
         ...                       ; Result includes 4 add. bits
```

Floating Point to Binary Coded Decimal Conversion

If the BCD result of this conversion contains n digits after the decimal point, the following procedure can be used.

The floating-point number is multiplied by the constant 10^n before the conversion call. Overflow can occur if the floating-point number is very large. A very large floating-point number cannot be converted to BCD format due to the buffer length limit (12 digits maximum).

EXAMPLE: The floating-point number contained in the RAM locations starting at label FPHI (MSBs) is converted to a BCD number (.DOUBLE format). Two fractional digits should be included in the BCD result. This means the BCD result needs to be 100-times larger.

For example, the floating-point number is 12.125_{10}, the resulting BCD number written to the TOS is 1212_{10} (SW_RND = 0) respective 1213_{10} (SW_RND = 1) not only 12_{10}.

```
DOUBLE   .EQU    1
         MOV     #FPHI,RPARG       ; Address of FP number (MSBs)
         MOV     #DBL100,RPRES     ; Address of constant 100
         CALL    #FLT_MUL          ; FP number x 100 -> TOS
         CALL    #CNV_FP_BIN       ; Call conversion subroutine
```

```
        ...                         ; Result includes 2 add. digits
DBL100  .DOUBLE  100                ; Constant 100
```

5.6.7.4 Rounding and Truncation

Two different modes for conversions can be selected during the assembly of the conversion subroutines.

Truncation: Intermediate results of the conversion process are used as they are independent of the status of the next lower bits. This is the case if SW_RND = 0 is selected during assembly.

Rounding: Intermediate results of the conversion process are rounded depending on the status of the 1st bit not included in the current result (LSB–1). If this bit is set (1), the intermediate result is incremented. Otherwise, the result is not affected. If a carry occurs during the incrementing, the exponent is also corrected. Rounding is used if SW_RND = 1 is selected during assembly.

Rounding is applied (when SW_RND = 1) at the following conversion steps:

Binary to Floating Point: .FLOAT: the MSB of the truncated word is added to the 24-bit mantissa

.DOUBLE: all 40 input bits are included, no rounding is possible

BCD to Floating Point: like with the binary to floating point conversion

Floating Point to Binary: the 2^{-1} bit (the bit representing 0.5) of the floating point number is added to the binary integer result

Floating Point to BCD: The 2^{-1} bit (the bit representing 0.5) of the floating point number is added to the binary integer that is converted to a BCD number.

If rounding is specified during assembly (SW_RND = 1), the ROM code of the conversion subroutines is approximately 26 bytes larger than with truncation selected (SW_RND = 0).

5.6.7.5 Execution Cycles

To illustrate how long data conversion takes, the required cycles for each conversion are given for the converted values 1 and the largest possible value (8 x 10^{11} –1 for BCD conversions and 2^{40} –1 for binary conversions). The cycle count is given for the .FLOAT and for the .DOUBLE format and rounding is used.

The cycle count for each conversion includes the loading of the pointer RPARG, the subroutine call and the conversion itself.

Table 5–9. Execution Cycles of the Conversion Routines

Conversion	.FLOAT 1	.FLOAT max	.DOUBLE 1	.DOUBLE max
CNV_BIN40	418	67	422	71
CNV_BCD_FP	1223	890	1227	894
CNV_FP_BIN	535	67	531	63
CNV_FP_BCD	1174	706	1170	701

5.6.8 Memory Requirements of the Floating Point Package

The memory requirements of an implemented floating-point package depend on the routines used and the precision applied. The following values refer to a completely implemented package. Truncation is used with the conversion routines. The given numbers indicate bytes.

Table 5–10. Memory Requirements without Hardware Multiplier

Package	.FLOAT	.DOUBLE
Basic Arithmetic Operations	604	696
Conversion Subroutines	342	338
Complete FPP	946	1034

Table 5–11. Memory Requirements with Hardware Multiplier

Package	.FLOAT	.DOUBLE
Basic Arithmetic Operations	638	786
Conversion Subroutines	342	338
Complete FPP	980	1124

5.6.9 Inclusion of the Floating-Point Package into the Customer Software

This section shows how to insert the floating-point package into the user's software. The symbolic definition of the working registers makes it necessary to include the FPP-definition file (FPPDEF4.ASM) before the customer's software. Otherwise, the assembler allocates an address word for every use of one of the working registers during the first pass of the assembler. During the second assembler pass, this proofs to be wrong and the assembler run fails. The two files FPP04.ASM and CNV04.ASM need to be located together as shown in the following examples. This is due to the common parts that are connected with jumps.

The constant DOUBLE decides which FPP version is generated. It is assumed that the FPP files are located in a directory named `c:\fpp`. If this is not the case, then the name of this directory is to be used.

```
;
          .text    08000h              ; ROM/EPROM start address
STACK     .equ     0600h               ; Initial value for SP
;
DOUBLE    .equ     1                   ; Use .DOUBLE format FPP
SW_UFLOW  .equ     0                   ; Underflow is no error
SW_RND    .equ     1                   ; Use rounding for conversions
HW_MPY    .equ     1                   ; Use the hardware multiplier
;
          .copy    c:\fpp\fppdef4.asm      ; FPP Definitions
          .copy    c:\fpp\fpp04.asm        ; FPP file
          .copy    c:\fpp\cnv04.asm        ; FPP Conversions
;
; Customer software starts here
;
START     MOV      #STACK,SP                ; Allocate stack
          .....                             ; User's SW starts here
; Power-up start address:
;
          .sect    "RstVect",0FFFEh
          .word    START                    ; Reset vector
```

A second possibility is shown in the following. The FPP is located after the user's software:

```
;
          .text    0E000h              ; ROM start address
STACK     .equ     0300h               ; Initial value for SP
;
DOUBLE    .equ     0                   ; Insert .FLOAT format FPP
SW_UFLOW  .equ     1                   ; Underflow is an error
SW_RND    .equ     0                   ; No rounding for conversions
HW_MPY    .equ     0                   ; No hardware multiplier
;
          .copy    c:\fpp\fppdef4.asm      ; FPP Definitions
;
; Customer software starts here
```

```
;
START    MOV      #STACK,SP                   ; Allocate stack
         .....                                ;
         .....                                ; End of user's software
         .copy    c:\fpp\fpp04.asm            ; Copy FPP file
         .copy    c:\fpp\cnv04.asm            ; Copy conversions
;
; Power-up start address:
;
         .sect    "RstVect",0FFFEh
         .word    START                       ; Reset vector
```

5.6.10 Software Examples

The following subroutines for mathematical functions use the same conventions like the basic arithmetic functions described previously.

❑ RPARG points to the operand X for single operand functions (lnX, e^X)

❑ RPRES points to the first operand (base) and RPARG to the second one if two operands are used (e.g. for the power function a^b)

❑ The result of the operation is placed on the top of the stack, RPARG, RPRES and SP point to the result.

5.6.10.1 Square Root Subroutine

The following subroutine shows the use of the floating-point package for the calculation of the square root of a number X. The NEWTONIAN approach is used:

$$x_{n+1} = 0.5 \times \left(x_n + \frac{X}{x_n} \right)$$

The subroutine uses the RPARG register as a pointer to the number X and places the result on the top of the stack.

The algorithm used for the first estimation – exponent/2 and different correction for even and odd exponents – leads to the worst case estimation errors of +8% and –13%. This relatively exact estimations lead to only four iteration loops to get the full accuracy.

The number range of X for the square-root function contains all positive numbers including zero. Negative values for X return the previous result on the top of the stack and the N bit set as an error indication.

The calculation errors for the square-root function are shown in the following table. They indicate relative errors.

Table 5–12. Relative Errors of the Square Root Function

X	.FLOAT	.DOUBLE	Comment
$+3.0 \times 10^{-39}$	6.8×10^{-8}		Smallest FPP number
0.0	0	0	Zero
1.0	0	0	
6.0	$+5.4 \times 10^{-9}$	$+1.3 \times 10^{-12}$	
8.0	$+6.7 \times 10^{-8}$	$+1.3 \times 10^{-12}$	
$+3.4 \times 10^{38}$	$+4.6 \times 10^{-9}$	$+2.2 \times 10^{-11}$	Largest FPP number

Calculation times:

.FLOAT with hardware multiplier: 2300 cycles   4 iterations

.FLOAT without hardware multiplier: 2300 cycles   (no multiplication used)

.DOUBLE with hardware multiplier: 4000 cycles   4 iterations

.DOUBLE without hardware multiplier: 4000 cycles

```
; Square Root Subroutine X^0.5      Result on TOS = (@RPARG)^0.5
;
; Call:  MOV      #addressX,RPARG  ; RPARG points to address of X
;        CALL     #FLT_SQRT        ; Call the square root function
;        ...                       ; RPARG, RPRES and SP point to
;                                  ; result X^0.5. N-bit for error
;
; Range: 0 =< X < 3.4x10^+38
;
; Errors:      X < 0:   N = 1    Result: previous result
;
; Stack: FPL + 2 bytes
;
; Calculates the square root of the number X, RPARG points to.
; SP, RPARG and RPRES point to the result on TOS
;
FLT_SQRT .equ $
         TST.B    0(RPARG)         ; Argument negative?
         JN       SQRT_ERR         ; Yes, return with N = 1
```

```
        MOV     @RPARG+,2(SP)    ; Copy X to result area
        MOV     @RPARG+,4(SP)
        .if     DOUBLE=1
        MOV     @RPARG+,6(SP)
        .endif
        CLR     HELP
        .if     DOUBLE=1
        TST     6(SP)            ; Check for X = 0
        JNE     SQ0
        .endif
        TST     4(SP)
        JNE     SQ0
        TST     2(SP)
        JEQ     SQ3              ; X = 0: result 0, no error
;
SQ0     PUSH    #4               ; Loop count (4 iterations)
        PUSH    FPL+4(SP)        ; Push X on stack for Xn
        PUSH·   FPL+4(SP)
        .if DOUBLE=1
        PUSH    FPL+4(SP)
        .endif
;
; 1st estimation for X^0.5: exponent even: 0.5 x fraction + 0.5
;                                          exponent odd:  fraction .or. 0.30h
;                                          exponent/2
;
        RRA.B   1(SP)            ; Exponent/2
        JC      SQ1              ; Exponent even or odd?
        RRA.B   @SP              ; Exponent is even:
        JMP     SQ2              ; 0.5 + 0.5 x fraction
SQ1     BIS.B   030h,0(SP)       ; Exponent is odd: correction
SQ2     XOR.B   #040h,1(SP)      ; Correct exponent
;
SQLOOP  MOV     SP,RPARG         ; Pointer to Xn
        MOV     SP,RPRES
        ADD     #FPL+4,RPRES     ; Pointer to X
```

```
          SUB       #FPL,SP            ; Allocate stack for result
          CALL      #FLT_DIV           ; X/xn
          ADD       #FPL,RPARG         ; Point to xn
          CALL      #FLT_ADD           ; X/xn + xn
          DEC.B     1(RPRES)           ; 0.5 x (X/xn + xn) = xn+1
          MOV       @SP+,FPL-2(SP)     ; xn+1 -> xn
          MOV       @SP+,FPL-2(SP)
          .if       DOUBLE=1
          MOV       @SP+,FPL-2(SP)
          .endif
          DEC       FPL(SP)            ; Decrement loop counter
          JNZ       SQLOOP
          MOV       @SP+,FPL+2(SP)     ; N = 0 (FLT_ADD)
          MOV       @SP+,FPL+2(SP)     ; Root to result space
          .if       DOUBLE=1
          MOV       @SP+,FPL+2(SP)
          .endif
          ADD       #2,SP              ; Skip loop count
SQ3       BR        #FLT_END           ; To completion part
SQRT_ERR  MOV       #FN,HELP           ; Root of negative number: N = 1
          JMP       SQ3                ;
```

5.6.10.2 Cubic-Root Subroutine

The cubic root of a number is calculated the same as the square root, using the Newtonian approach. The formula for the cubic root of X is:

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{X}{x_n^2}\right)$$

The subroutine uses the RPARG register as a pointer to the number X and places the result on the top of the stack.

The algorithm used for the first estimation – exponent/3 and a constant fraction value ±1.4 – leads to worst case estimation errors of +40% and –37%. This estimation leads to four (.FLOAT) or five (.DOUBLE) iteration loops to get the full accuracy.

The number range of X for the cubic-root function contains all numbers including zero. No error is possible.

The calculation errors for the cubic-root function are shown in the following table. They indicate relative errors.

Table 5–13.   Relative Errors of the Cubic Root Function

X	.FLOAT	.DOUBLE	Comment
-3.4028×10^{38}	1.2×10^{-8}	$+2.2\times10^{-13}$	Most negative number
-1.0	0	0	-1.0
-2.9387×10^{-39}	1.7×10^{-7}	-3.8×10^{-13}	Least negative number
0.0	0	0	Zero
$+2.9387\times10^{-39}$	-1.7×10^{-7}	$+3.8\times10^{-13}$	Least positive number
$+1.0$	0	0	$+1.0$
$+3.4028\times10^{38}$	-1.2×10^{-8}	-2.2×10^{-13}	Most positive number

Calculation times:

.FLOAT with hardware multiplier: 5000 cycles   4 iterations

.FLOAT without hardware multiplier: 6100 cycles

.DOUBLE with hardware multiplier: 10200 cycles   5 iterations

.DOUBLE without hardware multiplier: 12600 cycles

```
; Cubic Root Subroutine X^1/3  Result on TOS = (@RPARG)^1/3
;
; Call:  MOV     #addressX,RPARG ; RPARG points to X
;        CALL    #FLT_CBRT        ; Call the cubic root function
;        ...                      ; RPARG, RPRES, SP point to result
;                                 ; Result on the top of the stack
;
; Formula:      xn+1 = 1/3(2xn + X x xn^-2)
;
; Range:        -3.4x10^+38 =< X =< 3.4x10^+38
;
; Errors:                       No errors possible
;
; Stack:        2 x FPL + 2 bytes
;
; Calculates the cubic root of the number X, RPARG points to.
; SP, RPARG and RPRES point to the result on TOS
;
```

```
FLT_CBRT MOV     @RPARG+,2(SP)      ; Copy X to result area
         MOV     @RPARG+,4(SP)
         .if     DOUBLE=1
         MOV     @RPARG+,6(SP)
         .endif
         .if DOUBLE=1
         TST     6(SP)              ; Check for X = 0
         JNE     CB0
         .endif
         TST     4(SP)
         JNE     CB0
         TST     2(SP)
         JEQ     CB3                ; X = 0: result 0
;
CB0      .equ    $
         .if     DOUBLE=0           ; Loop count
         PUSH    #4                 ; .FLOAT  4 iterations
         .else
         PUSH    #5                 ; .DOUBLE 5 iterations
         .endif
         PUSH    FPL+4(SP)          ; Push X on stack for Xn
         PUSH    FPL+4(SP)
         .if DOUBLE=1
         PUSH    FPL+4(SP)
         .endif
;
; 1st estimation for X^1/3:         exponent/3, fraction = +-1.4
;
         MOV.B   1(SP),RPARG        ; Exponent of X 00xx
         AND     #080h,0(SP)        ; Only sign of X remains
         ADD     #08034h,0(SP)      ; +-1.4 for 1st estimation
         TST.B   RPARG              ; Exponent's sign?
         JN      DCL$2              ; positive
DCL$1    DEC.B   1(SP)              ; Neg. exp.: exponent - 1
         ADD.B   #3,RPARG           ; Add 3 until 080h is reached
         JN      CBLOOP             ; 080h is reached,
```

```
            JMP     DCL$1           ; Continue
DCL$3       INC.B   1(SP)           ; Pos. exp.: exponent + 1
DCL$2       SUB.B   #3,RPARG        ; Subtr. 3 until 080h is reached
            JN      DCL$3           ; Continue
;
CBLOOP      MOV     SP,RPARG        ; Point to xn
            MOV     SP,RPRES
            SUB     #FPL,SP         ; Allocate stack for result
            CALL    #FLT_MUL        ; xn^2
            ADD     #2*FPL+4,RPRES  ; Point to A
            CALL    #FLT_DIV        ; X/xn^2
            INC.B   FPL+1(SP)       ; xn x 2
            ADD     #FPL,RPARG      ; Point to 2xn
            CALL    #FLT_ADD        ; X/xn^2 + 2xn
            MOV     #FLT3,RPARG     ; 1/3 x (X/xn^2 + 2xn) = xn+1
            CALL    #FLT_DIV
            MOV     @SP+,FPL-2(SP)  ; xn+1 -> xn
            MOV     @SP+,FPL-2(SP)
            .if     DOUBLE=1
            MOV     @SP+,FPL-2(SP)
            .endif
            DEC     FPL(SP)         ; Decr. loop count
            JNZ     CBLOOP
            MOV     @SP+,FPL+2(SP)  ; Result to result area
            MOV     @SP+,FPL+2(SP)  ; Cubic root to result space
            .if     DOUBLE=1
            MOV     @SP+,FPL+2(SP)
            .endif
            ADD     #2,SP           ; Skip loop count
CB3         CLR     HELP            ; No error
            BR      #FLT_END        ; Normal termination
;
            .if     DOUBLE=1
FLT3        .DOUBLE 3.0             ; Constant for cubic root
            .else
FLT3        .FLOAT  3.0
```

```
        .endif
```

5.6.10.3 Fourth-Root Subroutine

The fourth root of a number is calculated by calling the square root subroutine twice.

EXAMPLE: the fourth root is calculated for a number residing in RAM at address NUMBER (MSBs). The fourth root is written to RESULT. The previous result on TOS must not be overwritten.

```
        SUB     #ML/8+1,SP      ; Allocate work area
        MOV     #NUMBER,RPARG   ; Address of NUMBER to RPARG
        CALL    #FLT_SQRT       ; Square root of NUMBER on TOS
        JN      ERROR           ; NUMBER is negative
        CALL    #FLT_SQRT       ; Fourth root on TOS
        MOV     @SP+,RESULT      ; 4th root MSBs
        MOV     @SP+,RESULT+2    ; Correct SP to previous result
        .if     DOUBLE=1
        MOV     @SP+,RESULT+4    ; LSBs for DOUBLE
        .endif
```

5.6.10.4 Other Root Subroutines

Using the same calculations shown previously, higher roots can also be calculated using the Newtonian approach. The generic formula for the mth root out of A is:

$$x_{n+1} = \frac{1}{m}\left((m-1)x_n + \frac{A}{x_n^{m-1}}\right)$$

To get short calculation times – which means only few iterations are necessary – the choice of the first estimation x_0 is very important. For the above formula a good first iteration x_0 is (M = mantissa, E = exponent):

$$x_0 = \left(\frac{(M-1)}{m}+1\right) \times 2^{E/m}$$

5.6.10.5 Calculations With Intermediate Results

If a calculation cannot be executed simply and has intermediate results, a new result space is used. This is done by subtracting 4 (.FLOAT) or 6 (.DOUBLE) from the stack pointer.

EXAMPLE: The following function for e is to be calculated. The example is valid for both formats:

$$e = a \times b - \frac{c}{d}$$

```
FPL       .equ     (ML/8)+1            ; Length of a FPP number
;
          SUB      #FPL,SP            ; Allocate result space 0 (RS0)
          MOV      #a,RPRES          ; Address argument 1
          MOV      #b,RPARG          ; Address argument 2
          CALL     #FLT_MUL          ; a x b -> RS0
          SUB      #FPL,SP            ; Allocate result space 1 (RS1)
          MOV      #c,RPRES          ; Address c
          MOV      #d,RPARG          ; Address d
          CALL     #FLT_DIV          ; c/d -> RS1
          ADD      #FPL,RPRES        ; Address (a x b) in RS0
          CALL     #FLT_SUB          ; e = (a x b) - c/d -> RS1
          MOV      @SP+,FPL-2(SP)    ; Result e to RS0
          MOV      @SP+,FPL-2(SP)    ; Overwrite (a x b) with e
          .if      DOUBLE=1
          MOV      @SP+,FPL-2(SP)    ; LSBs for DOUBLE
          .endif
;
; Housekeeping is made, SP points to RS0 again, but not
; RPARG and RPRES
```

EXAMPLE: The multiply-and-add (MAC) function for e shown in the following is calculated. The example is written for both formats:

$$e_{n+1} = a \times b + e_n$$

```
          SUB      #ML/8+1,SP        ; Allocate result space
          MOV      #a,RPRES          ; Address argument 1
          MOV      #b,RPARG          ; Address argument 2
          CALL     #FLT_MUL          ; a x b
          MOV      #e,RPARG          ; Address e
          CALL     #FLT_ADD          ; (a x b)+ e
```

```
          MOV       @RPARG+,e          ; Actualize e with result
          MOV       @RPARG+,e+2        ; MIDs or LSBs
          .if       DOUBLE=1
          MOV       @RPARG+,e+4        ; LSBs
          .endif
;
; SP and RPRES still point to the result, RPARG may be used
; for the next argument address.
```

5.6.10.6 Absolute Value of a Number

If the absolute value of a number is needed, this is done by simply resetting the sign bit of the number.

EXAMPLE: the absolute value of the result on the top of the stack is needed.

```
BIC       #080h,0(SP)        ; |result| on TOS
```

5.6.10.7 Change of the Sign of a Number

If a sign change is necessary (multiplication by –1), this is done by simply inverting the sign bit of the number.

EXAMPLE: the sign of the result on the top of the stack is changed.

```
XOR       #080h,0(SP)        ; Negate result on TOS
```

5.6.10.8 Integer Value of a Number

The integer value of a floating-point number can be calculated with the subroutine FLT_INTG in the following example. The pointer RPARG is loaded with the address of the number. The result is then placed on the top of the stack. No error is possible. Numbers below one are returned as zero. The subroutine can handle .FLOAT and .DOUBLE formats.

```
;
; Calculate the integer value of the number RPARG points to.
; Result: on top of the stack. RPARG, RPRES and SP point to it
; Call    MOV       #number,RPARG     ; Address to RPARG
;         CALL      #FLT_INTG         ; Call subroutine
;         ...                         ; Result on TOS
;
FLT_INTG MOV.B    1(RPARG),COUNTER    ; Exponent to COUNTER
         MOV      @RPARG+,2(SP)       ; MSBs and Exponent
```

```
        MOV     @RPARG+,4(SP)    ; LSBs .FLOAT
        .if     DOUBLE=1
        MOV     @RPARG,6(SP)     ; LSBs .DOUBLE
        .endif
        MOV     #0FFFFh,ARG2_MSB ; Mask for fractional part
        .if     DOUBLE=1
        MOV     #0FFFFh,ARG2_MID
        .endif
        MOV     #0FFFFh,ARG2_LSB
        JMP     L$30
;
INTGLP  CLRC                     ; Shift 0 in always
        RRC.B   ARG2_MSB         ; Shift mask to next lower bit
        .if     DOUBLE=1
        RRC     ARG2_MID
        .endif
        RRC     ARG2_LSB
        DEC     COUNTER          ; Shift as often as:
L$30    CMP     #080h,COUNTER    ; SHIFT COUNT = EXPONENT - 07Fh
        JHS     INTGLP
        BIC     ARG2_MSB,2(SP)   ; Mask out fract. part
        .if     DOUBLE=1
        BIC     ARG2_MID,4(SP)   ; For .DOUBLE format
        BIC     ARG2_LSB,6(SP)
        .else
        BIC     ARG2_LSB,4(SP)   ; For .FLOAT format
        .endif
        MOV     SP,RPARG         ; Both pointer to result's MSBs
        ADD     #2,RPARG
        MOV     RPARG,RPRES
        RET                      ; Return with Integer on TOS
```

EXAMPLE: the integer value of the floating point number residing at address VOL1 is placed on TOS.

```
        MOV     #VOL1,RPARG      ; Load pointer with address
        CALL    #FLT_INTG        ; Calculate integer of VOL1
```

```
    ....                    .            ; Integer on TOS
```

5.6.10.9 Fractional Part of a Number

The fractional part of a floating-point number can be calculated with the sub-routine FLT_FRCT in the following example. The pointer RPARG is loaded with the address of the number. The result is placed on the top of the stack. No error is possible. The subroutine can handle both floating-point formats. The subroutine calls the subroutine FLT_INTG shown previously.

Integer values or very large numbers return a zero value due to the given resolution.

.DOUBLE format: numbers $> 1.099512 \times 10^{12}$ ($>2^{40}$)

.FLOAT format: numbers $> 1.6777216 \times 10^{7}$)   ($>2^{24}$)

```
; Calculate the fractional part of the number RPARG points to.
; Result: on top of the stack. RPARG, RPRES and SP point to it
; Subroutine FLT_INTG is used
; Call     MOV      #number,RPARG       ; Address to RPARG
;          CALL     #FLT_FRCT           ; Call subroutine
;          ...                          ; Result on TOS
;
FLT_FRCT PUSH       RPARG               ; Copy operand's address
         .if        DOUBLE=1
         PUSH       4(RPARG)            ; Copy operand to allow the use
         .endif                         ; of the value on TOS
         PUSH       2(RPARG)
         PUSH       @RPARG
         CALL       #FLT_INTG           ; Integer part of operand to TOS
         MOV        ML/8+1(SP),RPRES    ; Operand address to RPRES
         CALL       #FLT_SUB            ; Operand - Integer part to TOS
         .if        DOUBLE=1            ; Housekeeping:
         MOV        @SP+,ML/8+3(SP)     ; Fractional part back
         .endif
         MOV        @SP+,ML/8+3(SP)     ; To result area
         MOV        @SP+,ML/8+3(SP)
         ADD        #2,SP               ; Skip saved operand address
         CLR        HELP                ; No error
         BR         #FLT_END            ; Use FPP termination
```

;

EXAMPLE: the fractional part of the floating-point number R5 points to is placed on TOS.

```
MOV      R5,RPARG         ; Load pointer with address
CALL     #FLT_FRCT        ; Calculate fractional part
....                      ; Fractional part on TOS
```

5.6.10.10 Approximation of Integrals

Simpson's Rule states that the area A limited by the function f(x), the x-axis, x_0 and x_N is approximately:

$$A = \int_{x_0}^{x_N} f(x) \approx \frac{1}{3} \times h \times \left[(y_0 + y_N) + 2(y_2 + y_4 \dots y_{N-2}) + 4(y_1 + y_3 \dots y_{N-1}) \right]$$
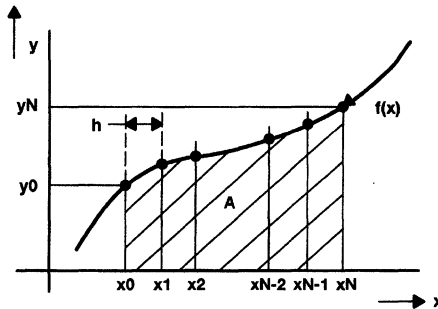
Figure 5–29. Function f(x)

The subroutine SIMPSON, in the following code, processes N+1 inputs pointed to by register RPARG and computes the area A after the measurement of sample N. The result is written back to the RAM location A.

This integration method can be used for the calculation of the apparent power with electronic electricity meters. The absolute values of current and voltage are added up and are multiplied afterwards.

```
;
; Subroutine for the approximation of integrals. Samples
; y0 to yN are processed and stored in location A.
; Nmax = 254 (if larger, a word has to be used for INDEXn)
```

```
;
; Call:   CLR.B    INDEXn          ; Before 1st call: n = 0
; LOOP    MOV      #sample,RPARG    ; Address of yn
;         CALL     #SIMPSON         ; Process sample yn
;         CMP.B    #N+1,INDEXn      ; YN processed?
;         JLO      LOOP             ; No, proceed
;         ...                       ; Yes, integral in A
;
N        .equ     8                ; Max. index (must be even)
         .if      DOUBLE=0
A        .equ     0200h            ; summed up value (integral)
INDXn    .equ     0204H            ; Index n (0 to N)
FLT3     .float   3.0
h        .float   0.32             ; Difference h: yn+1 - yn
         .else
A        .equ     0200h            ; Summed up value (integral)
INDXn    .equ     0206H            ; Index n (0 to N)
FLT3     .double  3.0
h        .double  0.32             ; Difference h: yn+1 - yn
         .endif
SIMPSON  SUB      #(ML/8)+1,SP     ; Allocate new workspace
         MOV      @RPARG+,0(SP)    ; Fetch yn
         MOV      @RPARG+,2(SP)
         .if DOUBLE=1
         MOV      @RPARG,4(SP)
         .endif
         CMP.B    #0,INDXn         ; 1st value y0?
         JEQ      Y0
         CMP.B    #N,INDXn         ; Last value yN?
         JEQ      YN
         BIT      #1,INDXn         ; Odd or even n?
         JZ       YEVEN
         INC.B    1(SP)            ; Odd: value x 4
YEVEN    INC.B    1(SP)            ; Even: value x 2
         MOV      #A,RPARG         ; Fetch summed-up value A
         MOV      SP,RPRES         ; New sample yn on TOS
```

```
          CALL    #FLT_ADD          ; Add it to A
          JMP     Y0                ; Store added result in A
;
YN        MOV     #A,RPARG          ; Last value yN: calculate
          MOV     SP,RPRES          ; New sample yn on TOS
          CALL    #FLT_ADD          ; Add last result to A
          MOV     #FLT3,RPARG       ; To constant 3.0
          CALL    #FLT_DIV          ; Divide summed-up value by 3.0
          MOV     #h,RPARG          ; Multiply with distance h
          CALL    #FLT_MUL
;
Y0        MOV     @SP+,A            ; Store result to A
          MOV     @SP+,A+2          ; and correct stack
          .if     DOUBLE=1
          MOV     @SP+,A+4
          .endif
          INC.B   INDXn             ; Next n
          RET                       ; Return with integral in A
```

EXAMPLE: The function f(x) described by the calculated results on top of the stack is integrated using Simpson's rule..

```
          CLR.B   INDXn             ; Initialization: INDXn = 0
INTLOP    ...                       ; Calculation, result on TOS
          CALL    #SIMPSON          ; Process samples y0 to yN
          CMP.B   #N+1,INDXn        ; Last sample yN processed?
          JLO     INTLOP            ; No, continue
          ...                       ; Yes, result in A
```

5.6.10.11 Statistical Calculations

The mean value, the standard deviation, and the variance of measured samples can be calculated with the following subroutines.

❏ STAT_INIT clears the RAM locations used for data gathering.

❏ STAT_PREP adds the input sample to the RAM location SUMYi, the squared input sample to SUM2Yi and increments the sample counter N.

☐ STAT_CALC calculates mean, standard deviation, and variance from these three values and writes them back to the RAM locations used for data recording.

$$MeanValue = \frac{\sum_{i=1}^{i=N} y_i}{N}$$

$$Variance = \frac{\sum_{i=1}^{i=N} y_i^2 - \frac{\left(\sum_{i=1}^{i=N} y_i\right)^2}{N}}{N} = \frac{\sum_{i=1}^{i=N} y_i^2 - MeanValue \times \sum_{i=1}^{i=N} y_i}{N}$$

$$StandardDeviation = \sqrt{\frac{\sum_{i=1}^{i=N} y_i^2 - \frac{\left(\sum_{i=1}^{i=N} y_i\right)^2}{N}}{N-1}} = \sqrt{Variance \times \frac{N}{N-1}}$$

```
;
; RAM locations for the input samples:
;
N          .equ    0200h               ; Number of input samples (binary)
SUMYi      .equ    N+(ML/8)+1          ; Summed-up samples yi
SUM2Yi     .equ    SUMYi+(ML/8)+1      ; Sum of squared samples yi
;
; The same RAM-locations are used for the three results:
;
MEANV      .equ    N                   ; Mean Value after return
STDDEV     .equ    SUMYi               ; Standard Deviation after return
VARIANCE   .equ    SUM2Yi              ; Variance after return
;
           .if     DOUBLE=1
FLT1       .DOUBLE 1.0                 ; Floating 1.0
           .else
FLT1       .FLOAT  1.0
           .endif
;
```

```
; STAT_INIT initializes the RAM-locations for statistics
;
STAT_INIT CLR    N                     ; Clear sample counter
          CLR    SUM2Yi                ; Clear sum of squared samples
          CLR    SUM2Yi+2
          .if    DOUBLE=1
          CLR    SUM2Yi+4
          .endif
          CLR    SUMYi                 ; Clear sum of input samples
          CLR    SUMYi+2
          .if    DOUBLE=1
          CLR    SUMYi+4
          .endif
          RET
; STAT_PREP sums-up the sample pointed to by RPARG in SUMYi
; (summed-up yi) and in SUM2Yi (summed-up squared yi).
; The binary sample counter N is incremented
;
STAT_PREP PUSH   RPARG                 ; Save address of input sample
          SUB    #(ML/8)+1,SP          ; Allocate stack space
          MOV    RPARG,RPRES           ; Copy input sample address
          CALL   #FLT_MUL              ; (yi)^2
          MOV    #SUM2Yi,RPRES         ; Add (yi)^2 to SUM2Yi
          CALL   #FLT_ADD              ; (yi)^2 + SUM2Yi
          MOV    @SP,SUM2Yi            ; Sum back to SUM2Yi
          MOV    2(SP),SUM2Yi+2
          .if    DOUBLE=1
          MOV    4(SP),SUM2Yi+4
          .endif
          MOV    (ML/8)+1(SP),RPARG         ; Fetch sample address
          MOV    #SUMYi,RPRES          ; Add yi to SUMYi
          CALL   #FLT_ADD
          MOV    @SP+,SUMYi            ; Summed-up yi
          MOV    @SP+,SUMYi+2          ; House keeping
          .if    DOUBLE=1
          MOV    @SP+,SUM2Yi+4
```

```
            .endif
        ADD     #2,SP               ; Remove sample address
        INC     N                   ; Increment N
        RET
;
; STAT_CALC calculates the Mean Value, the Variance and the
; Standard Deviation from the N samples input to the subroutine
; STAT_PREP.
; The three calculated statistical values are stored in:
; Mean Value:              N
; Variance:               SUM2Yi
; Standard Deviation:       SUMYi
;
STAT_CALC       SUB             #(ML/8)+1,SP     ; Allocate stack space
        MOV     #N,RPARG            ; Convert N to FP-format
        CALL    #CNV_BIN16U         ; Binary to FPP on TOS
        SUB     #(ML/8)+1,SP        ; To save N on stack
        MOV     #SUMYi,RPRES        ; Summed-up yi/N
        CALL    #FLT_DIV            ; Mean Value on TOS
        MOV     @SP,MEANV           ; Store Mean Value
        MOV     2(SP),MEANV+2
        .if     DOUBLE=1
        MOV     4(SP),MEANV+4
        .endif
;
; The Mean Value on TOS is used for the calculation
; of the Variance:
; Variance = (Sum(yi^2) - Mean Value x Sum(yi)/N)/N
;
        MOV     #SUMYi,RPARG        ; Mean Value x Sum(yi)
        CALL    #FLT_MUL            ;
        MOV     #SUM2Yi,RPRES       ; To Sum(yi^2)
        CALL    #FLT_SUB            ; Sum(yi^2) - MV x Sum(yi)
        ADD     #(ML/8)+1,RPARG     ; Point to N
        CALL    #FLT_DIV            ; Variance on TOS
        MOV     @SP,VARIANCE        ; Store Variance
```

```
        MOV     2(SP),VARIANCE+2
        .if     DOUBLE=1
        MOV     4(SP),VARIANCE+4
        .endif
;
; The Variance on TOS is used for the calculation of the
; Standard Deviation: Std Dev. = SQUROOT(Variance x N/(N-1))
;
        ADD     #(ML/8)+1,RPARG   ; Point to N
        CALL    #FLT_MUL          ; Variance x N
        MOV     @SP+,STDDEV       ; Store value for later use
        MOV     @SP+,STDDEV+2
        .if     DOUBLE=1
        MOV     @SP+,STDDEV+4
        .endif
        MOV     #FLT1,RPARG       ; Build N-1
        MOV     SP,RPRES          ; point to N
        CALL    #FLT_SUB          ; N-1 on TOS
        MOV     #STDDEV,RPRES     ; point to (Variance x N)
        CALL    #FLT_DIV          ; Variance x N/(N-1)
        CALL    #FLT_SQRT         ; StdDev = SQROOT(Var x N/(N-1))
        MOV     @SP+,STDDEV       ; Store Standard Deviation
        MOV     @SP+,STDDEV+2
        .if     DOUBLE=1
        MOV     @SP+,STDDEV+4
        .endif
        RET
;
```

EXAMPLE: The normal calling sequence for the statistical calculations is shown in the following. The input samples are contained in the ADC-result register ADAT.

```
        CALL    #STAT_INIT        ; Initialization: clear used RAM
STATLOP MOV     #ADAT,RPARG       ; Set pointer to ADC-result
        CALL    #CNV_BIN16U       ; Convert ADC-result to FP on TOS
        CALL    #STAT_PREP        ; Process samples y1 to yN
        ....                      ; Continue
```

```
        CMP.B   #xx,N           ; yN processed?
        JLO     STATLOP         ; No, next sample
;
; N samples are pre-processed: Calculate Mean Value, Variance,
; and Standard Deviation out of SUMYi, SUM2Yl and N
;
        CALL    #STAT_CALC      ; Call calculation subroutine
        ....                    ; Results in sample locations
```

5.6.10.12 Complex Calculations

Complex numbers of the form (a + jb) can be used in calculations also. The four basic arithmetic operations are shown for complex numbers. Pointers RPARG and RPRES are used in the same way as with the normal FPP subroutines. They point to the real parts of the complex numbers used for input and to the result on the TOS after the completion of the subroutine. The real and imaginary part of a complex number need to be allocated in the way shown in Figure 5–30 (shown for .FLOAT format).

Stack Usage: The subroutines need up to 36 bytes (.DOUBLE) or 28 bytes (.FLOAT) of stack space (complex division). Not included in this numbers is the initially allocated result space. No error handling is provided. It is assumed that the numbers used stay within the range of the floating-point package.
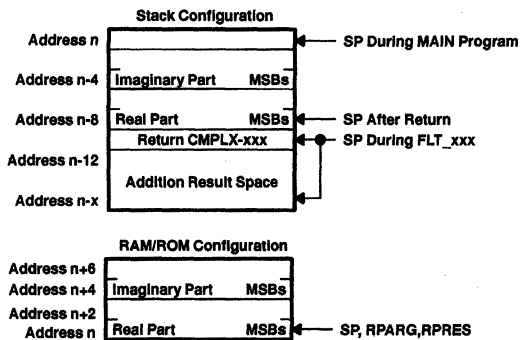
Figure 5–30. Complex Number on TOS and in Memory (.FLOAT Format)

```
FPL     .equ    (ML/8)+1        ; Length of an FP-number (bytes)
;
; Complex calculation is made with the complex numbers RPARG
```

```
; and RPRES point to.
;
; Call:  MOV     #arg1,RPRES      ; Address argument1
;        MOV     #arg2,RPARG      ; Address argument2
;        CALL    #CMPLX_xxx       ; Calculate arg1 op arg2
;        ...                      ; Result on TOS. Pointed to
;        ...                      ; by SP, RPARG and RPRES
;
; Complex Subtraction: (a + jb) - (c + jd). @RPRES - @RPARG.
; Stack Usage: 16 bytes (.DOUBLE)  12 bytes (.FLOAT)
;
CMPLX_SUB MOV     #0FFFFh,HELP    ; Define subtraction
          JMP     CL$1            ; To common part
;
; Complex Addition: (a + jb) + (c + jd). @RPRES + @RPARG
; Stack Usage: 16 bytes (.DOUBLE)  12 bytes (.FLOAT)
;
CMPLX_ADD CLR     HELP            ; Define addition
CL$1      PUSH    RPRES           ; Save argument pointer
          .if     DOUBLE=1
          PUSH    10(RPARG)       ; LSBs imaginary part d
          PUSH    8(RPARG)        ; MIDS imaginary part d
          .endif
          PUSH    6(RPARG)        ; The coming words depend on
          PUSH    4(RPARG)        ; DOUBLE
          PUSH    2(RPARG)        ; LSBs real part c
          PUSH    @RPARG          ; MSBs real part c
          TST     HELP            ; Addition or subtraction?
          JZ      CA
;
; Subtraction: the complex number (c + jd) is negated
;
          XOR     #080h,0(SP)     ; Negate real part c
          XOR     #080h,FPL(SP)   ; Negate imaginary part d
CA        MOV     SP,RPARG        ; Point to c
          CALL    #FLT_ADD        ; Add real parts (a + c)
```

```
        MOV     @SP+,2*FPL+2(SP)  ; To real storage
        MOV     @SP+,2*FPL+2(SP)  ; Housekeeping
        .if     DOUBLE=1
        MOV     @SP+,2*FPL+2(SP)
        .endif
        MOV     SP,RPARG          ; Point to d
        MOV     FPL(SP),RPRES     ; Restore RPRES
        ADD     #FPL,RPRES        ; To imaginary part b
        CALL    #FLT_ADD          ; Add imaginary parts (b + d)
        MOV     @SP+,2*FPL+2(SP)  ; To imaginary storage
        MOV     @SP+,2*FPL+2(SP)  ; Housekeeping
        .if     DOUBLE=1
        MOV     @SP+,2*FPL+2(SP)
        .endif
        ADD     #2,SP             ; Skip saved RPRES
        JMP     CMPLX_RT          ; Result on TOS
;
; Complex Division: (a + jb)/(c + jd). @RPRES/@RPARG
;
; The Complex Division uses the inverted divisor and
; the multiplication afterwards:
; (a + jb)/(c + jd) = (a + jb) x 1/(c + jd)
; with: 1/(c + jd) = (c - jd)/(c^2 + d^2)
;; Stack Usage: 36 bytes (.DOUBLE)   28 bytes (.FLOAT)
;
CMPLX_DIV PUSH  RPRES             ; Save RPRES (dividend a + jb))
        PUSH    RPARG             ; Save RPARG (divisor c + jd)
        SUB     #FPL,SP           ; Allocate result space
        MOV     RPARG,RPRES       ; Fetch real part c
        CALL    #FLT_MUL          ; c^2
        SUB     #FPL,SP           ; Allocate result space
        MOV     2*FPL(SP),RPARG   ; Fetch imaginary part d
        ADD     #FPL,RPARG
        MOV     RPARG,RPRES       ; Copy address of d
        CALL    #FLT_MUL          ; d^2
        ADD     #FPL,RPARG        ; to c^2
```

```
          CALL    #FLT_ADD            ; c^2 + d^2
          PUSH    FPL(SP)             ; Copy c^2 + d^2
          PUSH    FPL(SP)
          .if     DOUBLE=1
          PUSH    FPL(SP)
          .endif
          MOV     SP,RPARG            ; To (c2 +d2)
          MOV     3*FPL(SP),RPRES     ; Pointer to (c + jd)
          ADD     #FPL,RPRES          ; Address d
          CALL    #FLT_DIV            ; d/(c^2 + d^2) imag. part
          XOR     #080h,0(SP)         ; -d/(c^2 + d2)
          MOV     @SP+,2*FPL-2(SP)    ; Store imaginary part
          MOV     @SP+,2*FPL-2(SP)    ; to final location
          .if     DOUBLE=1
          MOV     @SP+,2*FPL-2(SP)
          .endif
          MOV     SP,RPARG            ; To copy of c^2 + d^2
          MOV     2*FPL(SP),RPRES     ; To (c + jd)
          CALL    #FLT_DIV            ; c/(c^2 + d2)
;
; Prepare the interface to the multiplication and call it:
; RPARG points to 1/(c + jd)   yet made by FLT_DIV
; RPRES points to   (a + jb)
;
          MOV     2*FPL+2(SP),RPRES   ; address of (a + jb)
          CALL    #CMPLX_MUL          ; (a +jb) x 1/(c +jd)
          MOV     #FPL,HELP           ; Result to final location
CDIVL     MOV     @SP+,2*FPL+4(SP)
          DEC     HELP
          JNZ     CDIVL
;
          JMP     CMPLX_RET           ; To common housekeeping
;
; Complex Multiplication: (a + jb)x(c + jd). @RPRES x @RPARG
;(a + jb)x(c + jd) = ac + jad + jbc - bd
; Stack Usage: 24 bytes (.DOUBLE)   18 bytes (.FLOAT)
```

```
;
CMPLX_MUL PUSH    RPRES            ; Save pointer to (a + jb)
          PUSH    RPARG            ; Save pointer to (c + jd)
;
; real Part ac - bd
;
          SUB     #FPL,SP          ; Allocate result space for a x c
          CALL    #FLT_MUL         ; a x c
          SUB     #FPL,SP          ; Allocate result space for b x d
          MOV     2*FPL(SP),RPARG  ; To c + jd
          MOV     2*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPARG       ; To jd
          ADD     #FPL,RPRES       ; To jb
          CALL    #FLT_MUL         ; jb x jd = -bd
          ADD     #FPL,RPRES       ; To a x c
          CALL    #FLT_SUB         ; (a x c) - (b x d)
          MOV     @SP,FPL(SP)      ; Store ac - bd
          MOV     2(SP),FPL+2(SP)
          .if     DOUBLE=1
          MOV     4(SP),FPL+4(SP)
          .endif
;
; Imaginary Part j(ad + bc)
;
          MOV     2*FPL(SP),RPARG  ; To c + jd
          MOV     2*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPARG       ; To jd
          CALL    #FLT_MUL         ; a x d
          SUB     #FPL,SP          ; Allocate result space for b x c
          MOV     3*FPL(SP),RPARG  ; To c + jd
          MOV     3*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPRES       ; To b
          CALL    #FLT_MUL         ; b x c
          ADD     #FPL,RPARG       ; To a x d
          CALL    #FLT_ADD         ; ad + bc
          MOV     @SP+,4*FPL+4(SP) ; To imaginary result
```

```
        MOV       @SP+,4*FPL+4(SP)
        .if       DOUBLE=1
        MOV       @SP+,4*FPL+4(SP)
        .endif
        ADD       #FPL,SP          ; To real result
        MOV       @SP+,FPL+4(SP)   ; To real result
        MOV       @SP+,FPL+4(SP)
        .if       DOUBLE=1
        MOV       @SP+,FPL+4(SP)
        .endif
;
; RPARG, RPRES and SP point to the real part of the result
; on the TOS
;
CMPLX_RET ADD     #4,SP            ; Skip pointers
CMPLX_RT MOV      SP,RPARG
        ADD       #2,RPARG
        MOV       RPARG,RPRES
        RET
```

EXAMPLE: The complex number at address CN1 is divided by a complex number at address CN2. The result (on TOS) is added to a RAM value CST3 and stored there.

```
;
        SUB       #2*FPL,SP        ; Allocate result space
        ....
        MOV       #CN1,RPRES       ; Address of CN1
        MOV       #CN2,RPARG       ; Address of CN2
        CALL      #CMPLX_DIV       ; CN1/CN2 -> TOS
        MOV       #CST3,RPARG      ; Address of CST3
        CALL      #CMPLX_ADD       ; CN1/CN2 + CST3 -> TOS
        MOV       @RPARG+,CST3     ; Store result in CST3
        MOV       @RPARG+,CST3+2   ; Save result space
        MOV       @RPARG+,CST3+4
        MOV       @RPARG+,CST3+6
        .if       DOUBLE=1
        MOV       @RPARG+,CST3+8
```

```
MOV         @RPARG+,CST3+10

.endif

...                              ; Continue with complex calc.

ADD         #2*FPL,SP            ; Terminate complex calc.

...
```

5.6.10.13 Trigonometric and Hyperbolic Functions

Four subroutines are shown for the calculation of the sine, cosine, hyperbolic sine, and hyperbolic cosine. All four subroutines use the same kernel, only the initialization part is different for each of them. Expansion in series is used for the calculation. The formulas are (X is expressed in radians):

Sine function:

$$sin\ X\ =\ \sum_{1}^{n} \frac{X^{2n-1}}{(2n-1)!} \times (-1)^{n+1}$$

Cosine function:

$$cos\ X\ =\ \sum_{0}^{n} \frac{X^{2n}}{(2n)!} \times (-1)^{n}$$

Hyperbolic sine function:

$$sinh\ X\ =\ \sum_{1}^{n} \frac{X^{2n-1}}{(2n-1)!}$$

Hyperbolic cosine function:

$$cosh\ X\ =\ \sum_{0}^{n} \frac{X^{2n}}{(2n)!}$$

The number range for X is $\pm 2\pi$ for all four functions. Outside of this range, the error increases relatively fast due to the fast growing terms of the sequences (X^{2n} and X^{2n+1}).

If the trigonometric functions have to be calculated for numbers outside of this range, two possibilities exist:

❏ Sine and cosine: addition or subtraction of 2π until the number X is back in the range $\pm 2\pi$. The subroutine FLT_RNG can be used for this purpose.

❑ Hyperbolic sine and cosine: increase of the software variable Nmax (normally 30.0, see the following software) that defines the number of iterations. If this variable is changed to 120.0 (60 iterations), the deviations in the range 10^{-12} (.DOUBLE format) or. 10^{-6} (.FLOAT format) are possible for X input values up to 65. The input number that delivers results near the maximum numbers $\pm 10^{38}$.

Note:

The following subroutines are optimized for ROM space and accuracy, but not for run time. They are not intended as part of a floating point package, but as a place to begin if needed.

The calculation errors for the trigonometric functions are shown in the following table. They indicate absolute errors; the difference to the correct values.

Table 5–14. Errors of the Trigonometric Functions

Angle X	.FLOAT		.DOUBLE	
	Sin	Cos	Sin	Cos
0	0	-20×10^{-9}	0	0
$\pm\pi/2$	-21×10^{-9}	-64×10^{-9}	0	0
$\pm\pi$	3.8×10^{-9}	-225×10^{-9}	0	0
$\pm 2\pi$	-1.3×10^{-6}	2×10^{-6}	0	-80×10^{-12}

The errors of the hyperbolic functions are shown in the following table. They indicate relative errors. The differences to the correct values are related to the correct values.

Table 5–15. Errors of the Hyperbolic Functions

Angle X	.FLOAT		.DOUBLE	
	Hyperbolic Sine	Hyperbolic Cosine	Hyperbolic Sine	Hyperbolic Cosine
0	0	0	0	0
$\pm\pi/2$	85×10^{-9}	160×10^{-9}	-126×10^{-12}	255×10^{-12}
$\pm\pi$	55×10^{-9}	100×10^{-9}	-242×10^{-12}	-474×10^{-12}
$\pm 2\pi$	34×10^{-9}	218×10^{-9}	-153×10^{-12}	-309×10^{-12}

Calculation times (Nmax = 30.0: 15 iterations). The number of cycles is the same one for all four functions:

.FLOAT with hardware multiplier: 18000 cycles

.FLOAT without hardware multiplier: 26000 cycles

.DOUBLE with hardware multiplier: 28000 cycles

.DOUBLE without hardware multiplier: 42000 cycles

```
; Sine, Cosine, Hyperbolic Sine, Hyperbolic Cosine of X (radians)
;
; Call:  MOV       #addressX,RPARG ; RPARG points to operand X
;        CALL      #FPP_xxx          ; Call the function
;         ...                        ; RPARG, RPRES, SP point to result
;
; Range: -2xPi < X < +2xPi       for larger numbers FAST loss of
; accuracy
; Stack allocation: (4 x FPL + 4) words are needed (Basic FPP
; Functions are included)
;
; Initialization for the trigonometric and hyperbolic functions
;   +---------------+-------+-------+--------+--------+
;   | INIT          | sin X | cos X | sinh X | cosh X |
;   +---------------+-------+-------+--------+--------+
;   | Sign Mask     | 080h  | 080h  | 000h   | 000h   |
;   | n             | 1.0   | 0.0   | 1.0    | 0.0    |
;   | Series Term   |   X   | 1.0   |   X    | 1.0    |
;   | Result Area   |   X   | 1.0   |   X    | 1.0    |
;   +---------------+-------+-------+--------+--------+
;
FPL        .equ      (ML/8)+1        ; Length of FPP numbers (bytes)
;
; Floating Point Sine Function: Result on TOS = SIN(@RPARG)
; Prepare the stack with the initial constants
;
FLT_SIN  PUSH       #80h              ; Sign mask (toggle)
         JMP        SINc
;
; Hyperbolic Sine Function: Result on TOS = SINH(@RPARG)
;
FLT_SINH PUSH       #00h              ; Sign mask (always pos.)
SINc     PUSH       #0                ; n: 1
         .if DOUBLE=1
```

```
          PUSH     #0
          .endif
          PUSH     #08000h        ; .FLOAT 1.0
;
          .if DOUBLE=1
          PUSH     4(RPARG)       ; Series term: X
          .endif
          PUSH     2(RPARG)
          PUSH     @RPARG
          JMP      TRIGCOM        ; To common part
;
; Floating Point Cosine Function: Result on TOS = COS(@RPARG)
; Prepare the stack with the initial constants
;
FLT_COS   PUSH     #80h           ; Sign mask (toggle)
          JMP      COSc
;
; Hyperbolic Cosine Function: Result on TOS = COSH(@RPARG)
;
FLT_COSH  PUSH     #00h           ; Sign mask (always pos.)
COSc      PUSH     #              ; n: 0
          .if DOUBLE=1
          PUSH     #0
          .endif
          PUSH     #00h           ; .FLOAT 0.0
;
          .if DOUBLE=1
          PUSH     #0             ; Series term: 1.0
          .endif
          PUSH     #0
          PUSH     #08000h        ; .FLOAT 1.0
;
; Common part for sin X, cos X, sinh X and cosh X
; The functions are realized by expansions in series
;
TRIGCOM   .equ     $
```

```
        .if DOUBLE=1
        PUSH    4(RPARG)        ; Push X onto stack (gets X^2)
        .endif
        PUSH    2(RPARG)        ; X^2 is calculated once
        PUSH    @RPARG
        MOV     RPARG,RPRES     ; Both pointers to X
        CALL    #FLT_MUL        ; X^2 to actual stack
;
        ADD     #FPL,RPARG      ; Copy series term to result space
        MOV     @RPARG+,3*FPL+4(SP)     ; is X or 1.0
        MOV     @RPARG+,3*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,3*FPL+8(SP)
        .endif
        SUB     #FPL,SP         ; Result space for calculations
        MOV     SP,RPRES
;
; The actual series term is multiplied by X^2/(n+1)x(n+2) to
; get the next series term
;
TRIGLOP MOV     #FLT2,RPARG     ; Address of .FLOAT 2.0
        ADD     #3*FPL,RPRES    ; Address n
        CALL    #FLT_ADD        ; n + 2
        MOV     @RPARG+,3*FPL(SP) ; (n+2) -> n
        MOV     @RPARG+,3*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,3*FPL+4(SP)
        .endif
;
; Build (n+1)x(n+2) for next term. (n+2)^2 - (n+2) = (n+1)x(n+2)
;
        MOV     RPRES,RPARG     ; Both point to (n+2)
        CALL    #FLT_MUL        ; (n+2)^2
        ADD     #3*FPL,RPARG    ; Point to old n
        CALL    #FLT_SUB        ; (n+2)^2 -(n+2) = (n+1)x(n+2)
;
```

```
; The series term is divided by (n+1)x(n+2)
;
        ADD     #2*FPL,RPRES    ; Point to series term
        CALL    #FLT_DIV        ; Series term/(n+1)x(n+2)
        ADD     #FPL,RPARG      ; Point to x^2
        CALL    #FLT_MUL        ; ST x X^2/(n+1)x(n+2)
        JN      TRIGERR         ; Error, status in SR and HELP
;
; The sign of the new series term is modified dependent on
; the sign mask.  0: always positive   080h: alternating + -
;
        XOR     4*FPL(SP),0(SP)  ; Modify sign with sign mask
        MOV     @RPARG+,2*FPL(SP) ; Save new series term
        MOV     @RPARG+,2*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+4(SP)
        .endif
        ADD     #3*FPL+4,RPARG   ; Point to result area
        CALL    #FLT_ADD         ; Old sum + new series term
        MOV     @RPARG+,4*FPL+4(SP)      ; Result to result area
        MOV     @RPARG+,4*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,4*FPL+8(SP)
        .endif
;
; Check if enough iterations are made: iterations = Nmax/2
;
        CMP     Nmax,3*FPL(SP)   ; Compare n with Nmax
        JLO     TRIGLOP          ; Only MSBs are used
;
; Expansion in series done. Error indication (if any) in HELP
; The completion part of the FPP is used
;
TRIGERR ADD     #4*FPL+2,SP      ; Housekeeping: free stack
        BR      #FLT_END         ; To completion part of FPP
;
```

```
        .if DOUBLE=1
FLT2    .DOUBLE 2.0             ; Constant 2.0
        .else
FLT2    .FLOAT   2.0
        .endif
Nmax    .FLOAT   30.0           ; Iterations x 2 (MSBs used only)
```

5.6.10.14 Other Trigonometric and Hyperbolic Functions

With the previous calculated four functions (sin, cosin, hyperbolic sin, and hyperbolic cosin), five other important functions can be calculated: tangent, cotangent, hyperbolic tangent, hyperbolic cotangent, and exponential functions.

$$tan\,X = \frac{sin\,X}{cos\,X} \qquad cot\,X = \frac{cos\,X}{sin\,X} \qquad tanh\,X = \frac{sinh\,X}{cosh\,X} \qquad coth\,X = \frac{cosh\,X}{sinh\,X}$$

$$e^X = \sum \frac{X^n}{n!} = sinh\,X + cosh\,X$$

To calculate one of the five functions, the two functions it consists of are calculated and combined.

The errors of the five functions can be calculated with the errors of the two functions used and are shown in Table 5–14 and Table 5–15:

❏ tan X, cot X, tanh X and coth X: the resulting error is the difference of the two errors

❏ exp X: the resulting error is the sum of the two errors

Calculation times (Nmax = 30.0: 15 iterations). The number of cycles is the same one for all five functions:

.FLOAT with hardware multiplier: 36000 cycles

.FLOAT without hardware multiplier: 52000 cycles

.DOUBLE with hardware multiplier: 56000 cycles

.DOUBLE without hardware multiplier: 84000 cycles

The same software kernel is used for all five functions. The number contained in R4 decides which function is executed. The range for all five functions is ±2π. For larger numbers a relatively fast loss of accuracy occurs.

```
; Tangent of X (radians)
;
; Call:  MOV     #addressX,RPARG   ; RPARG points to operand X
;        CALL    #FLT_TAN          ; Call the tangent function
;        ...                       ; RPARG, RPRES, SP point to result
;
FLT_TAN CLR     R4                 ; Offset for tan X
        JMP     TRI_COM1           ; Go to common handler
;
; Cotangent of X (radians)
;
; Call:  MOV     #addressX,RPARG ; RPARG points to operand X
;        CALL    #FLT_COT          ; Call the cotangent function
;        ...                       ; RPARG, RPRES, SP point to result
;
FLT_COT MOV     #2,R4              ; Offset for cot X
        JMP     TRI_COM1           ; Go to common handler
;
; Hyperbolic Tangent of X (radians)
;
; Call:  MOV     #addressX,RPARG ; RPARG points to operand X
;        CALL  #FLT_TANH           ; Call the hyperbolic tangent
;        ...                       ; RPARG, RPRES, SP point to result
;
FLT_TANH MOV    #4,R4              ; Offset for tanh X
        JMP     TRI_COM1           ; Go to common handler
;
; Hyperbolic Cotangent of X (radians)
;
; Call:  MOV     #addressX,RPARG ; RPARG points to address of X
;        CALL    #FLT_COTH         ; Call the hyperbolic cotangent
;        ...                       ; RPARG, RPRES, SP point to result
;
FLT_COTH MOV    #6,R4              ; Offset for coth X
        JMP     TRI_COM1           ; Go to common handler
;
```

```
; Exponential function of X (ex)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;        CALL     #FLT_EXP        ; Call the exponential function
;        ...                      ; RPARG, RPRES, SP point to result
;
FLT_EXP MOV      #8,R4            ; Offset for exp X
;
; Common Handler for tan, cot, tanh, coth and exponent function
; Range: -2xPi < X < +2xPi. For larger numbers FAST loss of
; accuracy
;
TRI_COM1 .equ $
         MOV      @RPARG+,2(SP)   ; Copy X to result space
         MOV      @RPARG+,4(SP)
         .if      DOUBLE=1
         MOV      @RPARG,6(SP)
         .endif
         SUB      #FPL,SP         ; Allocate new result space
         SUB      #4,RPARG        ; Point to X again
         CALL     FT1(R4)         ; Calculate 1st function
         JN       TERR2           ; Error: error code in HELP
         SUB      #FPL,SP         ; Allocate cosine result space
         ADD      #FPL+2,RPARG    ; Point to X
         CALL     FT2(R4)         ; Calculate 2nd function
         ADD      #FPL,RPRES      ; Point to result of 1st function
         CALL     FT3(R4)         ; 1st result .OP. 2nd result
         MOV      @SP+,2*FPL(SP)  ; Final result to result area
         MOV      @SP+,2*FPL(SP)
         .if      DOUBLE=1
         MOV      @SP+,2*FPL(SP)
         .endif
TERR2    ADD      #FPL,SP         ; Skip 1st result
         BR       #FLT_END        ; Error code in HELP
;
FT1      .word    FLT_SIN         ; tan = sin/cos    1st function
```

```
                .word    FLT_COS           ; cot = cos/sin
                .word    FLT_SINH          ; tanh = sinh/cosh
                .word    FLT_COSH          ; coth = cosh/sinh
                .word    FLT_COSH          ; exp  = cosh + sinh
;
FT2             .word    FLT_COS           ; tan = sin/cos     2nd function
                .word    FLT_SIN           ; cot = cos/sin
                .word    FLT_COSH          ; tanh = sinh/cosh
                .word    FLT_SINH          ; coth = cosh/sinh
                .word    FLT_SINH          ; exp  = cosh + sinh
;
FT3             .word    FLT_DIV           ; tan = sin/cos     3rd function
                .word    FLT_DIV           ; cot = cos/sin
                .word    FLT_DIV           ; tanh = sinh/cosh
                .word    FLT_DIV           ; coth = cosh/sinh
                .word    FLT_ADD           ; exp  = cosh + sinh
```

If the argument X for trigonometric functions is outside of the range $\pm 2\pi$ then the subroutine FLT_RNG may be used. The subroutine moves the angle X into the range $\pm\pi$.

```
; Subroutine FLT_RNG moves angle X into the range -Pi < X < +Pi
;
; Call:  MOV      #addressX,RPARG    ; RPARG points to operand X
;        CALL     #FLT_RNG           ; Call the function
;        ...                         ; RPARG, RPRES, SP point to result
;
; Range: -100xPI < X < +100xPI       loss of accuracy increases with X
;
FLT_RNG PUSH     @RPARG            ; Save sign of X on stack
        AND      #080h,0(SP)       ; Only sign remains
        SUB      #FPL,SP           ; Reserve space for 2^n x Pi
        .if DOUBLE=1
        PUSH     4(RPARG)          ; X on stack
        .endif
        PUSH     2(RPARG)
        PUSH     @RPARG
        BIC      #080h,0(SP)       ; |X| remains
```

```
FR1      MOV       FLT2PI,FPL(SP)      ; 2xPi to stack
         MOV       FLT2PI+2,FPL+2(SP)
         .if DOUBLE=1
         MOV       FLT2PI+4,FPL+4(SP)
         .endif
         CMP       @SP,FLTPI           ; Pi - |X|
         JHS       FR2                 ; Pi > |X|: range process done
;
; Successive approximation by subtracting 2^n x2Pi
;
FR3      INC.B     FPL+1(SP)           ; 2Pi x 2
         CMP       @SP,FPL(SP)         ; 2^n x 2Pi - |X|
         JLO       FR3                 ; 2^n x 2Pi < |X|
         DEC.B     FPL+1(SP)           ; 2^n x 2Pi > |X| divide by 2
         MOV       SP,RPRES            ; Address |X|
         MOV       SP,RPARG
         ADD       #FPL,RPARG          ; Address 2^n x 2Pi
         CALL      #FLT_SUB            ; |X| - 2^n x 2Pi
         JMP       FR1                 ; Check if in range now
;
; Move X (now between -Pi and +Pi) to old result space
;
FR2      XOR       2*FPL(SP),0(SP)     ; Correct sign of X
         MOV       @SP+,2*FPL+2(SP)    ; Result to old RS
         MOV       @SP+,2*FPL+2(SP)
         .if DOUBLE=1
         MOV       @SP+,2*FPL+2(SP)
         .endif
         ADD       #FPL+2,SP           ; To return address of FLT_RNG
         BR        #FLT_END
;
         .if DOUBLE=1
FLTPI    .DOUBLE 3.141592653589793            ; Pi
FLT2PI   .DOUBLE 3.141592653589793*2          ; 2xPi
         .else
FLTPI    .FLOAT  3.141592653589793            ; Pi
```

```
FLT2PI    .FLOAT    3.141592653589793*2        ; 2xPi
          .endif
```

5.6.10.15 Faster Approximations for Trigonometric Functions

If the calculation times of the previous iterations are too long and the high accuracy is not needed (e.g. for the calculation of pulse widths for PWM), tables or cubic equations can be used. The table method is described in the MSP430 Software User's Guide (literature number SLAUE11).

With the following four definition points, a cubic approximation to the sin curve is made. The range is 0 to $\pi/2$. All other angles must be adapted to this range.

X1:= 0.0000000000	SIN X1:= 0.0000000000	(0°)
X2:= 0.3490658504	SIN X2:= 0.3420201433	(20°)
X3:= 1.2217304760	SIN X3:= 0.9396926208	(70°)
X4:= 1.5707963270	SIN X4:= 1.0000000000	(90°)

The resulting multiplication factors are:

$$\text{SIN } X = -0.11316874\ X^3 - 0.063641170\ X^2 + 1.01581976\ X$$

The following results and errors are obtained with the previous factors:

X = 0 ,	SIN X = 0.000000000	0.00%	(0°)
X = $\pi/12$	SIN X = 0.259548457	+0.28%	(15°)
X = $\pi/6$	SIN X = 0.498189297	−0.36%	(30°)
X:=$\pi/4$	SIN X = 0.703738695	−0.47%	(45°)
X:=$\pi/3$	SIN X = 0.864012827	−0.23%	(60°)
X:= $5\pi/12$	SIN X = 0.966827870	+0.09%	(75°)
X:=$\pi/2$	SIN X = 1.000000000	0.00%	(90°)

The error of the previous approximation is within ±0.5% from 0 to 2π. Calculation times:

.FLOAT with hardware multiplier:	880 cycles
.FLOAT without hardware multiplier:	1600 cycles
.DOUBLE with hardware multiplier:	1150 cycles
.DOUBLE without hardware multiplier:	2550 cycles

```
; Sine Approximation: Sin X = A3xX^3 + A2xX^2 + A1xX + A0
; Input range for X: 0 =< X =< Pi/2
; The terms Ax are stored in a table starting with the cubic term
;
          MOV     #X,RPARG         ; Address of X (radians)
          MOV     #A3,R4           ; Address of cubic term for sine
          CALL    #HORNER          ; Cubic approximation
          ...                      ; Use approximated value Sin X
HORNER    .equ    $                ; R4 points to cubic term
          .if     DOUBLE=1         ; Store X on stack
          PUSH    4(RPARG)         ; for later use
          .endif
          PUSH    2(RPARG)
          PUSH    @RPARG
          SUB     #FPL,SP          ; Locate new result space
          MOV     R4,RPRES         ; Address cubic term A3
          CALL    #FLT_MUL         ; XxA3
          ADD     #FPL,R4          ; Address quadratic term A2
          MOV     R4,RPRES         ;
          CALL    #FLT_ADD         ; XxA3 + A2
          ADD     #FPL,RPARG       ; to X
          CALL    #FLT_MUL         ; X^2xA3 + XxA2
          ADD     #FPL,R4          ; Address linear term A1
          MOV     R4,RPRES
          CALL    #FLT_ADD         ; X^2xA3 + XxA2 + A1
          ADD     #FPL,RPARG       ; to X
          CALL    #FLT_MUL         ; X^3xA3 + X^2xA2 + XxA1
          ADD     #FPL,R4          ; Address constant term A0
          MOV     R4,RPRES
          CALL    #FLT_ADD         ; X^3xA3 + X^2xA2 + XxA1 + A0
          MOV     @SP+,2*FPL(SP)   ; Copy to result area
          .if     DOUBLE=1
          MOV     @SP+,2*FPL(SP)
          .endif
          MOV     @SP+,2*FPL(SP)
          ADD     #FPL,SP          ; SP to return address
```

```
        BR      #FLT_END          ; Use standard FPP return
;
; Multiplication factors for the Sine generation (0 to Pi/2)
; SIN X = -0.11316874 X³-0.063641170 X^2 +1.01581976 X
        .if     DOUBLE=1
A3      .DOUBLE -0.11316874       ; cubic term
A2      .DOUBLE -0.063641170      ; quadratic term
A1      .DOUBLE 1.01581976        ; linear term
A0      .DOUBLE 0.0               ; constant term
        .else
A3      .FLOAT  -0.11316874       ; cubic term
A2      .FLOAT  -0.063641170      ; quadratic term
A1      .FLOAT  1.01581976        ; linear term
A0      .FLOAT  0.0               ; constant term
        .endif
```

> **Note:**
>
> The HORNER algorithm (used previously) can be used for several other pur-
> poses. It is only necessary to load the register R4 with the starting address
> of the appropriate block containing the factors (address A3 with the previous
> example).

5.6.10.16 The Natural Logarithm Function

The natural logarithm of a number X is calculated with the following formula:

$$ln\,X \;=\; \sum_{1}^{n} \frac{(X-1)^n}{n} \times (-1)^{n-1}$$

The number range of X for the natural logarithm contains all positive numbers
except zero. Values of X less than or equal to zero return the largest negative
number (-3.4×10^{38}) and the N bit set as an error indication.

The calculation errors for the natural logarithm function are shown in the fol-
lowing table. They indicate relative errors. The errors of the .DOUBLE routine
are estimated: no logarithm values greater than 12 digits were available.
Table 5–16 shows the relative large errors – especially for the .FLOAT format
– for input values X very near to 1.0. This is due to the (X – 1) operation neces-
sary for the calculation. Algorithms used should avoid the calculation of the
logarithm of numbers very close to 1.0.

Table 5–16. Relative Errors of the Natural Logarithm Function

X	.FLOAT	.DOUBLE	Comment
2.938736×10^{-39}	-1.5×10^{-7}	-4.5×10^{-12}	Smallest FPP number
1.00	0	0	
1.0001	-3.5×10^{-5}	-1.4×10^{-8}	Missing resolution
1.00001	$+5.2 \times 10^{-3}$	-8×10^{-8}	at results near zero
1.000001	$+6.7 \times 10^{-2}$	$+2.4 \times 10^{-7}$	See above
1.95	$+1.5 \times 10^{-7}$	$+5 \times 10^{-12}$	
10^6	$+3.6 \times 10^{-8}$	$+1.5 \times 10^{-11}$	
10^{12}	$+3.6 \times 10^{-8}$	$+4.5 \times 10^{-12}$	
3.402823×10^{38}	$+1.5 \times 10^{-7}$	$+4.5 \times 10^{-12}$	Largest FPP number

Calculation times:

.FLOAT with hardware multiplier: 13000 cycles   13 iterations

.FLOAT without hardware multiplier: 16000 cycles

.DOUBLE with hardware multiplier: 34000 cycles   22 iterations

.DOUBLE without hardware multiplier: 43000 cycles

```
; Natural Logarithm Function:      Result on TOS = LN(@RPARG)
;
; Call: MOV      #addressX,RPARG ; RPARG points to operand X
;
;        CALL     #FLT_LN           ; Call the function lnX
;        ...                        ; RPARG, RPRES and SP point to lnX
;
; Range: +2.9x10^-38 < X < +3.4x10^38
;
; Errors: X = 0:  N = 1, C = 1, Z = 0      Result: -3.4E38
;         X < 0:  N = 1, C = 1, Z = 0      Result: -3.4E38
;
; Stack usage: 3 x FPL + 6 bytes
;
FLT_LN   PUSH     #0               ; N binary (divisor, power)
         .if DOUBLE=1
         PUSH     4(RPARG)         ; Push X onto stack
```

```
        .endif
        PUSH    2(RPARG)            ;
        PUSH    @RPARG             ;
;
; Check for the legal range of X: 0 < X
;
        MOV     #FLT0,RPRES        ; Check valid range: 0 < X
        CALL    #FLT_CMP
        JHS     LNNEG              ; X is negative
;
; If X is 1.0 then 0.0 is used for the result
;
        MOV     #FLT1,RPRES        ; Check if X= 1
        CALL    #FLT_CMP
        JEQ     LN1P0              ; X is 1: result is 0.0
;
; The exponent of X is multiplied with ln2. Then ln1.5 is added
; to correct the division by 1.5. Result is base for final result
;
        SUB     #FPL,SP            ; Reserve working space
        MOV.B   1(RPARG),HELP      ; Copy exponent of X
        XOR     #80h,HELP          ; Correct sign of exponent
        SXT     HELP                   ;
        MOV     HELP,0(SP)
        MOV     SP,RPARG
        CALL    #CNV_BIN16         ; Exponent to FP format
        MOV     #FLN2,RPARG        ; To ln2
        CALL    #FLT_MUL           ; exp x ln2
        MOV     #FLN1P5,RPARG      ; To ln1.5
        CALL    #FLT_ADD           ; exp x ln2 + ln1.5
        MOV     @RPARG+,2*FPL+4(SP)        ; To result area
        MOV     @RPARG+,2*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+8(SP)
        .endif
;
```

```
; The mantissa of X is converted into the range -0.33 to +0.33
; to get fast convergion
;
        ADD     #FPL,SP         ; Back to X
        MOV     SP,RPRES        ; RPRES points to X
        MOV.B   #80h,1(SP)      ; 1.0 =< X < 2.0
        MOV     #FLT1P5,RPARG   ; To .FLOAT 1.5
        CALL    #FLT_DIV        ; 2/3 =< X < 4/3
        MOV     #FLT1,RPARG     ; To .FLOAT 1.0
        CALL    #FLT_SUB        ; -1/3 =< X < +1/3
;
        .if DOUBLE=1            ;
        PUSH    #0              ; 1.0 to X^N area
        .endif
        PUSH    #0
        PUSH    FLT1
;
        .if DOUBLE=1            ; N (FLT1.0) on stack
        PUSH    #0
        .endif
        PUSH    #0
        PUSH    FLT1
        SUB     #FPL,SP         ; Working area
;
LNLOP   .equ $
        MOV     SP,RPRES
        ADD     #2*FPL,RPRES    ; To X^N
        MOV     SP,RPARG
        ADD     #3*FPL,RPARG    ; To X
        CALL    #FLT_MUL        ; X^(N+1)
        MOV     @RPARG+,2*FPL(SP) ; New X^(N+1) -> X^N
        MOV     @RPARG+,2*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+4(SP)         ; RPARG points to N
        .endif
        CALL    #FLT_DIV        ; X^N/N
```

```
        INC     4*FPL(SP)         ; Incr. binary N
        BIT     #1,4*FPL(SP)      ; N even?
        JNZ     LN1
        XOR     #80h,0(SP)        ; Yes, change sign of X^N/N
;
LN1     ADD     #4*FPL+4,RPARG           ; Point to result area
        CALL    #FLT_ADD                 ; Old result + new one
        MOV     @RPARG+,4*FPL+4(SP)      ; New result to result area
        MOV     @RPARG+,4*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,4*FPL+8(SP)
        .endif
;
; Float N is incremented
;
        MOV     #FLT1,RPARG       ; To .FLOAT 1.0
        ADD     #FPL,RPRES        ; To N
        CALL    #FLT_ADD
        MOV     @RPARG+,FPL(SP)   ; N+1 to N area
        MOV     @RPARG+,FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,FPL+4(SP)
        .endif
;
; Check if enough iterations are made
;
        CMP     #LNIT,4*FPL(SP)   ; Compare with nec. iterations
        JLO     LNLOP             ; HELP = 0
;
        ADD     #4*FPL+2,SP       ; Housekeeping: free stack
LNE     BR      #FLT_END          ; To completion. Error in HELP
;
LN1P0   ADD     #FPL+2,SP         ; X = 1: result = 0
        BR      #RES0
LNNEG   ADD     #FPL+2,SP         ; X <= 0: -3.4E38 result
        MOV     #0FFFFh,2(SP)     ; MSBs negative
```

```
          BR       #DBL_OVERFLOW
          .if DOUBLE=1
FLT0      .DOUBLE  0.0              ; 0.0
FLT1      .DOUBLE  1.0              ; 1.0
FLT1P5    .DOUBLE  1.5              ; 1.5
FLN1P5    .DOUBLE  0.405465108107   ; ln1.5
FLN2      .DOUBLE  0.6931471805599  ; ln2.0
LNIT      .equ     22               ; Number of iterations
          .else
FLT0      .FLOAT   0.0
FLT1      .FLOAT   1.0
FLT1P5    .FLOAT   1.5
FLN1P5    .FLOAT   0.405465108107
FLN2      .FLOAT   0.6931471805599
LNIT      .equ     13
          .endif
```

To calculate the logarithm of X based to the number 10 the following sequence may be used:

```
          MOV      #addressX,RPARG  ; Address of X
          CALL     #FLT_LN          ; Calculate lnX
          MOV      #FLTMOD,RPARG
          CALL     #FLT_MUL         ; lnX/ln10 = logX
          ...                       ; logX on TOS
          .if      DOUBLE=0
FLTMOD    .FLOAT   0.4342944819033  ; log10/ln10
          .else
FLTMOD    .DOUBLE  0.4342944819033  ; log10/ln10
          .endif
```

5.6.10.17 The Exponential Function

The exponential function e^X is calculated. The number range of X is: $-88.72 \leq X \leq +88.72$. Values of X outside of this range return zero ($X < -88.72$) respective the largest positive number ($+3.4 \times 10^{38}$) and the N bit set as an error indication.

The calculation errors for the exponential function are shown in the following table. They indicate relative errors.

Table 5–17.   Errors of the Exponential Function

X	.FLOAT	.DOUBLE	Result
−88.72	-9.4×10^{-7}	-4.5×10^{-11}	$2.9470911 \times 10^{-39}$
−12.3456	-1.7×10^{-8}	-1.5×10^{-11}	$4.348846154014 \times 10^{-6}$
0.0	0	0	1.0
2^{-41}	-4.5×10^{-13}	-4.5×10^{-13}	1.0
2^{-25}	-30×10^{-9}		$1.0 + 29.8 \times 10^{-9}$
+88.72	-2.8×10^{-6}	-4.5×10^{-11}	Most positive FPP number

Calculation times:

.FLOAT with hardware multiplier: 3200 cycles

.FLOAT without hardware multiplier: 5100 cycles

.DOUBLE with hardware multiplier: 4500 cycles

.DOUBLE without hardware multiplier: 7500 cycles

```
; Exponential Function: e^X.        Result on TOS = e^(@RPARG)
;
; Call:  MOV     #addressX,RPARG  ; RPARG points to operand X
;        CALL    #FLT_EXP         ; Call the exp. function
;        ...                      ; RPARG, RPRES, SP point to result
;
; Range: -88.72 < X < +88.72
;
; Errors:
; X > +88.72: N = 1, C = 1, Z = 1  Result: +3.4E38
; X < -88.72: N = 1, C = 0, Z = 0  Result: 0.0 if SW_UFLOW = 1
;            N = 0, C = x, Z = x  Result: 0.0 if SW_UFLOW = 0
;
; Stack usage:   3 x FPL + 4 bytes
;
FLT_EXP MOV     @RPARG+,2(SP)    ; Copy X to result area
        MOV     @RPARG+,4(SP)
        .if DOUBLE=1
        MOV     @RPARG,6(SP)
        .endif
```

```
;
; Check if X is inside limits: -88.72 < X < +88.72   (8631h,7218h)
;
          MOV       2(SP),COUNTER       ; MSBs, exp and sign of X
          BIC       #080h,COUNTER       ; |X|
          CMP       #08631h,COUNTER     ; |X| > 88.72? ln3.4x10^38=88.72
          JLO       EXP_L3              ; |X| is in range
          JNE       EXP_RNGOUT          ; X > 88.72 .or. X < -88.72: error
          CMP       #07217h,4(SP)       ; Check LSBs
          JHS       EXP_RNGOUT          ; LSBs show: |X| > 88.72
;
; Prepare exponent of result: N = X/ln2   (rounded)
;
EXP_L3    MOV       SP,RPRES
          SUB       #FPL,SP             ; New working area
          ADD       #2,RPRES            ; To X (result area)
          MOV       #FLTLN2I,RPARG      ; To 2/ln2 (allows MPY)
          CALL      #FLT_MUL            ; 2 x X/ln2
          CALL      #CNV_FP_BIN         ; 2 x X/ln2 -> binary
          .if       DOUBLE=1
          SUB       #2,SP               ; LSBs contain N
          ADD       #FPL-2,RPARG        ; To N
          .else
          ADD       #FPL,RPARG          ; To binary N
          .endif                        ; N is at correct place yet
          RRA       @RPARG              ; /2 for rounding
          JNC       EXPL1               ; No carry, no rounding
          TST       0(RPARG)            ; Sign of N
          JN        EXPL1
          INC       0(RPARG)            ; Round N
EXPL1     CALL      #CNV_BIN16          ; N -> FPP format Xn
;
; Calculation of g: g = X - Xn*(C1 + C2)
;
          MOV       #EXPC,RPARG         ; C1 + C2
          CALL      #FLT_MUL            ; Xn*(C1 + C2)
```

```
          ADD       #FPL+4,RPRES      ; To X
          CALL      #FLT_SUB          ; g = X - Xn*(C1 + C2)
;
; Calculation of mantissa R(g):
; R(g) = 0.5 + g*P(z)/(Q(z) - g*P(z))
;
          SUB       #FPL,SP           ; Area for z = g^2
          CALL      #FLT_MUL          ; z = g^2
;
; Calculation of g*P(z): g*P(z) = g*(p1*z + p0)
;
          SUB       #FPL,SP           ; Area for g*P(z)
          MOV       #EXPP1,RPARG      ; To p1, RPRES points to z
          CALL      #FLT_MUL          ; p1*z
          MOV       #EXPP0,RPARG      ; To p0
          CALL      #FLT_ADD          ; p1*z + p0
          ADD       #2*FPL,RPARG      ; To g
          CALL      #FLT_MUL          ; g*P(z) = g*(p1*z + p0)
          MOV       @SP+,2*FPL-2(SP)  ; Store g*P(z)
          MOV       @SP+,2*FPL-2(SP)
          .if       DOUBLE=1
          MOV       @SP+,2*FPL-2(SP)
          .endif
;
; Calculation of Q(z): Q(z) = (q1*z + q0)          .FLOAT format
;                      Q(z) = (q2*z + q1)*z + q0   .DOUBLE format
;
          SUB       #FPL,SP           ; Area for Q(z)
          .if       DOUBLE=1          ; Quadratic equation
          MOV       #EXPQ2,RPARG      ; To q2
          ADD       #FPL,RPRES        ; To z
          CALL      #FLT_MUL          ; q2*z
          MOV       #EXPQ1,RPARG      ; To q1
          CALL      #FLT_ADD          ; q2*z + q1
          .else                       ; Linear equation
          MOV       #EXPQ1,RPARG      ; To q1
```

```
          .endif
          ADD       #FPL,RPRES        ; To z
          CALL      #FLT_MUL          ; (q2*z + q1)*z  resp. q1*z
          MOV       #EXPQ0,RPARG      ; To q0
          CALL      #FLT_ADD          ; (q2*z + q1)*z + q0 or q1*z + q0
;
; Result mantissa R(g) = 0.5 + g*P(z)/(Q(z) - g*P(z))
;
          ADD       #2*FPL,RPARG      ; To g*P(z), RPRES to Q(z)
          CALL      #FLT_SUB          ; Q(z) - g*P(z)
          ADD       #2*FPL,RPRES      ; To g*P(z)
          CALL      #FLT_DIV          ; g*P(z)/(Q(z) - g*P(z))
          MOV       #FLT0P5,RPARG     ; To 0.5
          CALL      #FLT_ADD          ; R(g)=0.5 + g*P(z)/(Q(z)-g*P(z))
          MOV       @SP+,3*FPL+2(SP)  ; Store R(g) to result area
          MOV       @SP+,3*FPL+2(SP)
          .if       DOUBLE=1
          MOV       @SP+,3*FPL+2(SP)
          .endif
;
; Insert exponent N+1 to result
;
          ADD       #2*FPL,SP         ; To binary N
          SETC                        ; N + 1
          ADDC.B    @SP+,3(SP)        ; Add N + 1 to exponent of result
          BR        #FLT_END          ; To normal return, HELP = 0
;
; X is out of range: test if overflow (+) or underflow (-)
;
EXP_RNGOUT TST.B 2(SP)                ; Overflow? (sign positive)
          JGE       EXP_OVFL          ; Yes, error: handling in FPP04
          BR        #DBL_UNDERFLOW    ; Underflow: depends on SW_UFLOW
EXP_OVFL BR         #DBL_OVERFLOW
;
          .if       DOUBLE=1
FLTLN2I   .double +1.4426950408889634074*2    ; 2/ln2
```

```
EXPC     .double +0.693359375-2.1219444005469058277E-4    ; c1+c2
EXPP1    .double +0.595042549776E-2          ;p1
EXPP0    .double  0.24999999999992           ;p0
EXPQ2    .double +0.29729363682E-3           ;q2
EXPQ1    .double +0.5356751764522E-1         ;q1
FLT0P5   .equ    $                           ; both are 0.5
EXPQ0    .double +0.50000000000000E+0        ; q0
         .else
FLTLN2I  .float  +1.4426950408889634074*2
EXPC     .float  +0.693359375-2.1219444005469058277E-4
EXPP1    .float  +0.00416028863
EXPP0    .float   0.24999999950
EXPQ1    .float  +0.04998717878
FLT0P5   .equ    $                           ; both are 0.5
EXPQ0    .float  +0.50000000000
         .endif
```

5.6.10.18 The Power Function

The power function A^B is calculated. The number range for A and B is:

$$2.9 \times 10^{-39} \le A \le 3.4 \times 10^{38}$$
$$-88.72 \le B \times \ln A \le +88.72$$

For the error handling, see the header of the software.

The used formula is:

$$A^B = e^{B \times \ln A}$$

The calculation errors for the power function are shown in the following table. They indicate relative errors.

Table 5–18.   Relative Errors of the Power Function

X	.FLOAT	.DOUBLE	Result
1^1	0	0	1.0
$(3.4 \times 10^{38})^0$	0	0	1.0
$(5.5 \times 10^{-20})^2$	-9×10^{-7}	0	3.025×10^{-39}
1.00007^{88}	$-4. \times 10^{-6}$	-9×10^{-10}	1.0061788
1.000071^{267513}	-5.5%	-7×10^{-7}	3.4027×10^{38}

0^5	0	0	0.0
0.1^{-5}	-1.3×10^{-7}	-1.6×10^{-10}	10^5

The previous table shows the large errors for small bases raised by very large exponents. This is due to the natural logarithm function.

Calculation times:

.FLOAT with hardware multiplier: 17000 cycles

.FLOAT without hardware multiplier: 20000 cycles

.DOUBLE with hardware multiplier: 40000 cycles

.DOUBLE without hardware multiplier: 50000 cycles

```
; Power Function: A^B.                        Result on TOS = (@RPRES)^(@RPARG)
;
; Call:  MOV     #addressA,RPRES  ; RPRES points to operand A
;        MOV     #addressB,RPARG  ; RPARG points to operand B
;        CALL    #FLT_POWR        ; Call the power function
;        ...                      ; RPARG, RPRES and SP point to A^B
;
; Range: 2.9x10^-39 < A < 3.4x10^+38
;        -88.72 < BxlnA < +88.72
;
; Errors:        A < 0:           N = 1, C = 1, Z = 0  Result: -3.4E38
;        B x lnA > +88.72: N = 1, C = 1, Z = 1  Result: +3.4E38
;        B x lnA < -88.72:
;          N = 1, C = 0, Z = 0: Result: 0.0 if SW_UFLOW = 1
;          N = 0, C = x, Z = x: Result: 0.0 if SW_UFLOW = 0
;          B x lnA > 3.4E38:    Error handling of multiplication
;
; Stack:  FPL + 4 + (3 x FPL + 8) bytes
;
FLT_POWR .equ    $
         .if     DOUBLE=1
         TST     4(RPRES)         ; Check if A = 0
         JNZ     PWRL1
         .endif
```

```
        TST     2(RPRES)
        JNZ     PWRL1           ; A # 0
        TST     0(RPRES)
        JZ      POWR0           ; A = 0: result = 0
;
PWRL1   PUSH    RPARG           ; Save pointer to exponent B
        SUB     #FPL,SP         ; Working area
        MOV     RPRES,RPARG     ; Pointer to base A
        CALL    #FLT_LN         ; lnA
        JN      PWERR           ; A is negative
        MOV     FPL(SP),RPARG   ; Pointer to exponent
        CALL    #FLT_MUL        ; BxlnA
        JN      PWERR           ; B is too large. HELP # 0
        CALL    #FLT_EXP        ; e^(BxlnA) = A^B
PWERR   MOV     @SP+,FPL+2(SP)  ; To result area
        MOV     @SP+,FPL+2(SP)
        .if     DOUBLE=1
        MOV     @SP+,FPL+2(SP)
        .endif
        ADD     #2,SP           ; Skip exponent pointer
        BR      #FLT_END        ; Error code in HELP
;
POWR0   BR      #RES0           ; A = 0: A^B = 0
```

5.7 Battery Check and Power Fail Detection

The detection of the near loss of the supply voltage is shown for battery driven and for ac–powered MSP430 systems.

Described in the following section are several methods of how to check if the voltage of a battery or an accumulator is above the minimum supply voltage of the MSP430–system. Possibilities are given for the family members having the 14–bit ADC on–chip and also for the members without it.

Three ways, with different hardware, are given to detect power fail situations for ac–driven systems.

For all examples, applications, schematics, diagrams, and proven software code are given for a better understanding.

5.7.1 Battery Check

In microcomputer systems driven by a battery or an accumulator it is necessary to detect when the lowest usable supply voltage is reached. A battery check executed in regular time intervals ensures that the supply voltage is still sufficient. If the lowest acceptable voltage is reached, normally with an added security value, a warning can be given with the LCD. The decision algorithms used can be very different:

❑ Simple checks; if the low threshold is reached or not

❑ Sophisticated methods using the speed of the voltage reduction (ΔV/Δt) dependent on the discharge behavior of the actual battery or accumulator type. For even better estimations the temperature of the battery can also be taken into account.

5.7.1.1 Battery Check With the 14–Bit ADC

Due to the ratiometric measurement principle of the ADC, the measured digital value of a constant, known reference voltage is an indication of the supply voltage of the MSP430C32x. The measured value is inversely proportional to the supply voltage Vcc. Figure 5–31 shows the connecting of the voltage reference for all three explained variants.

Using the auto mode of the ADC, the digital result, N, for an analog input voltage Vin is:

$$N = INT \left| \frac{Vin \times 2^{14}}{VsVcc} \right|$$

With a reference voltage Vref (Vin) of 1.2 V, the supply voltage Vcc (exactly VSVcc) can be measured in steps of approximately 0.3 mV near the voltage $Vcc_{min} = 2.5$ V.

Note:

If the other analog parts connected to the SVcc–terminal cause a voltage drop that cannot be neglected, it is recommended that the reference diode be connected to an unused TP-output or an O-output. Otherwise, the resulting voltage drop corrupts the result and the calculated value for Vcc is too small.
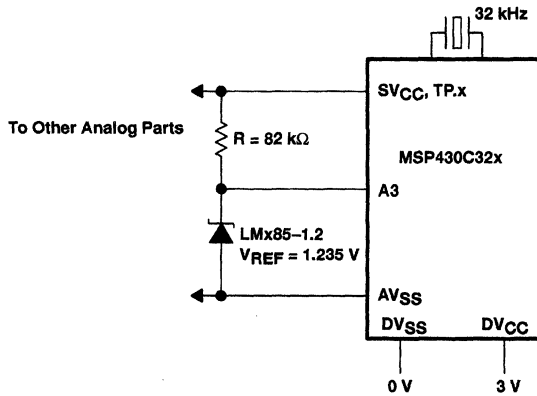
Figure 5–31. Connection of the Voltage Reference

Battery Check With a Reference Measurement

To get the reference for later battery checks, a measurement of the reference voltage (Vref) is made with Vcc = Vcc_{min}. The result is stored in RAM. If the battery should be tested, another measurement is made and the result is compared to the stored value. The result of the comparison determines the status of the battery.

❑ If the actually measured value exceeds the stored one, then Vcc < Vcc_{min} and a battery low indication is given by software.

❑ If the actually measured value is lower than the stored one, then Vcc > Vcc_{min}

EXAMPLE: The battery check with a reference measurement is shown for the analog input A3 (see Figure 5–31). During the calibration, a reference measurement is made with the lowest tolerable Vcc (Vcc$_{min}$). The battery check is then made in regular time intervals (in this example, every hour).

```
; RAM storage for the ADC value measured for Vref with Vccmin
;
ADVref    .EQU    0202h             ; ADC value for Vref at Vccmin
;
; Vccmin (+ security value) is adjusted. A certain code
; at Port0 or a temporary jumper between an input and an
; output leads to this software part
;
        CALL    #MEAS_A3          ; Vref connected to A3
        MOV     &ADAT,ADVref      ; Store reference ADC value
        ...
;
; One hour elapsed: check if Vcc is above Vccmin.
;
        CALL    #MEAS_A3          ; Vref connected to A3
        CMP     ADVref,&ADAT      ; (ADAT) - (ADVref)
        JLO     VCCok             ; Vcc > Vccmin
;
; The actual Vcc is lower than Vccmin. Indicate "Battery
; Low" in the LCD.
;
        CALL    #BATT_LOW         ; Output warning with LCD
VCCok   ...                       ; Continue program
;
; Measurement subroutine for analog input A3. Result in ADAT
;
MEAS_A3 BIC.B   #ADIFG,&IFG2      ; Reset ADC flag ADIFG
        MOV     #ADCLK2+RNGAUTO+CSOFF+A3+VREF+CS,&ACTL
L$101   BIT.B   #ADIFG,&IFG2      ; CONVERSION COMPLETED?
        JZ      L$101             ; IF Z=1: NO
        RET                       ; Yes, return. Result in ADAT
;
```

❑ Advantages

■ Very precise definition of one voltage

■ Small amount of software code

■ Different reference elements possible without software modifications

❑ Disadvantages

■ Calibration necessary

■ Relation to only one supply voltage value is known (calibration voltage)

Battery Check With the Calculation of the Voltage

If no reference measurement during a calibration phase is possible, the value of the supply voltage Vcc can be determined by calculation.

The formula is:

$$Vcc = 2^{14} \times \frac{Vref}{N}$$

With:
N   ADC result of the measurement of Vref
Vref   Voltage of the reference diode   [V]

EXAMPLE: The actual supply voltage (Vcc) needs to be checked. The previous formula is used for the calculation after the measurement of the reference voltage (Vref). The MSP430 floating point package (32-bit .FLOAT version) is used for all calculations. The hardware is shown in Figure 5–31.

```
FPL      .equ    (ML/8)+1          ; Length of FPP number
         ......                    ; Normal program sequence
;
; One hour elapsed: check if Vcc is above Vccmin or not.
;
         CALL    #FLT_SAV          ; Save FPP registers on stack
         SUB     #FPL,SP           ; Allocate stack for result
         CALL    #MEAS_A3          ; Measure ref. diode at A3 (N)
         MOV     #ADAT,RPARG       ; Address of ADC result
         CALL    #CNV_BIN16U       ; Convert ADC result N to FP
;
; Calculate Vcc = 2^14 x Vref/(ADC-Result)
```

```
;
          MOV       #Vref,RPRES        ; Load address of Vref voltage
          CALL      #FLT_DIV           ; Calculate Vref/N (N on TOS)
          ADD.B     #14,1(RPRES)       ; Vcc = 2^14 x Vref/N (exp+14)
          MOV       #VCCmin,RPARG      ; Compare Vcc to VCCmin
          CALL      #FLT_CMP           ; Vcc - VCCmin
          JHS       BATT_ok            ; Vcc > VCCmin: ok
          CALL      #BATT_LOW          ; Give "Battery Low" Indication
BATT_ok   ADD       #FPL,SP            ; Correct SP (result area)
          CALL      #FLT_REC           ; Restore FP registers
          ...                          ; Continue with program
Vref      .FLOAT    1.235              ; Voltage of ref. diode 1.235V
VCCmin    .FLOAT    2.5                ; Vccmin MSP430: 2.5V
```

- ❏ Advantages
  - ■ Battery voltage is known (trend calculation possible)
- ❏ Disadvantages
  - ■ Error of the reference element is not eliminated
  - ■ Calculation takes time

Battery Check With a Fixed Value for Comparison

This method uses a fixed ROM–based value for the decision; if Vcc is sufficient or not. According to the data sheet of the LMx85–1.2, the typical voltage of this reference diode is 1.235 V with a maximum deviation of ±0.012 V. Therefore, the fixed comparison value (Nref) for the minimum supply voltage (Vcc_{min}) can be calculated:

$$Nref = INT \left| \frac{Vref \times 2^{14}}{Vcc_{min}} \right|$$

With Vcc_{min} = 2.5 V and Vref = 1.235 V ± 0,012 V:

$$Nref = INT \left| \frac{(1.235 \pm 0,012 \ V) \times 2^{14}}{2.5 \ V} \right| = 8093 \pm 78$$

To ensure that the voltage of the battery is above Vcc_{min}, the reference value should be set to:

$$N_{ref} = 8093 - 78 = 8015$$

Every measured value below 8015 indicates that the battery voltage is higher than the calculated value, even under worst–case conditions. If the measured value is above 8015, a *Battery Low* warning should be given.

EXAMPLE: The battery check with a fixed value for comparison is executed. The hardware needed is shown in Figure 5–31. The comparison value is stored in ROM at address VCCmin.

```
; One hour elapsed: check if Vcc is above Vccmin or not.
;
        CALL    #MEAS_A3        ; Vref connected to A3
        CMP     VCCmin,&ADAT    ; (ADAT) - (VCCmin)
        JLO     VCCok           ; Vcc > Vccmin
;
; The actual Vcc is lower than Vccmin. Output "Battery
; Low" to the LCD.
;
        CALL    #BATT_LOW       ; Output warning to the LCD
VCCok   ...                     ; Continue program
;
; ROM storage for the calculated ADC value: Vref at Vccmin.
; (worst case value).
;
VCCmin  .WORD   8015            ; ADC value 1.235V at 2.5V
```

❑ Advantages

■ Small amount of software code

❑ Disadvantages

■ Error of the reference element is not eliminated

■ Fixed reference element

■ Relation to only one supply voltage value is known

5.7.1.2 Battery Check With an External Comparator

With an operational amplifier used as a comparator, a simple battery check can be implemented for MSP430 family members that do not have the 14–bit ADC. Figure 5–32 shows two possibilities:

1) On the left, a simple *Go/No Go* solution. The voltage at P0.7 is high, when Vcc is above Vcc$_{min}$ and is low when Vcc is below this voltage. The threshold voltage Vcc$_{min}$ is:

$$Vcc_{min} = Vref \times \left(\frac{R1}{R2} + 1\right)$$

2) On the right, a circuit that allows the comparison of the battery voltage (Vcc) to three different voltage levels; two of them can be determined, the third results from the calculated resistor values for R1, R2 and R3. This allows to distinguish four ranges of the supply voltage:

❏ Vcc < Vth$_{min}$ The supply voltage is below the lowest threshold

❏ Vth$_{min}$ < Vcc < Vth$_{mid}$ The supply voltage is between Vth$_{min}$ and Vth$_{mid}$

❏ Vth$_{mid}$ < Vcc < Vth$_{max}$ The supply voltage is between Vth$_{mid}$ and Vth$_{max}$

❏ Vth$_{max}$ < Vcc   The supply voltage is above the maximum threshold
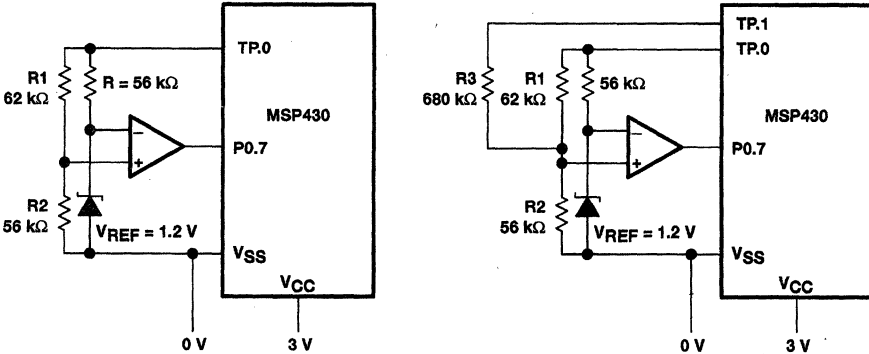
Figure 5–32. Battery Check With an External Comparator

The three different threshold levels are:

❏ Resistor R3 is switched off (TP.1 is switched to Hi–Z):

$$Vth_{mid} = Vref \times \left(\frac{R1}{R2} + 1\right)$$

❑ R3 is switched to Vcc by TP.1:

$$Vth_{min} = Vref \times \ (\frac{R1 \| R3}{R2} + 1)$$

❑ R3 is switched to Vss by TP.1:

$$Vth_{max} = Vref \times \ (\frac{R1}{R2 \| R3} + 1)$$

If the comparator's output (Vout) is high, Vcc is above the selected threshold voltage, if Vout is low, then Vcc is below this voltage.

The calculation of the resistors R1 to R3 starts with the desired threshold voltage (Vth_{mid}), R1 and R2 are derived from it. Then, the low threshold voltage (Vth_{min}) defines the value of R3. The 3rd threshold (Vth_{max}) results from the other two threshold voltages.

The resistor values shown in Figure 5–32 define the following threshold values:

Vth_{min} = 2.52 V   (calculated with second step)

Vth_{mid} = 2.66 V   (calculated first)

Vth_{max} = 2.78 V   (results from the other two thresholds)

EXAMPLE: With the hardware shown in Figure 5–32 (circuit on right side), the actual battery voltage (Vcc) is compared to three different thresholds. This allows the differentiation of four different ranges for Vcc. For any of the four supply levels, different actions are started at the appropriate labels (not shown). Dependent on the speed of the MSP430 and the comparator used, NOPs may be necessary between the setting of the TP-ports and the bit test instructions BIT.B.

```
; One hour elapsed: check the range Vcc falls in now.
;
        BIS.B   #TP0+TP1,&TPD       ; TP.0 and TP1 active high
        BIS.B   #TP0+TP1,&TPE       ; Comparison with Vthmin
        BIT.B   #P07,P0IN           ; Comparator output
        JZ      BATTlo              ; Vcc < Vthmin
        BIC.B   #TP1,&TPE           ; TP.1 to HI-Z
        BIT.B   #P07,P0IN           ; Comparator output
```

```
JZ        BATTmid                 ; Vthmin < Vcc < Vthmid

BIC.B     #TP1,&TPD               ; TP.1 active to Vss

BIS.B     #TP1,&TPE               ; Check Vthmax

BIT.B     #P07,P0IN               ; Comparator output

JNZ       BATThi                  ; Vcc > Vthmax

...                               ; Vthmid < Vcc < Vthmax

;
```

❑ Advantages

■ Four ranges defined (more ranges are possible if desired)

■ Very fast software

■ Different reference elements are possible without software change

❑ Disadvantages

■ Hardware effort (except if an unused operational amplifier of a quad–pack can be used)

5.7.1.3 Battery Check With the Universal Timer/Port Module

The Universal Timer/Port module allows a relatively accurate measurement of the battery voltage (Vcc). The principle (see Figures 5–33 and 5–34) is as follows: the capacitor (C), also used for the other measurements, is charged–up to the voltage (Vref) of the reference diode. C is then discharged with Rref and the time tref until VC reaches the lower threshold VIT– of the input CIN is measured. Afterwards, C is charged–up fully to the supply voltage (Vcc) and the discharge time (tVcc) is also measured. Vcc is then:

$$Vcc = Vref \times e^{\frac{tVcc-tref}{\tau}}$$

With:

Vcc	Actual supply voltage of the MSP430 [V]
Vref	Voltage of the reference diode [V]
tref	Time to discharge C from Vref to VIT– [s]
tVcc	Time to discharge C from Vcc to VIT– [s]
τ	Time constant for discharge: τ = Rref × C [s]
VIT–	Lower threshold voltage of input CIN [V]
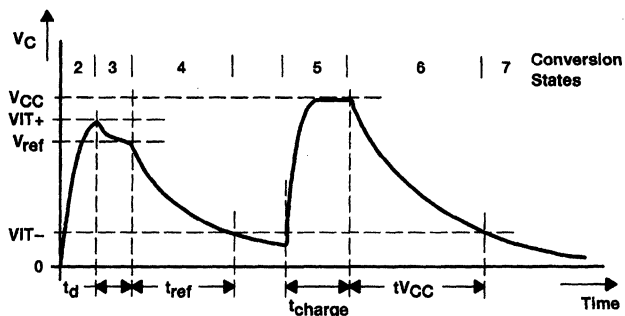
Figure 5–33. Discharge Curves for the Battery Check With the Universal Timer/Port Module

Two hardware possibilities are shown in Figure 5–34:

❏ The left side uses the existing ADC hardware for the battery check too.

❏ The right side uses different battery check hardware. This avoids any influence from the battery check and creates precise ADC–measurement hardware.

See the application report, *Voltage Measurement with the Universal Timer/ Port Module*, in Chapter 2 for more information. Here a formula is given that is independent of the time constant (τ).
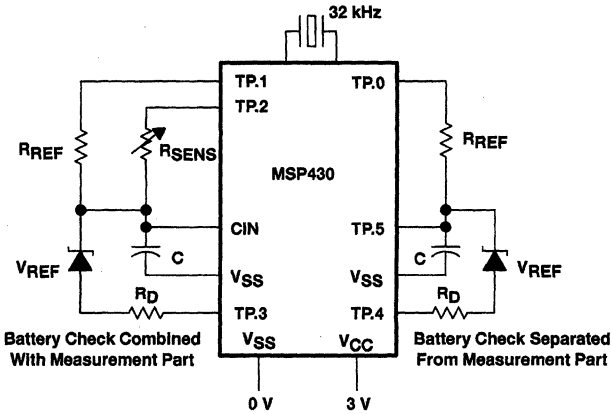
Figure 5–34. Battery Check With the Universal Timer/Port Module

The conditions to be met for the reference voltage (Vref) are:

Vref > VT–   Vref must be higher than the lower threshold voltage VT– of the input CIN at Vcc_{max}

Vref < Vcc_{min} Only voltages above Vref can be measured

The previous conditions mean for an MSP430 system supplied with 3 V: Vref = 1.5 V to 2.5 V.

The measurement sequence like shown in Figure 5–33 is described for the left–side circuitry of Figure 5–34 (the following sequence of numbers refer to the *Conversion States* of Figure 5–33):

1) Switch outputs TP.1 to TP.3 to HI–Z

2) Charge capacitor C with resistor (Rref) until input CIN gets high (or up to Vcc), then switch–off Rref (TP.1 is set to HI–Z)

3) Discharge capacitor C with the reference diode and Rd to Vref (TP.3 is set to LO). Discharge time: td > 5 × Rd × C. Set TP.3 to HI–Z.

4) Discharge capacitor C from Vref to VIT– with Rref (TP.1 set to LO). Measure discharge time tref

5) Charge capacitor C with Rref to Vcc (tcharge > 5 × Rref × C)

6) Discharge capacitor C from Vcc to VIT– with Rref (TP.1 set to LO). Measure discharge time (tVcc)

7) Calculate Vcc with the formula shown previously

For the supply voltage range of a 3–V system (Vcc = 2.5V to 3.5V) and a reference voltage of Vref = 2.3 V, the exponential part of the equation can be replaced by a linear function:

$$Vcc = Vref \times \left(1.29 \times \frac{tVcc - tref}{\tau} + 0.97 \right)$$

If the Universal Timer/Port Module is used in an ADC application with high accuracy (like a heat volume counter) then the battery–check circuitry should be connected to other I/Os as shown in Figure 5–34 on the right side. This way the measurement of the sensors cannot be influenced by the battery–check circuitry.

The software shown in the application report, *Using the MSP430 Universal Timer/Port Module as an Analog-to-Digital Converter* in Chapter 2, can also be used for the battery check with only a few modifications.

❏ Advantages

■ Minimum hardware effort if measurement part exists anyway

■ Supply voltage is known after the measurement

❏ Disadvantages

■ Slow measurement

5.7.2 Power Fail Detection

AC driven systems need a much faster indication of a power–down situation than battery–driven systems. It is a matter of milliseconds, not of hours or days. Therefore, other methods are used. Three of them are described in the following text.

❏ The non–regulated side of the power supply is observed. If the voltage (VC) of the charge capacitor falls below a certain level (VC_{min}), an interrupt is requested.

❏ The voltage at the secondary side of the ac transformer is observed. A sufficient level change there resets the watchdog. If the secondary voltage is too low or ceases, an interrupt is requested.

❏ The non–regulated side of the power supply is observed with a TLC7701. The output of this supply voltage supervisor requests an NMI interrupt or resets the microcomputer.

The interrupt requested by the previous three solutions is used to start the necessary emergency actions:

❏ Switching–off all loads to lengthen the available time for the emergency actions

❏ Reduction of the system clock MCLK to 1 MHz to be able to use Vcc down to Vcc_{min}

❏ Storage of all important values into an external EEPROM

❏ Use of LPM3 finally to bridge the power failure eventually

The three hardware proposals can be used with all members of the MSP430 family. The power–fail detection is also called *ac–low detection*. It issues the *ac–low* signal.

5.7.2.1 Power–Fail Detection by Observation of the Charge Capacitor

Here the voltage level of the charge capacitor (Cch) is observed. If the voltage level of this capacitor falls below a certain voltage level (VC_{min}), an interrupt is requested. With the circuit shown in Figure 5–35, VC_{min} is:

$$VC_{min} = Vcc \times \frac{(\frac{R3}{R4}+1)}{(\frac{R1}{R2}+1)}$$

R1, R2, R3 and R4 are chosen in a way that delivers the desired threshold voltage (VC_{min}). The regulated supply voltage (Vcc) is used as a reference. The NMI (non–maskable interrupt) can be used to get the fastest possible response.

The remaining time (trem) for actions after a power–fail interrupt is approximately:

$$trem = (VC_{min} - Vcc_{min} - Vr) \times \frac{Cch}{I_{AM}}$$

Where:

trem	Approximate time from interrupt to the reaching of Vcc_{min}	[s]
Cch	Capacity of the charge capacitor	[F]
I_{AM}	Supply current of the MSP430 system (medium value)	[A]
VC_{min}	Voltage at the charge capacitor that causes ac–low interrupt	[V]
Vcc_{min}	Lowest supply voltage for the MSP430	[V]
Vr	Dropout voltage (voltage difference between output and input) of the voltage regulator for function	[V]
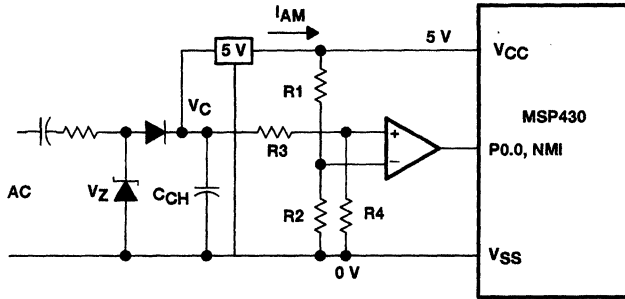
Figure 5–35. Power Fail Detection by Observation of the Charge Capacitor

With the following component values for the hardware shown in Figure 5–35, trem for emergency tasks can be calculated with the following formula.

$C_{CH} = 50 \ \mu F$, $V_{CCmin} = 2.5$ V, Vr = 1 V, IAM = 2 mA, Vz = 10 V, $VC_{min} = 7$ V

$$trem = (7V - 2.5V - 1V) \times \frac{50 \mu F}{2mA} = 87.5ms$$

This remaining time trem = 87.5 ms allows between 14000 and 87500 instructions (dependent on the addressing modes) for the saving of important values in an EEPROM and other emergency tasks.

Note:

The capacitor power supply shown in Figure 5–35 is used only to demonstrate this hardware possibility. A normal transformer supply as shown with the other hardware examples can also be used.
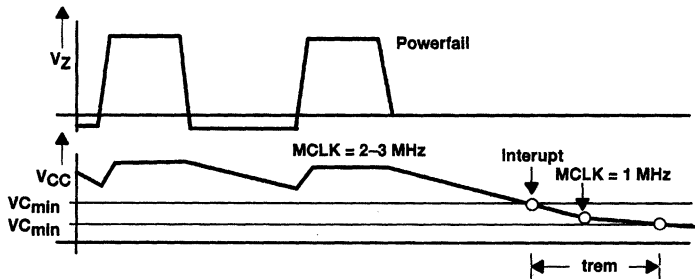
Figure 5–36. Voltages for the Power-Fail Detection by Observation of the Charge Capacitor

The equations shown previously are only valid if the dropout voltage (Vr) of the used voltage regulator (Vr = VC − Vcc) is relatively low. Vr must be:

$$Vr \ < \ VC0 - \frac{Vreg \times VC0}{VC\,min}$$

Where:
VC0 Lowest voltage at Cch that outputs low voltage to the MSP430 input [V]
Vreg Nominal output voltage of the voltage regulator [V]

If this condition for Vr is not possible, then another approach is necessary. Figure 5–37 shows a circuitry that is independent of the previously described restriction.
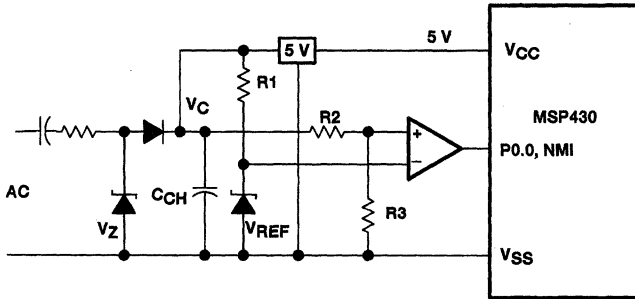
Figure 5–37. Power–Fail Detection by Observation of the Charge Capacitor

The threshold voltage level (VC$_{min}$) for the interrupt is :

$$VC_{min} = Vref \times \left(\frac{R2}{R3} + 1\right)$$

The time remaining (trem) for emergency tasks can be calculated:

$$trem = \left(VC_{min} - Vcc_{min} - Vr\right) \times \frac{Cch}{I_{AM}}$$

If brown outs are a serious problem, the hardware proposal shown in Figure 5–37 can be used with the RESET/NMI terminal as described in Section 5.7.2.3, *Power-Fail Detection with a Supply Voltage Supervisor.* Instead of the inverted RESET output of the TLC7701, the output of the operational amplifier is used.

EXAMPLE: The interrupt handler and its initialization is shown for the power–fail detection by observation of the charge capacitor with a comparator. After the completion of the emergency tasks, a test is made to check if the supply voltage is still low. If not, the software restarts at label PF_INIT. Otherwise, LPM3 is entered to eventually bridge the power failure. The basic timer checks with its interrupt handler in regular intervals for an indication that the voltage is above VC_{min} again. The hardware shown in Figure 5–35 is used.

```
; SYSTAT contains the current system status: calibration,
; normal run, power fail aso.
;
SYSTAT    .EQU    0200h           ; System status byte
;
; The program starts at label INIT if a power-up occurs
;
INIT      ...                     ; Normal initialization
;
; The program restarts at label PF_INIT if the supply voltage
; returns before Vccmin is reached (short power fail)
;
PF_INIT  MOV     #0300h,SP        ; Restart after power fail
         ...                      ; Special initialization
;
; Initialization: Prepare P0.0 for power fail detection.
;
         BIS.B   #P0IFG0,&IE1     ; Enable P0.0 interrupt
         BIS.B   #P00,&P0IES      ; Intrpt for trailing edge
         BIC.B   #P0IFG0,&IFG1    ; Reset flag (safety)
         ...                      ; Continue with initialization
         EINT                     ; Enable GIE
MAINLOOP MOV.B   #NORMAL,SYSTAT   ; Start normal program
         ...                      ;
;
; P0.0 Interrupt Handler: the voltage VC at Cch fell below a
; minimum voltage VCmin. Switch off all loads and interrupts
; except Basic Timer interrupt.
;
P00_HNDLR BIS    #PD,&ACTL        ; ADC to Power down
```

```
            MOV.B    #32-1,&SCFQCTL   ; MCLK back to 1MHz
            BIC.B    #01Ch,&SCFI0     ; DCO current source to 1MHz
            CLR.B    &TPD             ; Reset all TP-ports
            ...                       ; Store values to EEPROM
;
; All tasks are done, return to PF_INIT if Vcc is above Vccmin
; otherwise go to LPM3 to bridge eventually the power fail time
;
            BIT.B    #P00,&P0IN       ; Vcc above Vcmin again?
            JNZ      PF_INIT          ; Yes, restart program
            MOV.B    #PF,SYSTAT       ; System state is "Power Fail"
            BIS      #CPUoff+GIE+SCG1+SCG0,SR   ; Set LPM3
            JMP      PF_INIT          ; Continue here from BT
;
; Basic Timer Interrupt Handler: a check is made for power
; fail: if actual, only the return of Vcc is checked. If Vcc is
; above VCmin, LPM3 is terminated by modification of stack info
;
BT_HNDLR CMP.B      #PF,SYSTAT       ; System in "Power Fail" state?
            JNE      BT$1             ; No, normal system states
            BIT.B    #P00,&P0IN       ; Yes: Vcc above VCmin again?
            JZ       BT_RTI           ; No, return to LPM3
            BIC      #CPUoff+SCG1+SCG0,0(SP)    ; Yes, leave LPM3
BT_RTI   RETI
BT$1     ...                         ; Normal Basic Timer handler
;
            .SECT    "INT_VEC0",0FFE2h
            .WORD    BT_HNDLR         ; Basic Timer Vector
            .SECT    "INT_VEC1",0FFFAh
            .WORD    P00_HNDLR        ; P0.0 Inrtpt Vector
            .WORD    0                ; NMI not used
            .WORD    INIT             ; Reset Vector
```

❏ Advantages

■ Precise due to the use of the +5V regulator voltage for reference purposes

■ Fast response to charge losses

❏ Disadvantages

■ Hardware effort (except an unused operational amplifier of a multiple pack can be used)

5.7.2.2 Power–Fail Detection With the Watchdog

The ac–low detection can also be made with the internal watchdog. The watchdog is reset twice by one half–wave of the ac voltage (Vtr). If this does not occur, due to a power fail, the watchdog initializes the system. The reason for the system reset can be checked during the initialization routine and the necessary emergency actions taken. See the introduction of this section for details of these actions.

The advantage of this method is the unnecessary operational amplifier, the difficulty is to react to *brown–out* conditions. The ac voltage is still active but too low for an error–free run. If a brown out can be excluded or is impossible due to the hardware design, the watchdog solution is a very cheap and reliable possibility for ac–low detection.

If the restricted interval possibilities (only eight discrete time intervals) of the watchdog timer cannot satisfy the system needs, the watchdog timer can be used as a normal timer and the needed interval built by summing–up shorter intervals with software.
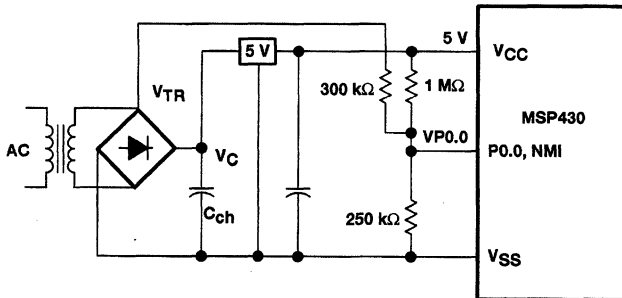
Figure 5–38. Power–Fail Detection With the Watchdog

With the component values shown in Figure 5–38, a square wave out of the ac voltage (Vtr) is reached (the MSP430 inputs have Schmitt–trigger characteristics). The voltages Vtr+ and Vtr– at the transformer output (Vtr) that switch the input voltage at the NMI (or P0.x) input are +7 V and +2 V, respectively. If these two voltage thresholds are carefully adapted to the actual environment, brown–out conditions can also be handled very safely. The equation for trem is:

$$trem \geq \left(Vtr + \ -Vcc_{min} - Vr - Vd\right) \times \frac{Cch}{I_{AM}} - tWD$$

Where:

Vtr+	Transformer voltage that switches the P0.0 input to high [V]
Vd	Voltage drop of one rectifier diode [V]
tWD	Watchdog interval [s]

All other definitions are equal to those explained in Section 5.7.2.1, *Power Fail Detection by Observation of the Charge Capacitor.*
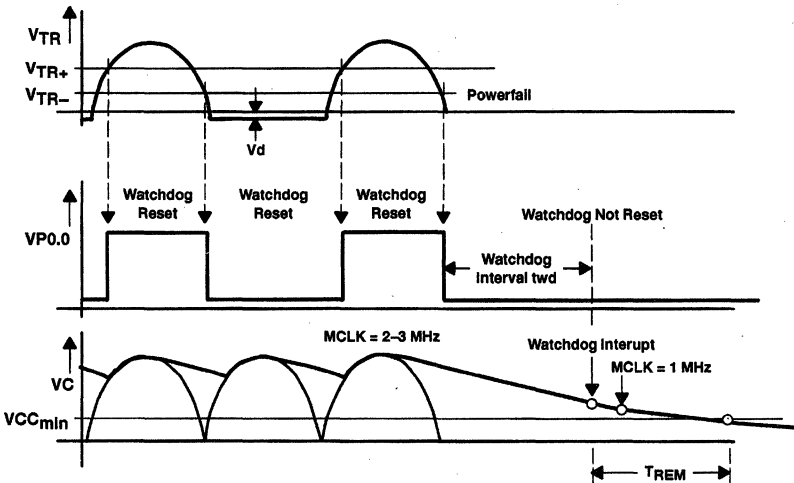
Figure 5–39. Voltages for the Power–Fail Detection With the Watchdog

EXAMPLE: An MSP430 system running with MCLK = 2 MHz uses the watchdog for power–fail detection. The watchdog uses the tap with ($t_{MCLK} \times 2^{15}$) = 16 ms (value after reset). After the completion of the emergency tasks, the soft-

ware checks if the ac voltage is back again with a loop. This is made by check-
ing if P0.0 goes high. If this is the case, the initialization part is entered. The
circuit shown in Figure 5–38 is used.

```
; Power-up and watchdog reset start at label INIT. The reason
; for the reset needs to be known (power-up or watchdog)
;
INIT      BIT.B    #WDTIFG,&IFG1    ; Reset by watchdog?
          JNZ      WD_RESET         ; Yes; power fail
;
; Normal reset caused by RESET pin or power-up: Init. system
;
          BIS.B    #4,&SCFI0        ; Switch DCO to 2MHz drive
          MOV.B    #64-1,&SCFQCTL   ; FLL to 2MHz
          MOV      #05A00h+CNTCL,&WDTCTL    ; Reset watchdog
          BIS.B    #P0IE0,&IE1      ; Enable P0.0 intrpt
          ...                       ; Continue initialization
          EINT                      ; Finally set GIE
MAINLOOP ...                        ; Start main program
;
; Reset caused by watchdog: missing main means power fail
; Supply current is minimized to enlarge active time. All
; interrupts except P0.0 interrupt are switched off
;
WD_RESET BIC.B    #03Fh,&TPD       ; Switch off all TP-outputs
          ...                       ; Switch off other loads
          BIS      #PD,&ACTL        ; Power down ADC
          MOV.B    #32-1,&SCFQCTL   ; MCLK back to 1MHz
          BIC.B    #01Ch,&SCFI0     ; DCO drive to 1MHz, FN_x = 0
          ...                       ; Store values to EEPROM
;
; All tasks are done: check if mains is back (P0.0 gets HI).
;
Llow      BIT.B    #P00,&P0IN       ; Actual state of P0.0 pin
          JZ       Llow             ; Still low
          BR       #INIT            ; P0.0 is HI, initialize
;
```

```
; The P00_HNDLR is called twice each period of the mains
; voltage. The watchdog is reset to indicate normal run,
; the edge selection bit of P0.0 is inverted.
;
P00_HNDLR MOV    #05A00h+CNTCL,&WDTCTL    ; Reset watchdog
          XOR.B  #P00,&P0IES     ; Invert edge select for P0.0
          RETI                   ;
;
          .SECT  "INT_VEC1",0FFFAh
          .WORD  P00_HNDLR        ; P0.0 Inrtpt Vector
          .WORD  0                ; NMI not used
          .WORD  INIT             ; Reset Vector
```

❏ Advantages

■ Minimum hardware effort

■ Minimum software effort

■ Very fast

■ Brown out conditions can be handled by a precise hardware definition

❏ Disadvantages

■ Remaining time trem can be calculated only for worst case

5.7.2.3 Power–Fail Detection With a Supply Voltage Supervisor

For extremely safe MSP430 applications, a TLC7701 supply voltage supervisor can be used. The voltage (VC) of the charge capacitor (Cch) is observed. The output signal /RESET indicates if VC is higher or lower than the threshold voltage (Vth). Figure 5–40 shows the schematic for this application. The output signal /RESET of the TLC7701 is used in two different ways, depending on the actual state of the application.

❏ During power–up, the TLC7701 output is used as a reset signal. The MSP430 is held in the reset state until VC reaches a certain voltage (Vth) (e.g., supply voltage + regulator voltage drop) (see Figure 5–41).

❏ During run mode the RESET/NMI terminal of the MSP430 is switched to NMI–mode (Non–Maskable Interrupt) by software. If VC falls below Vth, an NMI is requested. The interrupt handler can start all necessary emergency tasks. See the introduction of this section for the description of these tasks.

> **Note:**
>
> This method is quite different from the normal use of the TLC7701. If used the normal way, the device outputs a reset signal in case of a Vcc that is too low. This reset signal stops the CPU of the connected microcomputer and gives no opportunity to save important values to an EEPROM.
>
> With the method described, the output of the voltage regulator can also be observed. This allows the use of a TLC7705. The remaining time (trem) is shorter due to the lower threshold voltage used on the output side. For this application, the TPS7350, which includes the voltage regulator and the supply voltage supervisor, is ideally suited.
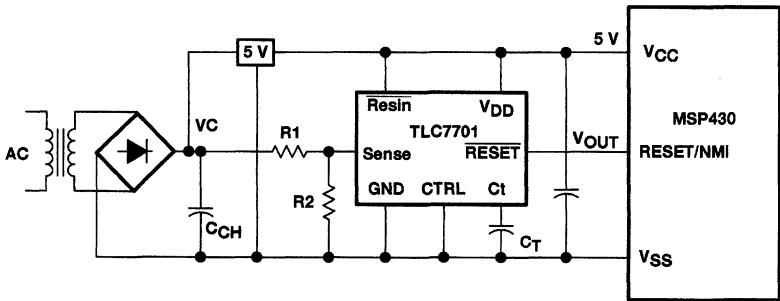
Figure 5–40. Power–Fail Detection with a Supply Voltage Supervisor

Figure 5–41 shows the different system states of the voltage supervisor solution. The voltage (VC) drawing is simplified for a better understanding of the system function. The different *system states* (shown in Figure 5–41) are:

1) The TLC7701 output is low until the voltage (Vth) is reached. The RESET/ NMI input of the MSP430 is a reset input after the power–up, so the MSP430–CPU is inactive.

2) After reaching Vth (and the expiration of the delay trc), the MSP430 starts working and switches the RESET/NMI input to NMI–mode (interrupt input).

3) If VC goes below Vth due to a power fail, an interrupt is requested and the necessary tasks (e.g., EEPROM saving.) are started. Finally the RESET/ NMI terminal is switched to the RESET function.

4) If (as shown in Figure 5–41) the power fail is short in duration (Vout is high again), the software continues at label INIT (after the elapse of trc).

5) If a real power fail occurs, the emergency tasks are completed and the re-set mode for the RESET/NMI terminal is switched on again.

6) This means stop for all MSP430 activities until ac power rises VC above Vth. The MSP430 then restarts with a normal power–up sequence as shown with system state 1.
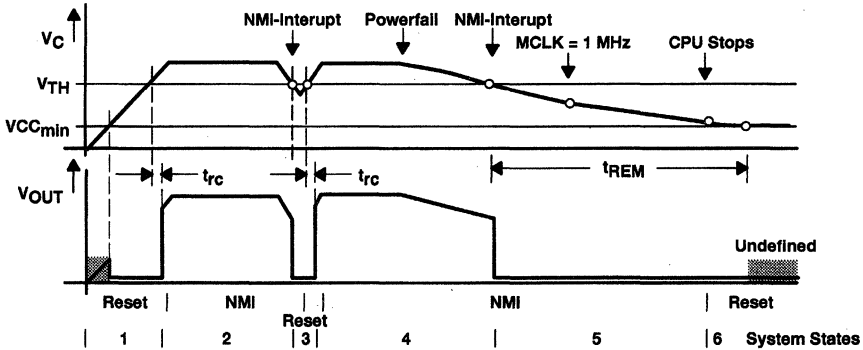
Figure 5–41. Voltages for the Power–Fail Detection With a Supply Supervisor

The formula for the remaining time (trem) is (the time available for emergency tasks):

$$trem = (Vth - Vcc_{min} - Vr) \times \frac{Cch}{I_{AM}}$$

Where:

trem   Approximate time from power-fail interrupt to the reaching of Vcc_{min} [s]

Vth   Threshold voltage for VC. Below this value Vout is low [V]

Vcc_{min} Lowest supply voltage for the MSP430 [V]

Vr   Dropout voltage of the voltage regulator [V]

Cch   Capacity of the charge capacitor Cch [F]

I_{AM} Supply current of the MSP430 system (medium value) [A]

The threshold voltage (Vth) of the TLC7701 can be calculated by:

$$Vth = Vref \times \left(\frac{R1}{R2} + 1\right)$$

Where:

Vref   Voltage of the internal reference diode of the TLC7701: +1.1 V

R2   Resistor from SENSE input to 0 V. Nominal value 100 kΩ to 200 kΩ

The delay (trc) after the return of VC is defined by the capacitor (Ct) shown in Figure 5–40. If this delay is not desired, Ct is omitted. The formula trc is:

$$trc = 21k\Omega \times Ct$$

EXAMPLE: The MSP430 system shown in Figure 5–40 with its initialization and run–time software.

```
; Initialization: prepare RESET/NMI as an NMI interrupt input.
;
INIT     MOV    #05A00h+NMI+NMIES+CNTCL,&WDTCTL  ; 1->0 edge
         ...                          ; Continue with initialization
         EINT                         ; Enable interrupt
MAINLOOP ...                          ; Start normal program here
;
; NMI Interrupt Handler: an oscillator fault or the trailing
; edge of the TLC7701 caused interrupt due to the low input
; voltage VC. Check first the cause of the interrupt.
; The load is reduced to gain time for emergency actions.
;
NMI_HNDLR BIT.B  #OFIFG,IFG1      ; Oscillator fault?
         JNZ    OSCFLT            ; Yes, proceed there
         BIC.B  #03Fh,&TPD        ; Switch off all TP-outputs
         ...                      ; Switch off other loads
         BIS    #PD,&ACTL         ; ADC Power down
         MOV.B  #32-1,&SCFQCTL    ; MCLK back to 1MHz
         BIC.B  #01Ch,&SCFI0      ; DCO drive to 1MHz
         ...                      ; Store values to EEPROM
;
; All tasks are done: switch RESET/NMI to RESET function.
; CPU stops until next power-up sequence. If the TLC7701 output
; is high again (mains back) the program restarts at INIT
;
         MOV    #05A00h+CNTCL,&WDTCTL   ; PC is set to INIT
         BR     #INIT             ; Short power fail: Vcc high
```

```
;
        .SECT    "INT_VEC1",0FFFCh
        .WORD    NMI_HNDLR          ; NMI Vector
        .WORD    INIT               ; Reset Vector
;
```

❑ Advantages

■ Extremely safe: can handle any environment with the appropriate software and hardware combination

❑ Disadvantages

■ Hardware effort (TLC7701 needed)

5.7.3 Conclusion

The concepts shown for battery check and power–fail detection are only possible due to the MSP430's hardware features:

❑ Battery–driven systems can be realized only with microcomputers that need only a very low supply current

❑ In ac–driven systems, the available security of MSP430 systems is due to three unique MSP430 features:

1) The low current consumption allows the remaining charge of the (relatively small) charge capacitor to be used for a lot of emergency tasks in case of a power fail

2) The high speed of the CPU allows to finish all these necessary emergency tasks during the remaining time from power–fail detection to the reaching of the lowest usable supply voltage.

3) The wide supply voltage range (+5.5 V down to +2.5 V) increases the time remaining for these tasks.

These three features together allow relatively simple hardware solutions for MSP430 systems, especially the use of small charge capacitors.

On-Chip Peripherals

6.1 The Basic Timer

The basic timer is normally used as a time base; it is programmed to interrupt the background program at regular time intervals. Table 6–1 shows all possible basic timer interrupt frequencies that can be set by the control bits in byte BTCTL (address 040h). The values shown are for MCLK = 1.048 MHz.

Table 6–1. Basic Timer Interrupt Frequencies

| IP2 | IP1 | IP0 | SSEL=0 | | SSEL=1 | |
			DIV=0	DIV=1	DIV=0	DIV=1
0	0	0	16348 HZ	64 Hz	[524288 Hz]	64 HZ
0	0	1	8192 HZ	32 Hz	[262144 Hz]	32 HZ
0	1	0	4096 HZ	16 Hz	[131072 Hz]	16 HZ
0	1	1	2048 HZ	8 Hz	65536 Hz	8 HZ
1	0	0	1024 HZ	4 Hz	32768 Hz	4 HZ
1	0	1	512 HZ	2 Hz	16348 Hz	2 HZ
1	1	0	256 HZ	1 Hz	8192 Hz	1 HZ
1	1	1	128 HZ	0.5 Hz	4096 Hz	0.5 HZ

Note: Interrupt frequencies shown in [brackets] exceed the maximum allowable frequency and cannot be used.

Example 6–1. Basic Timer Control

```
;
; DEFINITION PART FOR THE BASIC TIMER
;
BTCNT2   .EQU    047h            ; Basic Timer Counter2 (0.5s)
BTCTL    .EQU    040h            ; BASIC TIMER CONTROL BYTE:
SSEL     .EQU    080h            ; 0: ACLK         1: MCLK
RESET    .EQU    040h            ; 0: RUN          1: RESET BT
DIV      .EQU    020h            ; 0: fBT1=fBT     1: fBT1=128Hz
FRFQ     .EQU    008h            ; LCD FREQUENCY DIVIDER
IP       .EQU    001h            ; BT FREQUENCY Selection bits
;
IE2      .EQU    001h            ; INTERRUPT ENABLE BYTE 2:
BTIE     .EQU    080h            ; BT INTERRUPT ENABLE BIT
;
         .BSS    TIMER,4         ; 0.5s COUNTER
         .BSS    BTDTOL,1        ; LAST READ BT VALUE
;
```

```
; INITIALIZATION FOR 1 SECOND TIMING: 32768:(256x128)=1
;
; Input frequency ACLK:          SSEL = 0
; Input division by 256:         DIV = 1
; Add. input division by 128:    IP = 6
; LCD frequency = 128Hz:         FRFQ = 3
;
; Initialization part
;
HLD     .EQU    040h                ; 1: Disable BT
;
        MOV.B   #(DIV+(6*IP)+(3*FRFQ)),&BTCTL ; 1s interval
        BIS.B   #BTIE,&IE2          ; ENABLE INTRPT BASIC TIMER
        ...
;
; INTERRUPT HANDLER BASIC TIMER
; The register BTCNT2 needs to be read twice
;
BTHAN   PUSH    R5                  ; SAVE USED REGISTER
L$300   MOV.B   &BTCNT2,R5          ; READ ACTUAL TIMER VALUE
        CMP.B   &BTCNT2,R5          ; ENSURE DATA INTEGRITY
        JNE     L$300               ; READ AGAIN IF NOT EQUAL
;
; R5 CONTAINS ACTUAL TIMER VALUE, BTDTOL CONTAINS LAST VALUE
; READ. THE DIFFERENCE IS ADDED TO THE 1S COUNTER
;
        PUSH.B  BTDTOL              ; SAVE LAST TIMER VALUE
        MOV.B   R5,BTDTOL           ; ACTUAL VALUE -> LAST VALUE
        SUB.B   @SP+,R5             ; ACTUAL - LAST VALUE -> R5
        ADD     R5,TIMER            ; 16-BIT DIFFERENCE TO COUNTER
        ADC     TIMER+2             ; Carry to high word
        POP     R5                  ; Restore R5
        RETI
;
        .SECT   "Int_Vect",0FFE2h
        .WORD   BTHAN               ; Basic Timer Interrupt Vector
```

6.1.1 Change of the Basic Timer Frequency

If the basic timer is used as a time base (for example as a base for a clock), then it is necessary to compensate if the frequency is changed during the normal run. The necessary operations are different for changing from a faster frequency to a slower one than for the reverse operation. The timer register where the interrupts are counted needs to be implemented for the highest used basic timer frequency.

Slow to fast change: The change should be done only inside the basic timer interrupt routine. The status is to be changed to the new time value.

Fast to slow change: The change should only be done inside the basic timer interrupt routine. Afterward, all bits of the software timer register that represent the higher basic timer frequencies should be reset to zero. This is the correct time for the lower frequency.

Example 6–2. Basic Timer Interrupt Handler

A basic timer interrupt handler that works with two frequencies, 1 Hz and 8 Hz, is shown below. All necessary status routines are shown. The handler may be used for all other possible frequency combinations as well. The background software changes the status according to the needs.

```
HIF       .EQU    8               ; Hi frequency is 8Hz
LOF       .EQU    1               ; Lo frequency is 1Hz
LOBIT     .EQU    HIF/LOF         ; LSB position of low frequency
          .BSS    TIMER,2         ; 16-bit timer register
          .BSS    BTSTAT,1        ; Status byte
;
BT_INT    PUSH    R5              ; Save R5
          MOV.B   BTSTAT,R5       ; R5 contains status (0, 2, 4, 6)
          BR      BTTAB(R5)       ; Got to appropr. routine
BTTAB     .WORD   BT1HZ           ; ST0: 1Hz interrupt
          .WORD   BT8HZ           ; ST2: 8Hz interrupt
          .WORD   CHGT8           ; ST4: Change to 8Hz interrupt
          .WORD   CHGT1           ; ST6: Change to 1Hz interrupt
;
CHGT8     MOV.B   #2,BTSTAT       ; Change to 8Hz interrupt
          BIC.B   #IP2+IP1+IP0,&BTCTL ; Clear frequ. bits
          BIS.B   #IP1+IP0,&BTCTL    ; Set 8Hz, use BT1HZ for INCR.
```

```
BT1HZ    ADD     #LOBIT,TIMER      ; Incr. bit 3 of the 125ms timer
         POP     R5
         RETI                      ; No change of status
;
BT8HZ    INC     TIMER             ; Incr. bit 0 of the 125ms timer
         POP     R5
         RETI                      ; No change of status
;
CHGT1    INC     TIMER             ; Incr. bit 0 (evtl. carry)
         BIC     #LOBIT-1,TIMER    ; Reset 8Hz bits to zero
         MOV.B   #0,BTSTAT         ; New status: 1Hz interrupt
         BIC.B   #IP2+IP1+IP0,&BTCTL ; Clear frequ. bits
         BIS.B   #DIV+IP2+IP1,&BTCTL ; Set 1Hz
         POP     R5
         RETI
;

         .SECT   "Int_Vect",0FFE2h
         .WORD   BT_INT            ; Basic Timer Interrupt Vector
```

6.1.2 Elimination of Crystal Tolerance Error

For normal measurement purposes, the accuracy of 32768 Hz crystals is more than sufficient. But, if highly accurate timing has to be maintained for years, then it is necessary to know the frequency deviation from the exact frequency of the crystal used (together with the oscillator). An example for such an application is an electricity meter that must change the tariff at given times each day without any possibility of synchronizing the internal timer to a reference.

The time deviations for two crystal accuracies (+1 Hz and +10 ppm) are shown in Table 6–2. The data in the table indicates the amount of time required to accumulate a given time error.

Table 6–2. Crystal Accuracy

ACCURACY	DEVIATION = ± 1 S	DEVIATION = ± 1 m	DEVIATION = ± 1 h
32768 Hz, ± 1 Hz	9.10 hours	22.75 days	3.74 years
32768 Hz, ± 10 ppm	27.77 hours	69.44 days	11.40 years

If these time deviations are not acceptable, then a calibration and correction are necessary:

1) The crystal frequency is measured and the deviation stored in the RAM or EEPROM. All other interrupts have to be disabled during this measurement to get correct results.

2) The measured time deviation of the crystal is used for a correction that takes place at regular time intervals.

The crystal frequency can be measured during the calibration with a timing signal of exactly 10 or 16 seconds at one of the ports with interrupt capability. The MSP430 counts its internal oscillator frequency, ACLK, during this time with one of the timers (8-bit timer or 16-bit timer) and gets the deviation to 32768 Hz. The deviation measured is added at appropriate time intervals (32768 s x 10 or 32768 s x 16) to the timer register that counts the seconds.
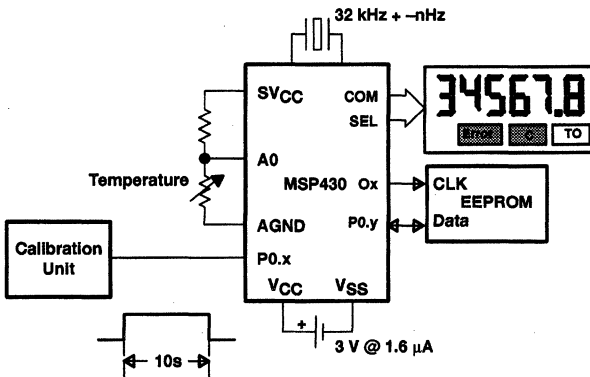
Figure 6–1. Crystal Calibration

If necessary, the temperature behavior of the crystal can also be taken into account. Figure 6–2 shows the typical temperature dependence of a crystal. T_O is the nominal frequency at a particular temperature. Above and below this temperature, the frequency is always lower (negative temperature coefficient). The frequency deviation increases with the square of the temperature deviation (-0.035 ppm/°C^2 for the example).
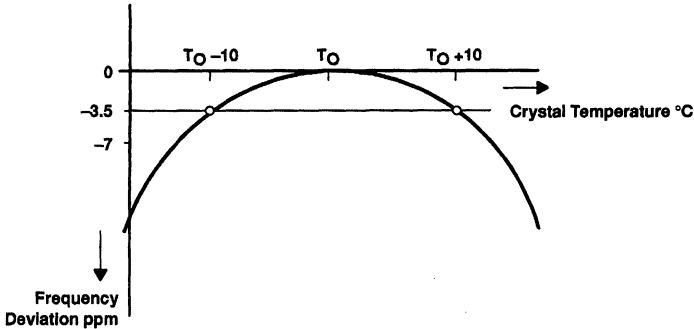
Figure 6–2. Crystal Frequency Deviation With Temperature

The quadratic equation that describes this temperature behavior is approximately (T_O = +19°C):

$$\Delta f = -0.035 \times (T - 19)^2$$

Where:

Δf	Frequency deviation in ppm
T	Crystal temperature in °C

To use the equation shown above, simply measure the crystal temperature (PC board temperature) every hour and calculate the frequency deviation. These deviations are added up until an accumulated deviation of one second is reached. The counter for seconds is then incremented by one and one second is subtracted from the accumulated deviation, leaving the remainder in the accumulation register.

Example 6–3. Quadratic Crystal Temperature Deviation Compensation

The crystal temperature is measured each hour (3600 s) and calculated. The result — with the dimension ppm/1024 — is added up in RAM location PPMS. If PPMS reaches 1024, one second is added to seconds counter SECONDS and PPMS is reduced by 1024. The numbers at the right margin show the digits before and after the assumed decimal point.

```
; Quadratic temperature compensation after each hour:
; tcorr = -|(T-19)^2 x -0.035ppm| x t
; Tmax = To+40C, Tmin = To-40C
;
To      .SET    19                      ; Turning point of temperature
PPM     .SET    35                      ; -0.035ppm/(T-To)^2
        .BSS    PPMS,2                  ; RAM word for adding-up deviation
        .BSS    SECOND,2                ; RAM word for seconds counting
;
TIMCORR CALL    #MEASTEMP               ;Meas. crystal temperature  6.4h
        POP     IROP2L                  ; Result to IROP2L      6.4h
        SUB     #(To*10h),IROP2L        ; T - To        6.4h
        MOV     IROP2L,IROP1            ; Copy result
        CALL    #MPYS                   ; |T-To|^2   (always pos.) 12.8
        CALL    #SHFTRS6                ; Adapt |T-To|^2     12.2
        ADC     IRACL                   ; Rounding
        MOV     IRACL,IROP2L            ; |T-To|^2 -> IROP2L    12.2
;
; tcorr = 3600 x -0.035 x 1E-6 x (T-19)^2  s/h
;
L$006   MOV     #(36*PPM),IROP1         ; 36 x PPM/1E4     ms/h
        CALL    #MPYS                   ; Signed multiplication
;
; IRAC contains: 36s x PPM x 4 (To-T)^2 x 1E-7  s/h
; = 36s x PPM x 4 (To-T)^2 x 1E-4  ms/h
;
        CALL    #SHFTLS6                ; to IRACM
;
; IRACM contains: tcorr = 4 x dT x 36 x PPM/1024
; Correction: 0.25 x 1E-7 x 1024 = 1/39062.5
;
        ADD     IRACM,PPMS              ; Add-up deviation
        CMP     #39062,PPMS             ; One second deviation reached?
        JLO     L$200
        INC     SECONDS                         ; Yes, add one second
        SUB     #39062,PPMS             ; and adjust deviation counter
L$200   RET
```

6.1.3 Clock Subroutines

The following two subroutines provide 24-hour clocks — one using decimal counting (RTCLKD) and one using hexadecimal counting (RTCLK). These subroutines are called every second by the basic timer handler.

```
;
SEC     .EQU    0200H           ; Byte for counting of seconds
MIN     .EQU    0201H           ; Byte for counting of minutes
HOURS   .EQU    0202H           ; Byte for counting of hours
;
; Subroutine provides a decimal clock: 00.00.00 to 23.59.59
;
RTCLKD  SETC                    ; Entry every second
        DADC.B  SEC             ; Increment seconds
        CMP.B   #060H,SEC       ; One minute elapsed?
        JLO     RTRETD          ; No, return (C = 0)
        CLR.B   SEC             ; Yes, clear seconds (C = 1)
        DADC.B  MIN             ; Increment minutes with set carry
        CMP.B   #060H,MIN       ;
        JLO     RTRETD
        CLR.B   MIN
        DADC.B  HOURS
        CMP.B   #024H,HOURS
        JLO     RTRETD
        CLR.B   HOURS           ; 00.00.00 Return to caller
RTRETD  RET                     ; C = 1: one day elapsed
;
; Subroutine provides a hex clock: 00.00.00 to 17.3B.3B
;
RTCLK   INC.B   SEC             ; Entry point every second
        CMP.B   #60,SEC         ; Increment seconds
        JLO     RTRET           ; One minute elapsed?
        CLR.B   SEC             ; No, return to caller
        INC.B   MIN             ; Yes, clear seconds
        CMP.B   #60,MIN         ; Increment minutes
        JLO     RTRET
```

```
        CLR.B   MIN
        INC.B   HOURS
        CMP.B   #24,HOURS
        JLO     RTRET
        CLR.B   HOURS           ; 00.00.00
RTRET   RET                     ; C = 1: one day elapsed
```

The next subroutine increments the date with each call. The handling of leap–years is included. The data is stored in binary format.

```
DAY     .EQU    0203h           ; Day of month 1 - 31 (byte)
MONTH   .EQU    0204h           ; Month 1 - 12 (byte)
YEAR    .EQU    0206h           ; Year 1990 - 2399 (word)
;
DATE    PUSH    R5              ; Save R5
        INC.B   DAY             ; To next day of month
        MOV.B   MONTH,R5        ; Look for length of month
        MOV.B   MT-1(R5),R5
        CMP.B   #2,MONTH        ; February now?
        JNE     NOFEB
        BIT     #3,YEAR         ; Yes, Leap Year?
        JNZ     NOFEB
        INC     R5              ; Yes, 29 days for February
NOFEB   CMP.B   R5,DAY          ; One month elapsed?
        JLO     DATRET          ; No
        MOV.B   #1,DAY          ; Yes, start with 1st day
        INC.B   MONTH           ; of next month
        CMP.B   #13,MONTH       ; Year over?
        JLO     DATRET          ; No
        MOV.B   #1,MONTH        ; Yes, start with 1st month
        INC     YEAR            ; of next year
DATRET  POP     R5              ; Restore R5
        RET
;
; Table with the length of the 12 months
;
```

```
MT        .BYTE    31+1,28+1,31+1,30+1,31+1,30+1 ; January to June
          .BYTE    31+1,31+1,30+1,31+1,30+1,31+1 ; July to December
```

6.1.4 The Basic Timer Used as a 16-Bit Timer

The two 8-bit registers BTCNT1 and BTCNT2 may be connected together and used as a simple 16-bit timer counting the ACLK. This 16-bit value can be used for time measurements by calculating the difference of two readings. The problem is that the two registers cannot be read with just one instruction, so BTCNT1 can overflow between the two readings and deliver an incorrect result. The following software corrects this possible error. If the LSBs change during the register read, then a second reading is made. This second register read is likely correct because of the relatively long time interval (30.5 μs). If interrupts between the readings can occur, then the interrupt can be disabled with the DINT instruction.

```
BTCTL     .equ     040h              ; Basic Timer1 Control Register
DIV       .equ     020h              ; Clock for BTCNT2 is ACLK/256
BTCNT1    .equ     046h              ; LSBs of Basic Timer1
BTCNT2    .equ     047h              ; MSBs of Basic Timer1
;
          MOV.B    #DIV+xx,BTCTL     ; Define BT as a 16-bit counter
          ...
L$1       MOV.B    &BTCNT1,R5        ; Read LSBs of Basic Timer1 00yy
          MOV.B    &BTCNT2,R6        ; Read MSBs 00xx
          CMP.B    &BTCNT1,R5        ; LSBs still the same?
          JNE      L$1               ; No, read once more, 30.5us time
          SWPB     R6                ; Yes, prepare 16-bit result xx00
          ADD      R5,R6             ; Correct result in R6 now: xxyy
```

If the result of the first reading is important, then the following subroutine may be used. The 16-bit value is read and corrected if an overflow to 0 may have happened between the reading of the low and high bytes.

```
; Read-out of the Basic Timer running as a 16-bit timer
;
          MOV.B    &BTCNT1,R5        ; Read LSBs  00yy
          MOV.B    &BTCNT2,R6        ; Read MSBs  00xx
          CMP.B    R5,&BTCNT1        ; BTCNT1 still >= R5?
          JHS      L$1               ; Yes, no overflow
```

```
;
; Transition from 0FFh to 0 occurred with LSBs, read actual
; MSB, it now has the value + 1.
;
          MOV.B    &BTCNT2,R6       ; Read actual MSBs   00xx
          DEC.B    R6               ; MSB - 1 is correct
L$1       SWPB     R6               ; MSBs to high byte   xx00
          ADD      R5,R6            ; 16-bit value to R6: xxyy
```

6.2 The Watchdog Timer

The internal watchdog of the MSP430 family may be used as a simple timer or as a watchdog that ensures system integrity. The watchdog function is enabled after power-on reset or a system reset. This means that if there are difficulties after the start-up of the MSP430, the watchdog will reset the system as often as it is needed for it to start successfully. The watchdog mode is described in this chapter.

6.2.1 Supervision of One Task With the Watchdog

In Section 5.7.2.2 *Power Fail Detection With the Watchdog*, an example is given of how to use the watchdog for the supervision of a power fail task only. This example shows the necessary hardware and the software needed to detect an impending power failure. As long as ac line voltage is present, an interrupt occurs for each polarity change of the ac line. These interrupts reset the watchdog, preventing it from timing out. If the line voltage falls below a certain level or fails completely, these interrupts disappear and the watchdog is not reset. When the watchdog times out, it initializes the MSP430 system.

6.2.2 Supervision of Multiple Tasks With the Watchdog

Normally, the watchdog can only supervise one task at a time. If this task does not reset the watchdog, the MSP430 is initialized by the watchdog. In complex systems, more than one function needs to be supervised to assure correct system functionality. This is possible with a small software effort — each supervised function sets a bit in a RAM byte if it runs correctly. The mainloop then resets the watchdog only if all bits are set. This approach can be enlarged to any number of supervised functions if more than one byte is used.

Example 6–4. Watchdog Supervision of Three Functions

A system running with MCLK = 3 MHz uses the watchdog for the supervision of three functions.

☐ **Power Fail** — by the checking of the 60 Hz AC line (see section *Battery Check and Power Fail Detection* for details)

☐ **Function 1** — a check is made if the software reaches this background part regularly

☐ **Function 3** — a check is made if this interrupt handler is called regularly

Each supervised function sets a dedicated bit in RAM byte WDB in intervals less than 10.66 ms (power-up value of the watchdog with MCLK = 3 MHz) if everything is functioning normally. The mainloop checks this byte (WDB) and resets the watchdog *ONLY* if all three bits are set (07h). If one of the functions fails, the watchdog is not reset and will therefore reset the system.

```
; HARDWARE DEFINITIONS
;
ACTL      .EQU    0114h           ; ADC CONTROL REGISTER:
PD        .EQU    1000h           ; 1: ADC POWERED DOWN
IFG2      .EQU    003h            ; INTERRUPT FLAG REGISTER 2
;
P00       .EQU    001h            ; P0.0 Bit Address
;
IE1       .EQU    000h            ; Intrpt Enable Reg. 1 Addr.
P0IE0     .EQU    004h            ; P0.0 Intrpt Enable Bit
IFG1      .EQU    002h            ; Intrpt Enable Reg. 1 Addr.
P0IFG0    .EQU    004h            ; P0.0 Flag Bit
P0IES     .EQU    014h            ; Intrpt Edge Sel. Reg. Addr.
SCFQCTL   .EQU    052h            ; Sys Clk Frequ. Control Reg.
SCFI0     .EQU    050h            ; Sys Clk Frequ. Integr. Reg.
;
WDTCTL    .EQU    0120h           ; Watchdog Timer Control Reg.
WDTIFG    .EQU    01h             ; Watchdog flag
CNTCL     .EQU    008h            ; Watchdog Clear Bit
WDB       .EQU    0202h           ; RAM byte for functional bits
;
          .TEXT 0E000h            ; Software Start Address
;
; Watchdog reset and Power-up both start at label INIT. The
; reason for the reset needs to be known
;
INIT      BIT.B   #WDTIFG,&IFG1   ; Reset by watchdog?
          JNZ     WD_RESET        ; Yes; check reason
;
; Normal reset caused by RESET pin or power-up: Init. system
;
```

```
INIT1   BIS.B   #8,&SCFI0       ; Switch DCO to 3MHz drive
        MOV.B   #96-1,&SCFQCTL  ; FLL to 3MHz MCLK
        MOV     #05A00h+CNTCL,&WDTCTL ; Define watchdog
;
        BIS.B   #P0IE0,&IE1     ; Enable P0.0 interrupt
        BIS.B   #P00,&P0IES     ; To trailing edge
        BIC.B   #P0IFG0,&IFG1   ; Reset flag (safety)
        ...                     ; Continue initialization
        CLR.B   WDB             ; Clear Functional Bits
        EINT                    ; Enable GIE
        BR      #MAINLOOP       ; Go to MAINLOOP
;
; Reset is caused by watchdog: check reason and handle
; individually
;
WD_RESET MOV.B  WDB,R5          ; Build handler address
        MOV.B   TAB(R5),R5
        SXT     R5              ; Offsets may be negative!
        ADD     R5,PC
TAB     .BYTE   INIT1-TAB       ; All functions failed: hang-up
        .BYTE   PF-TAB          ; power fail and function 3
        .BYTE   F1F3-TAB        ; Function 1 and 3 failed
        .BYTE   F3-TAB          ; Function 3 failed
        .BYTE   PF-TAB          ; Power fail and function 1
        .BYTE   PF-TAB          ; Power fail
        .BYTE   F1-TAB          ; Function 1 failed
        .BYTE   INIT1-TAB       ; All bits set: hang-up
;
; Missing mains voltage means power fail.
; Supply current is minimized to enlarge active time
;
PF      BIC.B   #03Fh,&TPD      ; Switch off all TP-outputs
        ...                     ; Switch off other loads
        BIS     #PD,&ACTL       ; Power down ADC
        MOV.B   #32-1,&SCFQCTL  ; MCLK back to 1MHz
        BIC.B   #01Ch,&SCFI0    ; DCO drive to 1MHz
```

```
          ...                      ; Store values to EEPROM
;
; All tasks are done: LPM3 to bridge eventually the power fail
;
          BIS       #CPUoff+GIE+SCG1+SCG0,SR
          JMP       INIT1          ; Continue here eventually
;
; The handlers for all failures except power fail.
; Every failure can be handled individually
;
F1        ...                      ; Function 1 failed
F3        ...                      ; Function 3 failed
F1F3      ...                      ; Function 1 and 3 failed
;
; Background: Main Loop. If RAM-byte WDB contains 07h then the
; watchdog is reset: all 3 supervised functions are OK.
;
MAINLOOP CMP.B    #07h,WDB          ; Test WDB
          JNE       L$1            ; WDB does not contain 7: continue
          MOV       #05A00h+CNTCL,&WDTCTL ; All OK: reset watchdog
          CLR.B     WDB            ; Clear WDB
L$1       ...                      ; Continue Mainloop
;
; Function 1: if the software reaches this address, the
; supervision bit 1 is set in WDB. This indicates normal run
;
          BIS.B     #1,WDB         ; Set supervision bit 1
          ...
;
; Function 3: if the software reaches this interrupt handler, the
; supervision bit 3 is set in WDB. This indicates normal run
;
INT_HNDLR          ...
          BIS.B     #4,WDB         ; Set supervision bit 3
          RETI
;
```

```
; The P00_HNDLR is called each the mains changes polarity.
; The bit 2 in WDB is set to indicate: "No Power Fail".
;
P00_HNDLR  BIS.B  #2h,WDB          ; Set mains control bit
           XOR.B  #P00,&P0IES      ; Invert edge select for P0.0
           RETI                    ;
;

           .SECT  "INT_VEC1",0FFFAh
           .WORD  P00_HNDLR        ; P0.0 Inrtpt Vector
           .WORD  0                ; NMI not used
           .WORD  INIT             ; Reset Vector
```

The interrupt handler for the watchdog operation can be simplified if a strict priority exists for the processing steps. If, for example, the priority is from power fail (highest priority), to function 3, and function 1 (lowest priority), then the watchdog handler may look like this:

```
; Reset is caused by watchdog: check reason and handle with
; priority from power fail to function 1.
;
WD_RESET BIT.B  #2,WDB          ; Power fail?
         JZ     PF              ; Yes, prepare for it
         BIT.B  #4,WDB          ; Function 3 failed?
         JZ     F3              ; Yes, handle it
         BIT.B  #1,WDB          ; Function 1 failed?
         JZ     F1              ; Yes, handle it
         JMP    INIT1           ; Hang-up occurred (WDB = 7)
```

6.3 The Timer_A

6.3.1 Introduction

The 16-bit Timer_A is a relatively complex timer consisting of the 16-bit timer register and several capture/compare registers. All capture/compare registers are identical, but one of them (CCR0) is used for additional functions. The architecture of the Timer_A shows some similarity to the MSP430 CPU — both of them use the principle of orthogonality (equal features for all registers).

The Timer_A, whose block diagram is shown in Figure 6–3, has several registers available for different tasks. These registers are described in Section 6.3.2 *The Timer_A Hardware.*

Note:

The software and hardware examples shown are related to the MSP430x33x family. Other MSP430 family members may use other I/O ports and addresses for the Timer_A registers and signals. Also, the number of capture/compare registers may be different. The programming principle will stay unchanged; only address definitions need to be modified.

It is recommended that the data book *MSP430 Family Architecture Guide and Module Library* (TI literature number SLAUE10B) be consulted. The hardware related information given there is very valuable and complements the information in this chapter.

The architecture of the Timer_A is not restricted to the configuration shown in Figure 6–3. Different family members of the MSP430 family have different configurations of the Timer_A:

❏ The minimum configuration is the timer register block and the capture/compare block 0. This allows one timing but no pulse width modulation (PWM).

❏ The next possible configuration is the timer register block and the capture/compare blocks 0 and 1. This allows two independent timings or one PWM timing.

❏ The configuration implemented in the MSP430x33x family allows up to five independent timings or three PWM signals and a capture input for the speed control (for a 3-phase digital motor control, for example).

❏ Larger configurations are also possible — eight capture/compare blocks for very complex applications, for example.

The upper limit for the number of capture/compare registers is only the overhead coming from the actualization of the registers and the overhead from the interrupts, themselves.
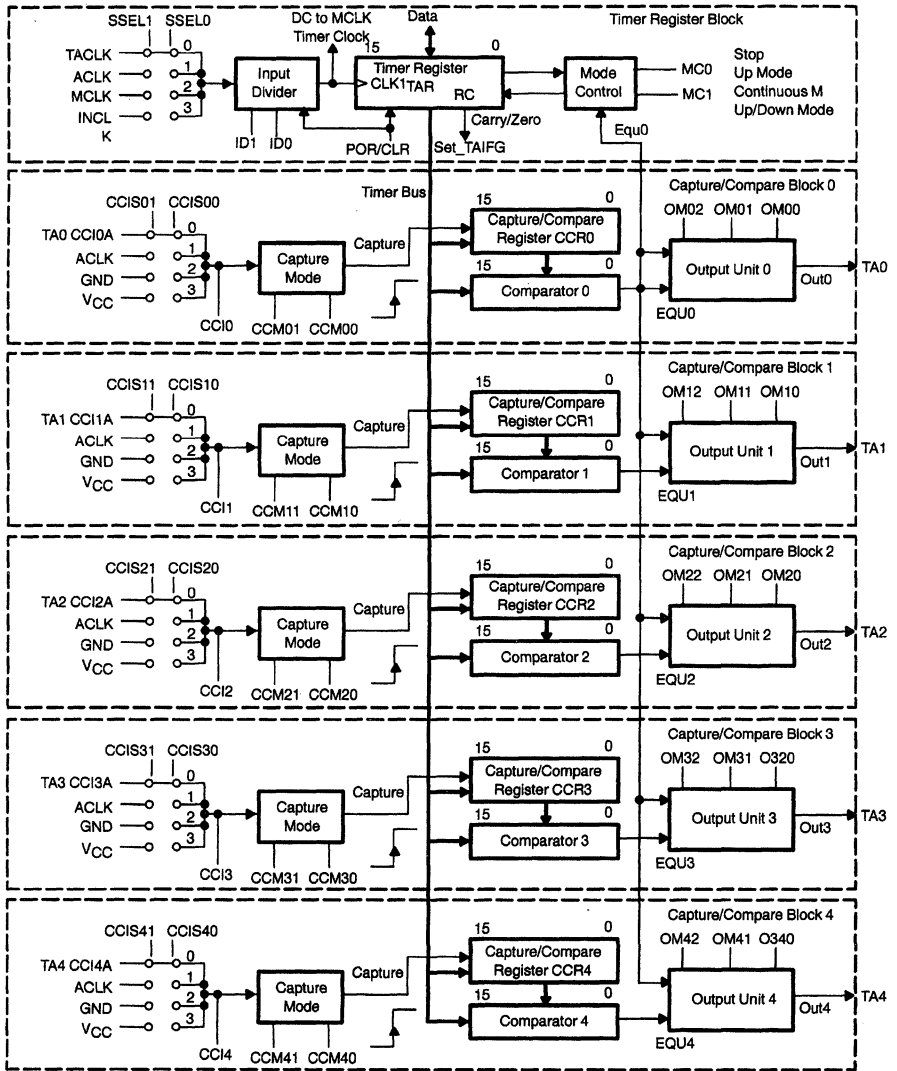
Figure 6–3. The Hardware of the 16-Bit Timer_A (Simplified MSP430x33x Configuration)

Applications for the Timer_A can be:

❏ Generation of up to five independent timings (MSP430x33x configuration)

❏ Frequency generation — using the output units, the internal generated timings can be output to the external periphery of the MSP430

❏ Generation of the timing for RF transmission (amplitude modulation, bi-phase code, biphase space modulation) for the transfer of metered data (gas meters, electric meters, heat allocation meters, etc.)

❏ Realization of a software SPI

❏ Realization of a software UART

❏ Digital motor control (DMC) — the MSP430x33x is able to control a 3-phase electric motor with PWM in closed loop mode

❏ TRIAC control for electric motors and other power applications

❏ Time measurement, period measurement, pulse width measurement

❏ Frequency measurement (using the capture mode for low frequencies)

❏ Analog-to-digital converter (ADC) — a single-slope ADC can be built using the capture mode. Normal I/O ports switch the reference resistors and sensors

❏ DTMF generation — the DTMF frequency pairs can be generated by software and output by three external operational amplifiers for filtering and mixing. See the third part of this chapter for hardware and software details

❏ Crystal replacement — the frequency locked loop (FLL) of the MSP430 may be locked to the ac line frequency instead of the 32-kHz frequency of a crystal. This eliminates the need for a crystal and provides a better adaptation to the ac line frequency (for DMC applications, for example)

❏ PWM generation with the output units

❏ Real Time Clock (RTC) — if fed by the ACLK (32 kHz), the Timer_A can be used as an RTC with all low power modes. Time intervals of up to two seconds in steps of 2^{-15} s are possible.

6.3.1.1 Definitions Used with the Application Examples

```
; HARDWARE DEFINITIONS
;
TAIV      .equ    12Eh        ; Timer_A Vector Register
TACTL     .equ    160h        ; Timer_A Control Register
;                             ; Bits of the TACTL Register:
TAIFG     .equ    001h        ; Interrupt flag
TAIE      .equ    002h        ; Interrupt enable bit
CLR       .equ    004h        ; Reset TAR and Input Divider
MSTOP     .equ    000h        ; Stop Mode
MUP       .equ    010h        ; Up Mode
MCONT     .equ    020h        ; Continuous Mode
MUPD      .equ    030h        ; Up/Down Mode
D1        .equ    000h        ; Input Divider: Pass
D2        .equ    040h        ;        /2
D4        .equ    080h        ;        /4
D8        .equ    0C0h        ;        /8
ISTACLK   .equ    000h        ; Input Selector:TACLK
ISACLK    .equ    100h        ;        ACLK
ISMCLK    .equ    200h        ;        MCLK
ISINCLK   .equ    300h        ;        INCLK
;
CCTL0     .equ    162h        ; Capture/Compare Control Reg. 0
CCTL1     .equ    164h        ; Capture/Compare Control Reg. 1
CCTL2     .equ    166h        ; Capture/Compare Control Reg. 2
CCTL3     .equ    168h        ; Capture/Compare Control Reg. 3
CCTL4     .equ    16Ah        ; Capture/Compare Control Reg. 4
;                             ; Bits in the CCTLx Registers:
CCIFG     .equ    001h        ; Interrupt flag
COV       .equ    002h        ; Capture overflow flag
OUT       .equ    004h        ; Output bit
CCI       .equ    008h        ; Input signal
CCIE      .equ    010h        ; Interrupt enable bit
OMOO      .equ    000h        ; Output Mode:  output only
OMSET     .equ    020h        ;        set
OMTR      .equ    040h        ;        toggle/reset
```

```
OMSR      .equ     060h           ;           set/reset
OMT       .equ     080h           ;           toggle
OMR       .equ     0A0h           ;           reset
OMTS      .equ     0C0h           ;           toggle/set
OMRS      .equ     0E0h           ;           reset/set
CAP       .equ     100h           ; Capture/Compare switch
SCCI      .equ     400h           ; Synchronized CCI
SCS       .equ     800h           ; Async/sync switch
ISCCIA    .equ     000h           ; Capture input:  CCIxA
ISCCIB    .equ     1000h          ;           CCIxB
ISGND     .equ     2000h          ;           GND
ISVCC     .equ     3000h          ;           Vcc
CMDIS     .equ     000h           ; Capture mode:   disabled
CMPE      .equ     4000h          ;           rising edge
CMNE      .equ     8000h          ;           falling edge
CMBE      .equ     0c000h         ;           both edges
CCR0      .equ     172h           ; Capture/Compare Register 0
CCR1      .equ     174h           ; Capture/Compare Register 1
CCR2      .equ     176h           ; Capture/Compare Register 2
CCR3      .equ     178h           ; Capture/Compare Register 3
CCR4      .equ     17Ah           ; Capture/Compare Register 4
TAR       .equ     0170h          ; Timer Register
;
TA0       .equ     008h           ; Bit address TA0 Port3: P3.3
TA1       .equ     010h           ; Bit address TA1 Port3: P3.4
TA2       .equ     020h           ; Bit address TA2 Port3: P3.5
TA3       .equ     040h           ; Bit address TA3 Port3: P3.6
TA4       .equ     080h           ; Bit address TA4 Port3: P3.7
;
P3SEL     .equ     01Bh           ; Port3 Select Register
P3DIR     .equ     01Ah           ; Port3 Direction Register
P3OUT     .equ     019h           ; Port3 Direction Register
;
; Definitions of other used peripherals
;
SCFQCTL   .equ     052h           ; FLL Multiplier and Mod. Bit
```

```
M          .equ    080h        ; Modulation Bit
SCFI0      .equ    050h        ; Current Switches FN_x, FLL
FN_2       .equ    004h        ; DCO Switch for 2MHz
FN_3       .equ    008h        ; DCO Switch for 3MHz
SCFI1      .equ    051h        ; Taps of DCO
P0FG       .equ    013h        ; Port0 Flag Register Address
P0IE       .equ    015h        ; Port0 Interrupt Enable Reg.
IE1        .equ    0           ; Interrupt Enable Register
P0IE.0     .equ    4           ; P0.0 Interrupt Enable Bit
;
CBCTL      .equ    053h        ; Crystal Buffer Control
CBE        .equ    001h        ; Enable XBUF output
CBACLK     .equ    000h        ; ACLK is output at XBUF
CBMCLK     .equ    006h        ; MCLK is output at XBUF
;
BTCTL      .equ    040h        ; Basic Timer Control Register
BTCNT1     .equ    046h        ; Basic Timer Counter 1
WDTCTL     .equ    120h        ; Watchdog Control Register
CNCTL      .equ    008h        ; Reset Watchdog Bit
HOLD       .equ    080h        ; Stop Watchdog
;                              ; Bits in the Status Register SR
GIE        .equ    008h        ; General Interrupt Enable
CPUOFF     .equ    010h        ; CPU-Off bit
SCG0       .equ    040h        ; Low Power Mode Bits
SCG1       .equ    080h
```

6.3.2 Timer_A Hardware

Timer_A has a modular structure, giving it considerable flexibility. At least one capture/compare block is necessary for all configurations, and an almost unlimited number of capture/compare blocks may be connected to the timer register block (see Figure 6–4). The general function of these blocks is described below. The user software controls the Timer_A with the registers that are described there.

Several registers control the function of Timer_A. Every capture/compare register (CCRx) has its own control register CCTLx and the timer register (TAR) is also controlled by its own control register TACTL. This section describes all registers contained in the Timer_A.

The Timer_A registers have two common attributes:

❏ All registers, with the exception of the interrupt vector register (TAIV), can be read and written to

❏ All registers are word-structured and should be accessed therefore by word instructions only. Byte addressing results in a nonpredictable operation.

Example 6–5. Timer Register Low Byte

If only the information contained in the low byte of the timer register is wanted, then the following code sequence may be used:

```
MOV     &TAR,R5          ; Read the complete TAR: yyxxh
MOV.B   R5,R5            ; 00xxh to R5
```

If only the high byte information of the timer register is wanted:

```
MOV     &TAR,R5          ; Read the complete TAR: yyxxh
SWPB    R5               ; Swap bytes: yyxxh -> xxyyh
MOV.B   R5,R5            ; 00yyh to R5
```

Table 6–3 shows the mnemonics and the hardware addresses of the Timer_A registers.

Table 6–3. Timer_A Registers

REGISTER NAME	ABBREVIATION	REGISTER TYPE	ADDRESS	INITIAL STATE
Timer_A control register	TACTL	Read/Write	160h	POR Reset
Timer register	TAR	Read/Write	170h	POR Reset
Cap/Com Control Register 0	CCTL0	Read/Write	162h	POR Reset
Capture/Compare Register 0	CCR0	Read/Write	172h	POR Reset
Cap/Com Control Register 1	CCTL1	Read/Write	164h	POR Reset
Capture/Compare Register 1	CCR1	Read/Write	174h	POR Reset
Cap/Com Control Register 2	CCTL2	Read/Write	166h	POR Reset
Capture/Compare Register 2	CCR2	Read/Write	176h	POR Reset
Cap/Com Control Register 3	CCTL3	Read/Write	168h	POR Reset
Capture/Compare Register 3	CCR3	Read/Write	178h	POR Reset
Cap/Com Control Register 4	CCTL4	Read/Write	16Ah	POR Reset
Capture/Compare Register 4	CCR4	Read/Write	17Ah	POR Reset
Interrupt Vector Register	TAIV	Read only	12Eh	(POR Reset)

Note: Future extensions — more capture/compare registers — will use the reserved addresses 16Ch, 16Eh, 17Ch, and 17Eh.

6.3.2.1 The Timer Register Block

The timer register block is the main block of the Timer_A. Even the simplest version contains this block, which includes the timer register (TAR). The timer register block consists of the following parts:

☐ **Input Multiplexer** — selects the timer input signal out of four possible sources

☐ **Input Divider** — selects the division factor for the timer input signal (1, 2, 4, 8)

☐ **Timer Register TAR** — a 16-bit counter

☐ **Mode Control** — selects one of the possible four modes (Stop, Continuous, Up, Up/Down)

☐ **Timer Control Register TACTL** — contains all control bits for the timer register Block

☐ **Timer Vector Register TAIV** — contains the vector of the interrupt with the actual highest priority

☐ **Interrupt Logic**
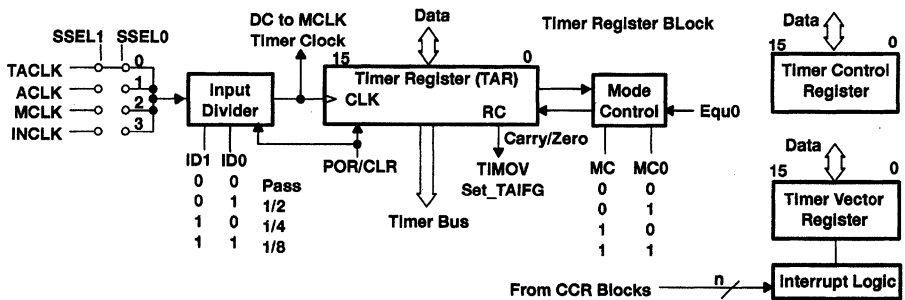
Figure 6–4. The Timer Register Block

6.3.2.1.1 The Timer Register (TAR)

The timer register (TAR) is the main register of the timer. The timer input frequency — selected from four different sources — is prescaled by the input divider (by a factor of 1, 2, 4, or 8) and counted with this 16-bit register. The timer register information is distributed to all other registers via the 16-bit tmer bus. This register contains the counted information in all three timer modes (Figure 6–4).

The timer register is incremented with the positive edge of its input signal, timer clock. The CCIFG flags and the TAIFG flag are also set with the positive edge if the programmed conditions are true.

The maximum resolution for the Timer_A is $1/f_{MCLK}$max. This relates to a maximum input frequency for the timer register equal to $f_{MCLKmax}$ (currently 4 MHz, 250 ns resolution for the MSP430C33x).

The 16 bits of the timer register can be cleared by two methods:

```
CLR    &TAR           ; 0 -> TAR, nothing else
BIS    #CLR,&TACTL    ; Clear TAR, Inp. Div. + count dir
```

The second method clears not only the timer register, but also the content of the input divider and sets the count direction of the timer register to upward.

6.3.2.1.2 The Timer_A Control Register TACTL

The timer control register (TACTL) contains all bits that control the timer register (TAR) and its operation. The control bits are reset with the power-on reset (POR) signal but the power-up clear (PUC) signal does not affect them. This allows a continued Timer_A operation if the watchdog times out or the watchdog security key is violated. The timer control register (Figure 6–5) is a word register and should therefore be accessed with word instructions only.

TACTL	15							0
	unused	Input Select	Input Divider	Mode Control	un–used	CLR	TAIE	TAIFG
160h	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	(w)–(0)	rw–(0)	rw–(0)
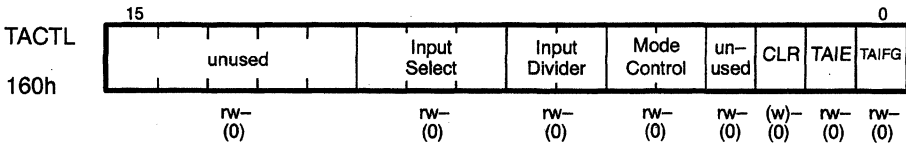
Figure 6–5. Timer Control Register (TACTL)

If the operation of the Timer_A needs to be modified — with the exception of the TAIFG and TAIE bits — then the Timer_A should be halted during the modification of the control bits. After the change of the TACTL register, Timer_A is restarted. Without this procedure, unpredictable behavior is possible.

Example 6-6. The Timer_A Control Register TACTL

The timer should be restarted in continuous mode. This is accomplished with two instructions. The first instruction defines the new state of the timer (except the mode), and stops it (Mode Control = 00). The second instruction sets the mode control bits to continuous mode and restarts the timer operation.

Input Selection: MCLK
Input Divider: /4, cleared
Interrupt: enabled, TAIE = 1, TAIFG = 0

```
MOV     #ISMCLK+D4+CLR+TAIE,&TACTL  ; Define new state

BIS     #MCONT,&TACTL       ; Restart Continuous Mode
```

The control bits of the Timer control register are explained below.

Timer Interrupt Flag   TAIFG
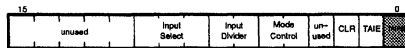
This flag indicates a timer overflow event: the timer register TAR reached the value zero. The way to get the flag TAIFG set depends on the mode used:

❏   **Continuous Mode** — TAIFG is set if the timer counts from 0FFFFh to 0000h.

❏   **Up Mode** — TAIFG is set if the timer counts from the CCR0 value to 0000h.

❏   **Up/Down Mode** — TAIFG is set if the timer counts down to 0000h.

See the *The Timer Vector Register TAIV* section for examples how to use the TAIFG flag.

Timer Overflow Interrupt Enable Bit   TAIE
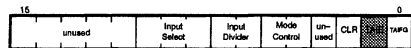
This bit enables and disables the interrupt for the timer interrupt flag TAIFG:

TAIE = 0: Interrupt is disabled

TAIE = 1: Interrupt is enabled

An interrupt is requested only if the TAIFG bit, the TAIE bit, and the GIE (SR.3) bit are set. The sequence of the bit setting does not matter. If two out of the three bits (mentioned above) are 1, and the third is set afterward, an interrupt will be requested.

Example 6-7. Timer Overflow Interrupt Enable Bit TAIE

Interrupt is enabled for the TAIFG flag. A pending interrupt is cleared.

```
BIC     #TAIFG,&TACTL     ; Clear TAIFG flag
BIS     #TAIE,&TACTL      ; Enable interrupt for TAIFG
```
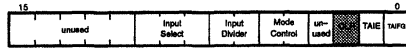
Timer Clear Bit CLR

The timer register (TAR) and the input divider are cleared, after POR or if bit CLR is set by the software. The CLR bit is automatically reset by the hardware and always read as 0. The Timer_A starts operation with the next positive edge of the timer clock. The counting starts in upward direction if it is not halted by cleared Mode Control bits.

Example 6-8. Timer Clear Bit CLR

Timer_A is restarted after the calibration process. It needs a complete reset: up/down mode, upward count direction, interrupt enabled, MCLK passed to the timer register, input divider cleared.

```
MOV     #ISMCLK+D1+CLR+TAIE,&TACTL ; Define state
BIS     #MUPD,&TACTL               ; Start Up/Down Mode
```

Bit 3

Not used. Read as 0. To maintain software compatibility, this bit should **NOT** be set.
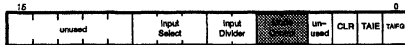
Mode Control Bits

The two mode control bits define the operation of the Timer_A. Table 6-4 lists the four possible modes. See Section 6.3.3 *The Timer Modes* for a detailed description of the timer modes. If the mode control bits are cleared (stop mode), a restart of the timer operation is possible exactly at the point where the operation was halted, including the count direction information used with the up/down mode.

Table 6-4. Mode Control Bits

MODE CONTROL BITS	COUNT MODE	COMMENT
0	Stop Mode	Timer is halted
1	Up Mode	Count up to CCR0 and restart at 0
2	Continuous Mode	Count up to 0FFFFh and restart at 0
3	Up/Down Mode	Count up to CCR0 and back to 0, restart

Example 6–9. Mode Control Bits

Timer_A is stopped and restarted in continuous mode.

```
BIC     #MUP+MCONT,&TACTL ; Stop Timer_A

...

BIS     #MCONT,&TACTL     ; Restart in Cont. Mode
```

Input Divider Control Bits

The two input divider control bits allow the use of a prescaled input frequency (timer clock) for the timer register (TAR). A prescaler may be necessary because of any of the following:

☐ The MCLK frequency (up to 4 MHz) is too high for the task.

☐ The MCLK frequency leads to an overflow of the timer register (TAR) during the necessary measurement periods. This makes a RAM extension of the TAR necessary, which takes time and occupies RAM space.

☐ The resulting resolution is not necessary.

☐ The resulting timer register contents lead to numbers that are too large during the calculations.

☐ Power savings is important.

If one the above reasons is true, then a prescaled input frequency should be used. The possible prescale factors are shown in Table 6–5.

Table 6–5. Input Divider Control Bits

INPUT DIVIDER BITS	MODE	COMMENT
0	Pass	Input signal is passed to the Timer Register
1	÷ 2	Input signal is divided by 2
2	÷ 4	Input signal is divided by 4
3	÷ 8	Input signal is divided by 8

Example 6–10. Input Divider Control Bits

The input divider is changed from pass mode (0) to divide-by-4 mode (2):

```
BIC     #MUP+MCONT,&TACTL ; Stop Timer_A
BIS     #MUP+D4+CLR,&TACTL ; Continue in Up Mode
```

Input Selection Bits

15										0
	unused				Input Divider	Mode Control	un- used	CLR	TAIE	TAIFG

The three input selection bits select the input signal of the input divider. Four different sources are provided as shown in Table 6–6. The INCLK input may be used for a fourth input source with other family members.

Table 6–6. Input Selection Bits (MSP430x33x — Source Depends on MSP430 Type)

INPUT SELECT BITS	SIGNAL	COMMENT
0	TACLK	Signal at the external pin TACLK is used
1	ACLK	ACLK is used
2	MCLK	MCLK is used
3	INCLK	MCLK for the MSP430C33x
4 – 7	N/A	Reserved for future expansion

The highest timer resolution is possible with the internal MCLK signal: the full range of the MCLK frequency may be used. If the external pin TACLK (P3.2 for the MSP430C33x) is selected, then the maximum input frequency is restricted due to the internal capacities of the signal path. See the specification for actual limits.

Example 6–11. Input Selection Bits

Timer_A is initialized. Continuous mode, interrupt enabled, ACLK — divided by 2 — routed to the timer register, timer register and input divider are cleared.

```
        MOV     #ISACLK+D2+CLR+TAIE,&TACTL   ; Define state
        BIS     #MCONT,&TACTL                ; Start timer: Cont.
Mode
```

Bit 11 to 15

Not used. Read as 0. To maintain software compatibility, these bits should *NOT* be set to 1.

6.3.2.1.3 The Timer Vector Register TAIV

This 16-bit register contains an even vector ranging from 0 (no interrupt pending) via 2 (CCR1 interrupt) to 10 (timer overflow interrupt TIMOV). See Figure 6–6 and Table 6–7 for more information.
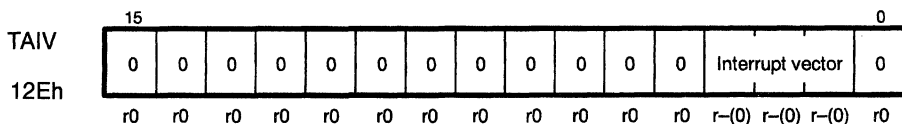
Figure 6–6. Timer Vector Register (TAIV)

If more than one interrupt is pending, then the vector with the highest priority is placed into the TAIV register. See figure 6–7. Table 6–7 illustrates the interrupt priority scheme of Timer_A:

Table 6–7. Timer Vector Register Contents

INTERRUPT PRIORITY	INTERRUPT SOURCE	FLAG	VECTOR ADDRESS	VECTOR REGISTER CONTENTS
Highest	Capture/Compare 0	CCIFG0	0FFF2h	N/A
	Capture/Compare 1	CCIFG1	0FFF0h	2
	Capture/Compare 2	CCIFG2	0FFF0h	4
	Capture/Compare 3	CCIFG3	0FFF0h	6
	Capture/Compare 4	CCIFG4	0FFF0h	8
	Timer Overflow	TAIFG	0FFF0h	10
Lowest	Reserved		N/A	12
	No interrupt pending		N/A	0

The timer vector register allows a very fast response to the different timer interrupts. Its content is simply added to the program counter (PC), using a JMP table located directly after the ADD instruction:

```
        ADD     &TAIV,PC        ; INTRPT with highest priority
        RETI                    ; 0: No INTRPT pending
        JMP     HCCR1           ; 2: CCIFG1 caused INTRPT
        JMP     HCCR2           ; 4: CCIFG2 caused INTRPT
        JMP     HCCR3           ; 6: CCIFG3 caused INTRPT
        JMP     HCCR4           ; 8: CCIFG4 caused INTRPT
HTIMOV  ...                     ; 10: TAIFG is reason
```

If the corresponding interrupt handlers are out of the reach of JMPs (more than ±511 words), then a word table containing the handler start addresses may be used:

```
        MOV     &TAIV,R5        ; TAIV contains vector: 0 - 10
        MOV     TTAB(R5),PC     ; Write handler address to PC
TTAB    .WORD   PRETI           ; 0: No INTRPT pending, RETI
        .WORD   HCCR1           ; 2: CCIFG1 caused INTRPT
        .WORD   .HCCR2          ; 4: CCIFG2 caused INTRPT
        .WORD   HCCR3           ; 6: CCIFG3 caused INTRPT
        .WORD   HCCR4           ; 8: CCIFG4 caused INTRPT
        .WORD   HTIMOV          ; 10: TAIFG is reason
```

A third (slower) method is to read the content of the register TAIV and to use the read value for the decision of where to proceed (the interrupt flag with the highest priority is reset by the MOV instruction):

```
        MOV     &TAIV,R5        ; Actual vector to R5. Reset flag
        CMP     #2,R5           ; Check for CCIFG1 interrupt
        JEQ     HCCR1           ; 2: CCIFG1 caused INTRPT
        CMP     #4,R5           ; Check for CCIFG2 interrupt
        JEQ     HCCR2           ; 4: CCIFG2 caused INTRPT
        ...                     ; a.s.o.
```

The next software example shows a method that does not use the register TAIV. A normal *skip chain* is used. Only the software for blocks 0 and 1 is shown (this example makes the advantages of using the TAIV register obvious):

```
        BIT     #CCIFG0,&CCTL0  ; Block 0: Flag set?
        JNZ     MOD0            ; Yes, serve it
        BIT     #CCIFG1,&CCTL1  ; Block 1: Flag set?
        JNZ     MOD1            ; Yes, serve it
        ...                     ; Continue with skip chain
MOD0    BIC     #CCIFG0,&CCTL0  ; Reset CCIFG0 flag
        ...                     ; Start handler for block 0
MOD1    BIC     #CCIFG1,&CCTL1  ; Reset CCIFG1 flag
        ...                     ; Start handler for block 1
```

The capture/compare block 0 is not included in the TAIV register; it has its own interrupt vector located at address 0FFF2h. The shorter interrupt latency time of register CCR0, makes it the preferred choice for the most time critical applications. The vector for the other Timer_A interrupts is located at address 0FFF0h.

Note:

The timer vector register contains only the vectors of timer blocks with enabled interrupt (set CCIEx resp. TAIE bits). Blocks with disabled interrupt bits (reset CCIEx resp. TAIE bits) can be checked by software if their CCIFG resp. TAIFG flag is set and the flag must be reset by software too. See the *skip chain* example above.

No interrupt flag (CCIFGx or TAIFG) needs to be reset if the register TAIV is used. The act of reading of the timer vector register TAIV resets the interrupt flag automatically that determines the actual register content. The interrupt flag with the next lower priority level defines the timer vector register TAIV afterward.

Note:

Any access to the timer vector register (read or write) resets the interrupt flag with the highest priority. The timer vector register should be read only and the read data should be used to determine the interrupt handler with the highest priority, otherwise the data is lost.

Figure 6–7 shows the internal interrupt logic that controls the register TAIV. The five controlling inputs are shown.
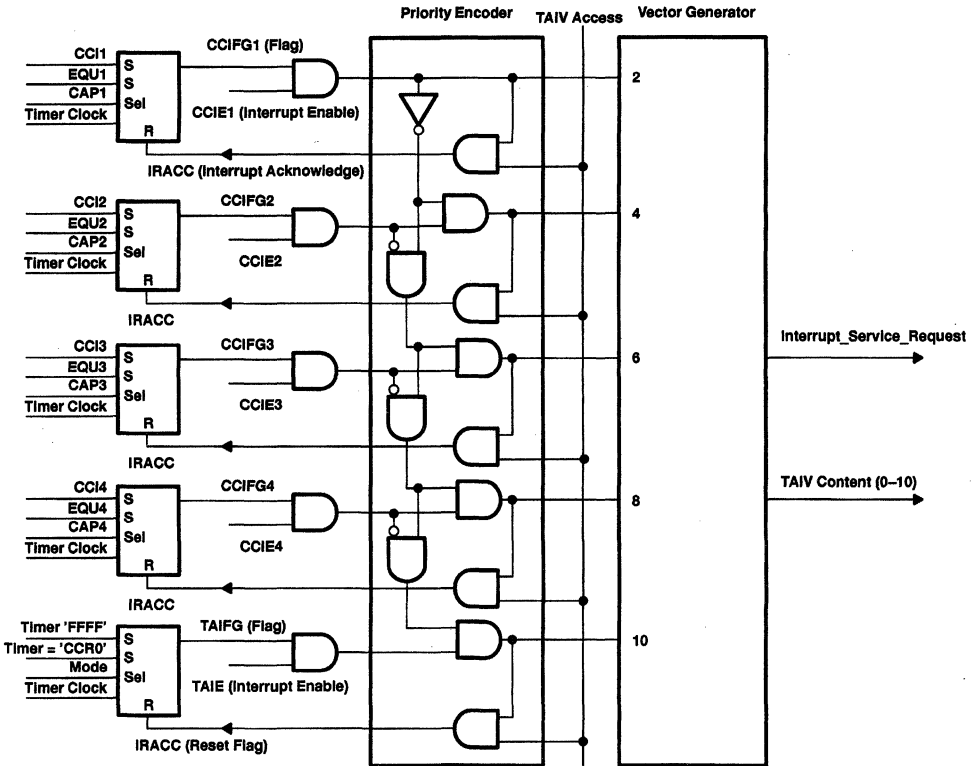
Figure 6–7. Simplified Logic of the Timer Interrupt Vector Register

6.3.2.2 The Capture/Compare Register Blocks

Figure 6–8 illustrates the capture/compare register block 1. The others, with the exception of the capture/compare register block 0, are identical The CCR0 block has additional functions. See section *The Period Register CCR0,* below.
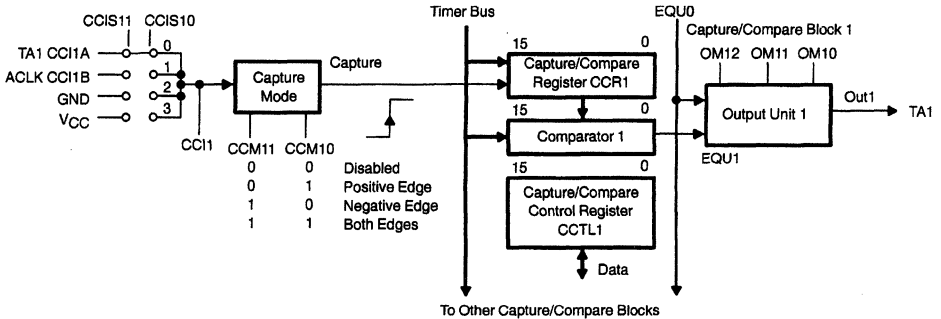
Figure 6–8. Capture/Compare BLock 1

6.3.2.2.1 The Capture/Compare Registers CCRx

These registers may be used individually as compare registers or as capture registers. Any combination is possible.
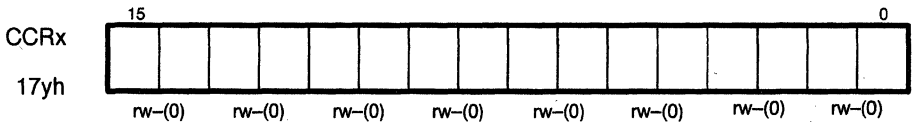
Figure 6–9. The Capture/Compare Registers CCRx

☐ **Compare Mode With Continuous Mode** — the register CCRx contains the time information for the next interrupt. Within the interrupt handler, the time information for the next interrupt is prepared. The number Δn (corresponding to the time interval Δt from the last interrupt to the next one) is added to CCRx. The interrupt latency time does not play a role in this method. See the example in section *The Continuous Mode.*

The output units may be used to generate output changes at output pins TAx with an exactly defined timing, independent of interrupt latency times.

☐ **Compare Mode With Up Mode or Up/Down Mode** — the capture/compare register 0 is used as the period register with these two modes.

The register CCRx contains the time interval between interrupts respective the pulse width of the output signal at TAx. The registers CCRx are modified depending on the result of the control calculations. If no pulse width change is necessary, the timing is repeated without CPU intervention.

❏ **Capture Mode With Continuous Mode** — a register (CCRx) used with the capture mode, copies the timer register at the precise moment the selected capture conditions are satisfied. This allows very accurate measurements of timings independent of the interrupt latency time. If the time intervals to be captured are longer than 65536 timer register steps, then a RAM extension (TIMAEXT) is necessary. This RAM extension is incremented with the TAIFG interrupt and used with the calculations as shown below:

$$n_{capt} = 65536 \times n_{ext} + n_{TAR}$$

This means: with the continuous mode the RAM extension contains simply the *extended* timer bits 17 through 31. No correction or calculation is necessary.

❏ **Capture Mode With Up Mode** — this method of capturing is exactly the same as described above for the continuous mode. But the up mode uses only a part of the timer register range. If the time interval to be captured is longer than the content of the period register (CCR0), then a RAM extension (TIMAEXT) is necessary. This RAM extension is incremented with the CCIFG0 or TAIFG interrupt and used with the calculations as shown below:

$$n_{capt} = n_{ext} \times (n_{CCR0} + 1) + n_{TAR}$$

❏ **Capture Mode With Up/Down Mode** — this method of capturing is exactly the same as described above for the continuous mode. But the up/down mode uses only a part of the timer register range and this part is counted up and down. Therefore, the actual count direction should also be considered. If the time interval to be captured is longer than the doubled content of the period register (CCR0), then a RAM extension (TIMAEXT) is necessary. This RAM extension is incremented with the CCIFG0 interrupt and with the TAIFG interrupt. The LSB of the RAM extension (TIMAEXT) indicates the count direction. The RAM extension TIM32 must be initialized to zero.

LSB of TIMAEXT = 0 — Timer register counts upwards

$$n_{capt} = n_{ext} \times n_{CCR0} + n_{TAR}$$

LSB of TIMAEXT = 1: Timer register counts downwards

$$n_{capt} = n_{ext} \times n_{CCR0} + \left(n_{CCR0} - n_{TAR}\right)$$

Where:

n_{capt} Resulting cycle value for captured signals (> 16 bits)
n_{ext} Content of the timer register RAM extension TIMAEXT
n_{CCR0} Content of the period register CCR0
n_{TAR} Captured content of the timer register TAR (captured in CCRx)

Figure 6–10 illustrates the logic used for the capture/compare registers.
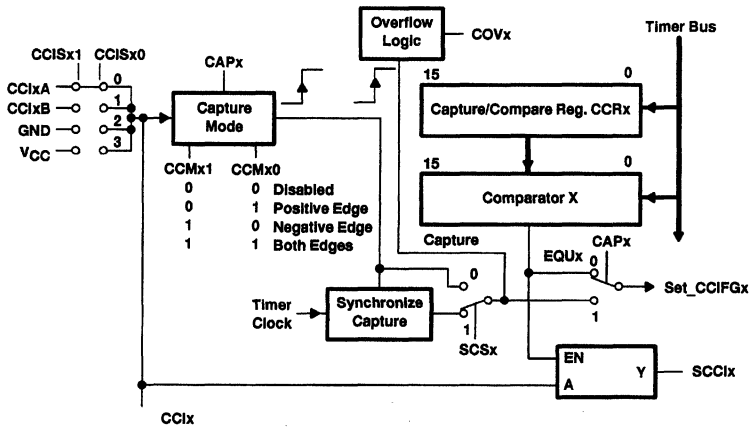
Figure 6–10. Function of the Capture/Compare Registers (CCRx)

6.3.2.2 The Capture/Compare Control Registers CCTLx

Each capture/compare block has its own control word CCTLx. Figure 6–11 illustrates the organization of these control words — it is the same for all of them. The main bit of these registers is the CAP bit (CCTLx.8), which determines if the capture/compare block works in the capture mode or in the compare mode.

CCTLx 162h to 16Ah	15											0
	CAPTURE MODE	INPUT SELECT	SCS	SCCI	un– used	CAP	OUTMODx	CCIE	CCI	OUT	COV	CCIFG
	rw– (0)	rw– (0)	rw– (0)	rw– (0)	rw– (0)	rw– (0)	rw– (0)	rw– (0)	r	rw– (0)	rw– (0)	rw– (0)
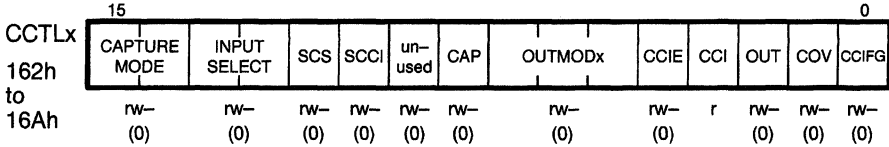
Figure 6–11. Timer Control Registers (CCTLx)

The POR signal resets all bits of the registers (CCTLx), but the PUC signal does not affect them. This permits continuation with the same timing after a watchdog reset, if this is necessary.

Capture/Compare Interrupt Flag   CCIFG
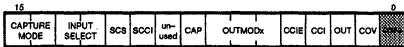
This flag indicates two different events depending on the mode in use:

■ Capture Mode

If set, it indicates that a timer register value was captured in the corresponding capture/compare register (CCRx).

■ Compare Mode

If set, it indicates that the timer register value was equal to the data contained in the corresponding capture/compare register (CCRx). The signal EQUx is also generated.

The CCIFG0 flag is reset automatically when the interrupt request is accepted. It is a single-source interrupt flag and its interrupt vector is located at address 0FFF2h.

The reset of the CCIFG1 to CCIFGx flags depends on:

■ The timer vector register TAIV is used

The flag that determines the actual vector word (content of TAIV) is reset automatically after the register TAIV is read.

■ The timer vector register TAIV is not used

The flags CCIFG1 to CCIFGx must be reset by the interrupt handler

If the interrupt capability is not enabled for a capture/compare block then the flag CCIFGx must be tested to check if the block x needs service. The CCIFG flag must be reset by software for this case:

```
BIT     #CCIFG,&CCTLx    ; Flag set?
JZ      NO_FLAG          ; No continue
BIC     #CCIFG,&CCTLx    ; Yes, reset flag
...                      ; Execute task for block x
```

Example 6–12. Capture/Compare Interrupt Flag CCIFG

See the examples in section *The Timer Vector Register TAIV.* Examples for the treatment of the CCIFG flags are given there.

Capture Overflow Flag COV
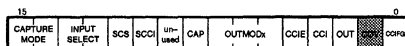
This flag indicates two different events depending on the mode in use:

- Compare Mode

  No function. The COV bit is always reset, independent of the state of the capture input.

- Capture Mode

  The capture overflow flag COV is set if a second capture event occurred before the first capture sample was read out of the capture register (CCRx). The COV flag allows the software to detect the loss of synchronization and helps to reacquire synchronization. The COV flag is not reset by the reading of the CCRx register and must be reset by software.

Example 6–13. Capture Overflow Flag COV

The interrupt handler of capture/compare block 2 — running in capture mode — checks first to see if a capture overflow occurred.

```
HCCR2   BIT     #COV,&CCTL2      ; Capture overflow ?
        JNZ     COV2             ; Yes, handle it
        MOV     &CCR2,CAPSTO2    ; Store valid captured value
        ...                      ; Proceed with task
;
; Error handler for Capture/Compare Block 2
```

```
;
COV2    BIC     #COV,&CCTL2         ; Reset overflow flag COV
        ...                         ; Check reason for overflow
        RETI
```

Output Bit   OUT
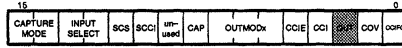
The state of the output bit OUT defines the output signal (TAx) of output unit x if the output mode 0 (output only) is selected. See section *The Output Units* for details. The state of the output signal (TAx) is always indicated by this bit, independent of the output mode in use. A modification of the output signal is possible only if the output mode 0 is selected. The OUT bit allows the definition of the start condition for PWM.

Example 6–14. Output Bit   OUT

The output unit 3 is not used currently by Timer_A. To place TA3 in a defined state, output mode 0 is used and output TA3 is reset.

```
BIC     #0E0h+OUT,&CCTL3 ; Output only to OUT3: 0
```

If output TA3 should be set initially the following sequence is used:

```
BIC     #0E0h,&CCTL3        ; Output only to OUT3
BIS     #OUT,&CCTL3         ; Set OUT3
```

Capture/Compare Input Bit   CCI
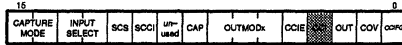
The CCI bit allows to read the state of the selected capture input: the input signal (CCIxA at pin TAx, ACLK, Vcc or Vss) can be read independent of the selected mode. See figure 6–10 for details.

Example 6–15. Capture/Compare Input Bit   CCI

The timer block 4 — running in capture mode — uses different software parts for the leading and the trailing edges of the input signal. The interrupt handler checks via the CCI4 bit which edge is the actual one.

```
; Initialization part: Capture both edges, TA4 input,
; synchronized capture, Capture Mode, interrupt enabled
;
```

```
            MOV      #CMBE+ISCCIA+SCS+CAP+CCIE,&CCTL4  ; Initialize
            . . .
HCCR4       .equ     $                       ; Interrupt handler Block 4
            BIT      #CCI,&CCTL4             ; Input signal positive?
            JNZ      TA4POS                  ; Yes: leading edge occurred
            . . .                            ; No, handle trailing edge
            RETI
TA4POS      . . .                            ; Handle leading edge
            RETI
```

Capture/Compare Interrupt Enable Bit CCIE
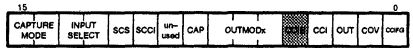
This bit enables and disables the interrupt for the capture/compare interrupt flag CCIFGx:

CCIE = 0: Interrupt is disabled
CCIE = 1: Interrupt is enabled

Interrupt is requested only if the CCIFG bit, the corresponding CCIE bit, and the GIE bit (SR.3) are set. The sequence of the bit setting does not matter. If two out of the above-mentioned three bits are 1 and the third is set afterward, an interrupt will be requested.

Example 6–16. Capture/Compare Interrupt Enable Bit CCIE

The interrupt of timer block 2 is disabled. Now the interrupt should be enabled again. But if the CCIFG2 flag is set, no interrupt should occur. All other bits in register CCTL2 should retain their states.

```
            BIC      #CCIE,&CCTL2    ; Disable interrupt Block 2
            . . .                    ; Continue
;
; The interrupt for Timer Block 2 is enabled again.
; A pending interrupt is cleared
;
            BIC      #CCIFG,&CCTL2   ; Reset CCIFG2 flag
            BIS      #CCIE,&CCTL2    ; Enable interrupt Block 2
```

Output Mode Bits
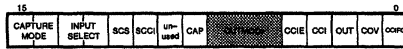
These three bits define the behavior of the output unit x. Table 6–8 illustrates the influence of the signals EQUx and EQU0 to the output signal TAx. The table shows the actions when timer register TAR is equal to CCRx or CCR0. Table 6–8 is valid for all timer modes.

Table 6–8. Output Modes of the Output Units

OUTPUT MODE	NAME	TAR COUNTED UP TO CCRx	TAR COUNTED UP TO CCR0
0	Output only	TAx is set according to bit OUTx (CCTLx.2)	
1	Set	Sets output	No action
2	Toggle/Reset	Toggles output	Resets output
3	Set/Reset	Sets Output	Resets output
4	Toggle	Toggles output	No action
5	Reset	Resets output	No action
6	Toggle/Set	Toggles output	Sets output
7	Reset/Set	Resets output	Sets output

See the examples given in the section *The Output Units*.

Capture/Compare Select Bit CAP

The CAP bit defines if the capture/compare block works in the capture mode or in the compare mode. This bit influences the function of nearly all other control bits located in the same capture/compare control register. See figure 6–10 for an explanation of the used logic.

CAP = 0: The compare mode is selected
CAP = 1: The capture mode is selected

Example 6–17. Capture/Compare Select Bit CAP

The use of this bit is explained with all other control bits.

Bit 9

Not used. Read as 0. To maintain software compatibility this bit should **NOT** be set to 1.

Synchronized Capture/Compare Input SCCI

■ Compare Mode

The SCCI bit is the output of a transparent latch. This latch is in transparent mode as long as the timer register TAR is equal to CCRx. The SCCI bit stores the selected capture input (ACLK, Vcc, or Vss) when the timer register TAR becomes unequal to register CCRx.

■ Capture Mode

The state of this bit is not defined. No EQUx signal is available in capture mode.

Example 6–18. Synchronized Capture/Compare Input SCCI

The timer bock 4 — running in capture mode — uses different software parts for the two possible states of the ACLK signal when the EQU4 signal comes true. The interrupt handler checks via the SCCI4 bit the state of the ACLK signal when CCR4 was equal to the timer register (TAR). The read information is shifted into a RAM word DATA.

```
; Initialization part: ACLK, Compare Mode, interrupt enabled
; Output Unit disabled, clear CCIFG
;
          MOV       #ISCCIB+OMOO+CCIE,&CCTL4 ; Init. Timer_A
          ...
HCCR4     MOV       &CCR4,DATA        ; Interrupt handler Block 4
          BIT       #SCCI,&CCTL4      ; ACLK signal -> Carry
          JNZ       TA4POS            ; ACLK was high during EQU4
          RRC       DATA              ; Shift captured info in DATA
          ...                         ; Execute task for low input
          RETI
;
TA4POS    RRC       DATA              ; Shift captured info in DATA
          ...                         ; Execute task for high input
          RETI
```

Synchronization of Capture Signal Bit SCS
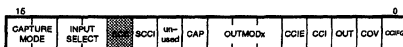
The capture signal can be read in asynchronous mode or synchronized with the selected timer clock . The SCS bit selects the mode to be used. See also Figure 6–10 for a depiction of the internal logic.

■ SCS = 0

The asynchronous capture mode sets the CCIFG flag immediately when the capture conditions are met (rising edge, falling edge, both edges) and also immediately captures the timer register. This mode may be used if the period of the captured data is much longer than the period of the selected timer clock. The captured data may be incorrect for high input frequencies at terminal TAx.

■ SCS = 1

The synchronous capture mode — which is used normally — synchronizes the setting of the CCIFG flag and the capturing of the selected capture input with the selected timer clock. The captured data is always valid.

Example 6–19. Synchronization of Capture Signal Bit SCS

See Example for *Capture Compare Input bit CCI.*

Capture/Compare Input Selection Bits

15												0
CAPTURE MODE		SCS	SCCI	un- used	CAP	OUTMODx	CCIE	CCI	OUT	COV	CCIFG
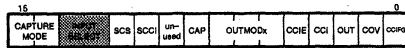
These two bits select the input signal to be captured. The operation for the captured signal is different for capture mode and compare mode:

■ Compare Mode

The selected input signal is read and stored with the EQUx signal. See the description of the SCCI bit, above.

■ Capture Mode

The selected input signal captures the timer register TAR into the capture/compare register CCRx when the conditions defined in the capture mode bits are met. See the description of the capture mode bits below.

Table 6–9. Capture/Compare Input Selection Bits (MSP430x33x)

INPUT SELECTION BITS	INPUT SIGNAL	COMMENT
0	CCIxA	Signal at the external pin TAx is selected
1	CCIxB	ACLK is selected
2	V_{SS}	For software capturing
3	V_{CC}	For software capturing

Capture Mode Selection Bits

15												0
	INPUT SELECT	SCS	SCCI	un- used	CAP	OUTMODx	CCIE	CCI	OUT	COV	CCIFG
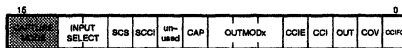
These two bits select the capture operation for the input signal to be captured:

■ Compare Mode

No function.

■ Capture Mode

The content of the timer register TAR is stored in the capture/
compare register CCRx when the capture condition is true for the se-
lected input signal. The capture conditions are listed in Table 6–10.

Table 6–10. Capture Mode Selection Bits

CAPTURE MODE BITS	COMMENT
0	Capture mode is disabled
1	Capturing is done with the rising edge (0 to 1)
2	Capturing is done with the falling edge (1 to 0)
3	Capturing is done for both edges

Example 6–20. Capture Mode Selection

The capture/compare block 3 — running in capture mode — measures the pe-
riod of the input signal CCI3A at terminal TA3. The measurement is made con-
tinuously between two rising edges. The calculated period is stored in the RAM
location PERIOD for the use by the background software. The actual value of
CCR3 is stored in OLDVAL for the next calculation. Timer_A uses the continu-
ous mode.

```
; Initialization part: Capture rising edge, TA3 input,
; synchronous capture, Capture Mode, interrupt enabled
;
PERIOD   .equ    0200h           ; Calculated period
OLDVAL   .equ    0202h           ; Storage of last pos. edge time
;
         MOV     #CMPE+ISCCIA+CAP+CCIE+SCS,&CCTL3    ; Init.
         ...
HCCR3    .equ    $               ; Interrupt handler Block 3
         PUSH    &CCR3           ; Captured TAR, rising edge
         MOV     @SP,PERIOD
         SUB     OLDVAL,PERIOD   ; New - old = period
         POP     OLDVAL          ; For next calculation
         RETI
```

6.3.2.2.3 The Period Register CCR0

The purpose of register CCR0 changes with the used timer mode.

❑ **Continuous Mode** — if this mode is used, CCR0 is a capture/compare register exactly like the other four registers (CCR1 to CCR4). See section *The Timer_A Modes* for details.

❑ **Up Mode or Up/Down Mode** — with one of these modes selected, the register CCR0 works as the period register for the Timer_A, which defines the length of the timer period. Whenever the timer register (TAR) reaches the value of CCR0 (EQU0 = 1), the following actions occur, depending on the mode in use:

■ Up Mode

The timer register is cleared with the next timer clock and restarts from the value 0. This continues automatically without any software intervention necessary. See section *The Timer_A Modes*.

■ Up/Down Mode

The timer register changes the count direction and starts to count down to 0 with the next timer clock. If 0 is reached, the timer register counts up again with the next timer clock until the value of CCR0 is reached again. This continues automatically without any further software intervention necessary. See section *The Timer_A Modes*.

With the up mode or up/down mode selected, the EQU0 signal is valid if the timer register (TAR) equals the period register (CCR0), or if it is greater than CCR0. This is not the case for the other registers (CCRx).

The value 0 is not a valid content for the period register: the Timer_A blocks.

The content of the period register CCR0 is not modified normally. The timer period is a constant value (50 μs for a repetition rate of 20 kHz — this means 200 cycles for a 4 MHz MCLK). But this value may also be modified if necessary.

6.3.3 Timer Modes

Timer_A provides three different operating modes as well as the stop mode:

❏ **Continuous Mode** — the normal mode, except when high-speed PWM generation is necessary

❏ **Up Mode** — used for high-speed, asymmetric PWM generation

❏ **Up/Down Mode** — used for high-speed, symmetric PWM generation

❏ **Stop Mode** — Timer_A is halted, all control bits retain their status

One of the advantages of Timer_A is the absolute synchrony of all timings and output signals. This is due to the single timer register (TAR) that controls all timings. This synchrony is very important for the interdependence of timings, for example, if the MSP430 is used with a 3-phase digital motor control application (DMC).

The equations shown in the next sections use the following abbreviations:

Δt	Time interval between two similar interrupts	[s]
t	Time e.g. period of a PWM signal	[s]
t_{pw}	Pulse width of a PWM signal	[s]
Δn	Cycle value added to a CCRx register (timer clock cycles)	
n	Number — content of a register (CCRx)	
k	Predivider constant of the input divider (1, 2, 4 or 8)	
f_{CLK}	Input frequency at the input divider input of Timer_A	[Hz]
n_{CCR0}	Content of the period register CCR0	

The calculation formulas and explanations for the capture mode are given in the section *Capture/Compare Blocks*.

6.3.3.1 The Continuous Mode

This mode allows up to five completely independent, synchronous timings. The capture/compare register, CCR0, works exactly the same as the other registers (CCRx) when running in continuous mode.

Note:

The signal EQU0 has the same influence on the Mode Control Logic as it does in the other timer modes. This means that only the *Set, Reset*, and *Toggle* modes should be used if independent output signals are desired.

Figure 6–12 shows two independent timings generated by capture/compare registers CCR0 and CCR1. The content of the capture/compare registers

(CCRx) is updated by software during each interrupt sequence by the addition of a calculated value, Δn. The value Δn represents a time interval, Δt, expressed in timer clock cycles. The software is described below. See the example for details.

The formulas for a given time interval, Δt, respective the corresponding cycle value Δn are:

$$\Delta t \;=\; \frac{\Delta n \times k}{f_{CLK}} \;\rightarrow\; \Delta n \;=\; \frac{\Delta t \times f_{CLK}}{k}$$

The limitation for Δn is:

$$\Delta n \;<\; 2^{16}$$

If this limitation is not given, a RAM extension for the timer register and the capture/compare registers must be used.

The number of timer steps between two equal timer register contents is 65536 (10000h). If the time interval Δn is smaller than 65536, no checks for overflow are necessary between two interrupts. The calculated next register content is always correct.
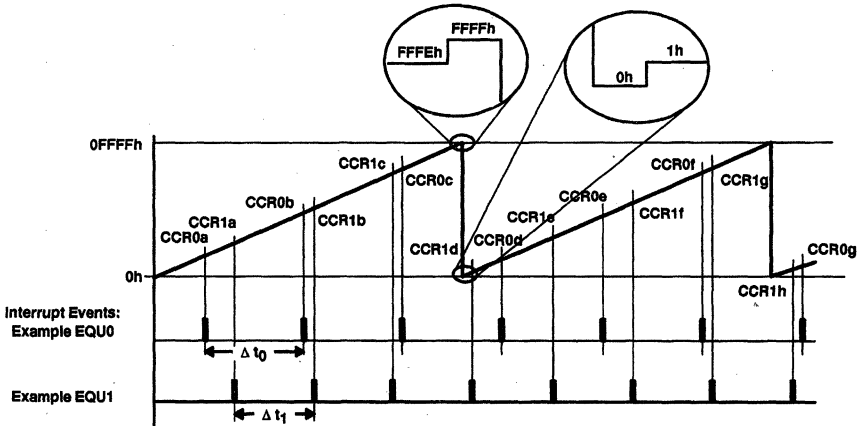
Figure 6–12. Two Different Timings Generated With the Continuous Mode

Example 6–21. Continuous Mode

The software for the example illustrated in figure 6–12 is shown below. The system clock frequency is 1.048 MHz. capture/compare Block 0 — running in compare Mode — uses a constant interrupt repetition rate of 20971 cycles (equivalent to $\Delta t0 = 20.0$ ms @ 1.048 MHz), capture/compare Block 1 — running in compare mode — uses 17476 cycles (equivalent to $\Delta t1 = 16.67$ ms). These cycle values are added to the corresponding capture/compare registers, CCR0 and CCR1, respectively. They define the time for the next interrupt (previous cycle count + number of cycles Δn).

The capture/compare block 2 runs in capture mode. It checks if the time interval between two positive input edges is shorter than a given value stored in MIN. If this is the case, the error byte ERR is set to 1.

```
; Initialization of the Timer_A: Cont. Mode, /1, interrupt
; enabled, MCLK = 1.048MHz. Output Units 0 and 1 not used.
;
INIT     MOV      #ISMCLK+CLR+TAIE,&TACTL ; Prepare Timer_A
         MOV      #OMOO+CCIE,&CCTL0 ; CCR0: timing only
         MOV      #OMOO+CCIE,&CCTL1 ; CCR1: timing only
         MOV      #CMPE+SCS+CAP+CCIE,&CCTL2 ; CCR2: capt. TA2
         MOV      #20971,&CCR0     ; delta t0 = 20ms
         MOV      #17476,&CCR1     ; delta t1 = 16.6667ms
         BIS.B    #TA2,&P3SEL      ; P3.5 (CCI2A) Timer_A input
         BIS      #MCONT,&TACTL    ; Start initialized timer
         ....                      ; Continue
;
; C/C Block 0 uses a repetition rate of 20971 cycles (20ms)
;
TIMMOD0  .EQU     $                ; Start of handler6
         ADD      #20971,&CCR0     ; Prepare next INTRPT
         ...                       ; Task0 starts here
         RETI                      ;
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; (The C/C Blocks 3, 4 and timer overflow are not shown)
;
TIM_HND  .EQU     $                ; Interrupt latency time
```

```
        ADD     &TAIV,PC        ; Add Jump table offset
        RETI                    ; TAIV = 0: No interrupt
        JMP     TIMMOD1         ; TAIV = 2: C/C Block 1
        JMP     TIMMOD2         ; TAIV = 4: C/C Block 2
        JMP     TIMMOD3         ; TAIV = 6: C/C Block 3
        JMP     TIMMOD4         ; TAIV = 8: C/C Block 4
TIMOVH  ...                     ; TAIV = 10: Timer OVFL
;
; C/C Block 1 uses a repetition rate of 17476 cycles (16.67ms)
;
TIMMOD1 .EQU    $               ; Vector 2: C/C Block 1
        ADD     #17476,&CCR1    ; Add time interval
        ...                     ; Task1 starts here
        RETI                    ; Back to main program
;
; C/C Block 2 checks if the time interval between two pos.
; input edges is shorter than a given value in MIN
;
TIMMOD2 .EQU    $               ; Vector 4: C/C Block 2
        BIT     #COV,&CCTL2     ; Frequency much too high?
        JNZ     COV2            ; Yes, overflow!
        PUSH    &CCR2           ; Time of last transition
        SUB     OLDC2,0(SP)     ; Time difference to stack
        ADD     @SP,OLDC2       ; Old + difference = new
        CMP     @SP+,MIN        ; Time interval >= MIN
        JLO     RET2            ; Yes, ok
COV2    MOV.B   #1,ERR          ; No, set error state 1
        BIC     #COV,&CCTL2     ; Reset overflow flag
RET2    RETI                    ; Back to main program
```

The tasks started by the interrupt handlers are not shown; these include:

❑ Incrementing software counters

❑ Checks after regular time intervals (keyboard, watchdog reset, etc.)

❑ Input tests

❑ Update of status bytes, etc.

❑ Measurement intervals

❑ Frequency generation with the output units

6.3.3.2 The Up Mode

The up mode is mainly used for the generation of asymmetric PWM signals. These PWM signals are absolutely synchronous due to the single timer register used for all signals. The period of the PWM repetition frequency is loaded into the period register (CCR0) and the pulse width for each of the outputs, TA1 through TA4, is loaded into the capture/compare registers, CCR1 through CCR4. The formula for a given timer period, t, with respect to the corresponding cycle value n_{CCR0} is ($n_{CCR0} < 65536$):

$$t = \frac{(n_{CCR0} + 1) \times k}{f_{CLK}} \quad \rightarrow \quad n_{CCR0} = \frac{t \times f_{CLK}}{k} - 1$$

The formula for a given pulse width t_{pw} and the corresponding cycle value n (the content of CCRx) is ($n < 65536$):

$$t_{pw} = \frac{n \times k}{f_{CLK}} \quad \rightarrow \quad n = \frac{t_{pw} \times f_{CLK}}{k}$$

As long as no modifications to the period register or the capture/compare registers are made, the PWM signals are repeated without any CPU intervention necessary. The number of timer clock cycles between two equal timer register contents is n_{CCR0} +1.
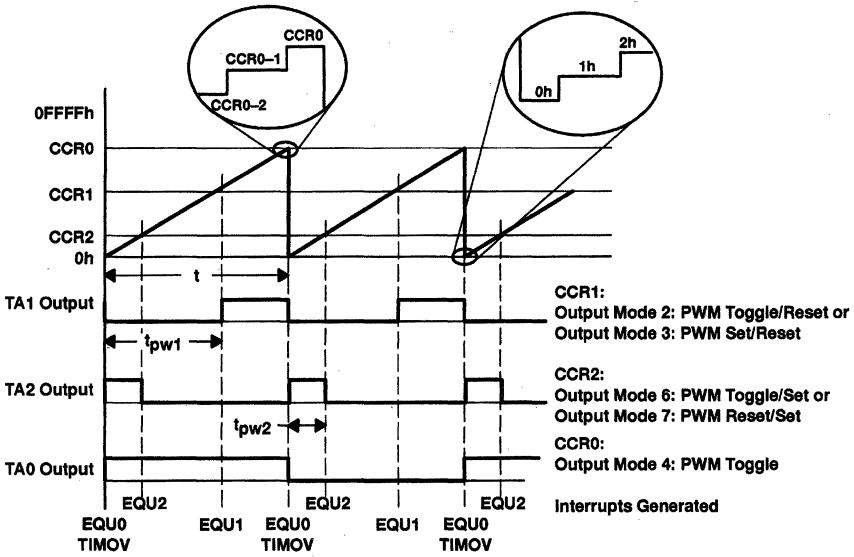
Figure 6–13. Three Different Asymmetric PWM Timings Generated With the Up Mode

If the timer register (TAR) reaches the content of capture/compare register CCRx (EQUx = 1), with compare mode selected for capture/compare block x, then the content of the output unit x is modified. Depending on the output mode defined in the control register CCTLx, the output is toggled, set, reset, or not affected. If the interrupt for the capture/compare block is enabled, an interrupt is also generated.

If the timer register (TAR) counts up to the content of the period register CCR0 (EQU0 = 1), then the timer register (TAR) is reset to 0 with the next timer clock and the content of the output units are toggled, set, reset, or not affected, depending on the selected output mode in control register CCTLx. The timer register continues with the counting starting at 0. If the interrupt for the reaching of CCR0 is enabled, then an interrupt is also requested. See Figure 6–13.

Notes:

The three interrupts caused by the TAIFG flag, the CCIFG0 flag, and CCIFGx flags do not occur simultaneously if used with the up mode:

❏ The CCIFGx flag is set when the capture/compare register x equals the timer register (TAR) (EQUx = 1)

❏ The CCIFG0 flag is prepared when the timer register equals the period register CCR0 (EQU0 = 1). The CCIFG0 flag is delayed one timer clock cycle and set, therefore, together with the TAIFG flag (timer register TAR contains 0)

❏ The TAIFG flag is set when the timer register is reset to 0 (TIMOV = 1)

This means for the up mode: only one interrupt handler is necessary together for the TAIFG flag and the CCIFG0 flag.

If the period register CCR0 contains 0, then the timer register TAR continues counting until it also reaches 0. Then the counting stops until a nonzero value is written to CCR0.

Example 6–22. Three Different Asymmetric PWM Timings Generated With the Up Mode

The software for the example illustrated in figure 6–13 is shown below. capture/compare block 1 generates a negative pulse with output unit 1, capture/compare block 2 generates a positive pulse with the output unit 2 and capture/compare block 0 (the period register block) outputs an evenly spaced output pulse with its output unit 0. The initializing part of the example is also shown. If no tasks must be executed (here tasks 0, 1, and 2), the interrupts may be switched off; the pulse generation continues.

```
; Initialization of the Timer_A: Up Mode, /1, interrupt
; enabled, MCLK = 3.8MHz
;
INIT    MOV     #ISMCLK+D1+CLR+TAIE,&TACTL  ; Prepare Timer_A
        MOV     #OMT+CCIE,&CCTL0  ; CCR0: toggle TA0
        MOV     #OMTR+CCIE,&CCTL1 ; CCR1: toggle/reset TA1
        MOV     #OMRS+CCIE,&CCTL2 ; CCR2: reset/set TA2
        MOV     #190-1,&CCR0      ; fccr0 = 20kHz
        MOV     #114,&CCR1        ; tpw1 = 30us
        MOV     #48,&CCR2         ; tpw2 = 12.6us
        BIS.B   #TA2+TA1+TA0,&P3SEL; Enable TA2, TA1 TA0
        BIS     #MUP,&TACTL       ; Start init. timer. Up Mode
```

```
                 ....                    ; Continue
; Interrupt handler for the Period Register CCR0
; C/C Block 0 outputs a signal with 1/2 of the frequency
; of the other C/C Blocks (50%/50%). It also increments the
; RAM extension of the Timer Register TIMAEXT. It is
; initialized to: toggle (EQU0)
;
TIMMOD0  .EQU    $                      ; Start of handler
         INC     TIMAEXT                ; Incr. timer extension
         ...                            ; Task0 starts here
         RETI                           ;
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
;
TIM_HND  .EQU    $                      ; Interrupt latency time
         ADD     &TAIV,PC               ; Add Jump table offset
         RETI                           ; TAIV = 0: No interrupt
         JMP     TIMMOD1                ; TAIV = 2: C/C Block 1
         JMP     TIMMOD2                ; TAIV = 4: C/C Block 2
         JMP     TIMMOD3                ; TAIV = 6: C/C Block 3
         JMP     TIMMOD4                ; TAIV = 8: C/C Block 4
TIMOVH   ...                            ; TAIV = 10: Timer OVFL
;
; C/C Block 1 outputs a negative pulse automatically. It is
; initialized to: toggle/reset (EQU1/EQU0)
;
TIMMOD1  .EQU    $                      ; Vector 2: C/C Block 1
         ...                            ; Task1 starts here
         RETI                           ; Back to main program
;
; C/C Block 2 outputs a positive pulse automatically. It is
; initialized to: reset/set (EQU2/EQU0)
;
TIMMOD2  .EQU    $                      ; Vector 4: C/C Block 2
         ...                            ; Task2 starts here
         RETI                           ; Back to main program
```

The tasks started by the interrupt handlers are not shown; these may include:

❏ Pulse width modulation for control purposes with the output units

❏ DC generation (DAC) with the output units

❏ Tasks like those shown for the continuous mode, but with special treatment due to the short period. The RAM extension (TIMAEXT) must therefore be taken into account for measurement.

6.3.3.3 The Up/Down Mode

The up/down mode is a symmetric PWM mode. Up to four absolutely synchronous PWM outputs may be generated. The advantage of this PWM mode is a minimum of generated harmonics due to the distributed switching of the output units. The half period of the PWM repetition frequency is loaded into the capture/compare register (CCR0) and a calculated number for the pulse width for each one of the used outputs (TAx) is loaded into the capture/compare registers (CCRx).

The formulas for a given time period, t, of the Timer_A frequency with respect to the corresponding cycle value, n_{CCR0}, are:

$$t = \frac{2 \times n_{CCR0} \times k}{f_{CLK}} \quad \rightarrow \quad n_{CCR0} = \frac{t \times f_{CLK}}{2 \times k}$$

The formulas for a given pulse width time, t_{pw}, with respect to the corresponding cycle value, n, are:

$$t_{pw} = \frac{2 \times n \times k}{f_{CLK}} \quad \rightarrow \quad n = \frac{t_{pw} \times f_{CLK}}{2 \times k}$$

As long as no modifications to the period register or the capture/compare registers are made, the PWM signals are repeated indefinitely without any CPU intervention.
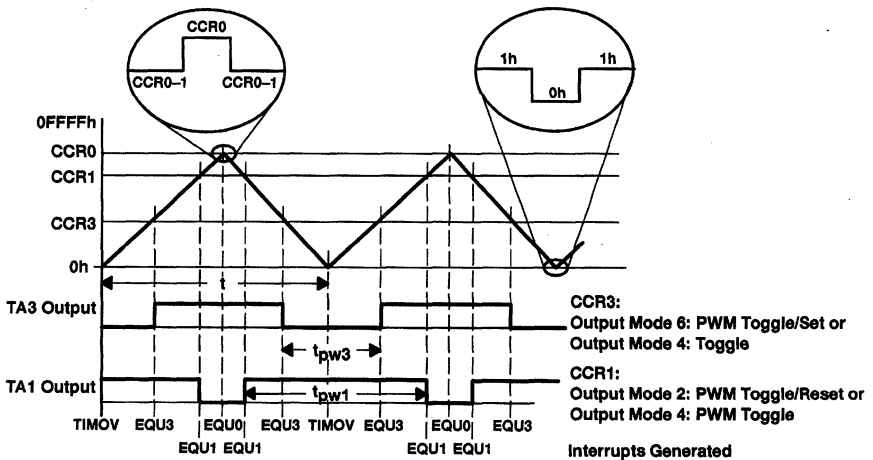
Figure 6–14. Two Different Symmetric PWM Timings Generated with the Up/Down Mode

If the timer register (TAR) reaches the content of capture/compare register, CCRx (EQUx = 1), and the corresponding capture/compare block is switched to the compare mode, then the content of output unit x is modified (toggled, set, reset, or not affected) depending on the output mode of the control register (CCTLx). If the interrupt for the capture/compare block is enabled, then an interrupt is also generated.

The timer register TAR reverses its count direction when it reaches the content of the period register (CCR0), and the content of output unit x is modified again (toggled, set, reset, or not affected) depending on the output mode of the control register (CCTLx). If the interrupt for the reaching of CCR0 is enabled, then an interrupt is also requested. If the timer register reaches the value 0 again, it starts counting upward with the next timer clock cycle. If the interrupt for the reaching of 0 is enabled (TIMOV = 1), then an interrupt is also requested with the TAIFG flag. See Figure 6–14.

Note:

If the period register (CCR0) contains 0, then the timer register (TAR) continues counting until it also reaches 0. Then the counting stops until a nonzero value is written to CCR0.

Example 6–23. Two Different Symmetric PWM Timings Generated With the Up/Down Mode

The software for the example illustrated in figure 6–14 is shown below. capture/compare block 3 generates a negative pulse with output unit 3 at the output TA3. capture/compare block 1 generates a positive pulse — symmetrically to the zero point — with the output unit 1 at the output TA1. The initializing part of the example is also shown. If no software tasks must be executed, the interrupts for the capture/compare blocks 0, 1, and 3 may be switched off.

```
TIMAEXT  .EQU    200h            ; RAM extension (bits 17 - 23)
; Initialization of Timer_A: Up/Down Mode, /2, interrupt
; enabled, MCLK = 3.8MHz
;
INIT    MOV     #ISMCLK+D2+CLR+TAIE,&TACTL  ; Prepare Timer_A
        MOV     #OMOO+CCIE,&CCTL0 ; CCR0: normal I/O pin
        MOV     #OMTR+CCIE,&CCTL1 ; CCR1: toggle/reset TA1
        MOV     #OMTS+CCIE,&CCTL3 ; CCR3: toggle/set TA3
        MOV     #190,&CCR0       ; fccr0 = 5kHz
        MOV     #114,&CCR1       ; tpw1 = 120.0us
        MOV     #48,&CCR3        ; tpw3 = 50.5us
```

```
            BIS.B    #TA3+TA1,&P3SEL   ; Enable TA3 and TA1 outputs
            BIS      #MUPD,&TACTL      ; Start initialized timer
            ....                       ; Continue
; Interrupt handler for the Period Register CCR0
; Block 0 sets the RAM extension TIMAEXT of the Timer Register
; (count down). The LSB of TIMAEXT indicates the
; count direction:               LSB = 0: count up
;                                LSB = 1: count down
; This indication is necessary if the Capture Mode is used.
; The count direction indication is self-synchronizing
;
TIMMOD0  .EQU    $                    ; Start of handler
         BIS     #1,TIMAEXT           ; LSB = 1: count down now
         ...                          ; Task0 starts here
         RETI                         ;
;
; Interrupt handlers and decision (only 3 handlers shown)
;
TIM_HND  .EQU    $                    ; Interrupt latency time
         ADD     &TAIV,PC             ; Add Jump table offset
         RETI                         ; TAIV = 0: No interrupt
         JMP     TIMMOD1              ; TAIV = 2: C/C Block 1
         JMP     TIMMOD2              ; TAIV = 4: C/C Block 2
         JMP     TIMMOD3              ; TAIV = 6: C/C Block 3
         JMP     TIMMOD4              ; TAIV = 8: C/C Block 4
;
; Timer Register reached zero: LSB is set to 0 (count up)
;
TIMOVH   .EQU    $                    ; TIMOV interrupt
         INC     TIMAEXT              ; TAIV = 10: Block 5
         RETI                         ;
;
; C/C Block 1 outputs a positive pulse automatically.
; Initialized to: toggle/reset (EQU1/EQU0)
;
TIMMOD1  .EQU    $                    ; Vector 2: C/C Block 1
```

```
        ...                     ; Task1 starts here
        RETI                    ; Back to main program
;
; C/C Block 3 outputs a negative pulse automatically.
; Initialized to: toggle/set (EQU3/EQU0)
;
TIMMOD3 .EQU    $               ; Vector 6: C/C Block 3
        ...                     ; Task3 starts here
        RETI                    ; Back to main program
```

The tasks started by the interrupt handlers are not shown; these may be:

❑ Symmetric pulse width modulation for control purposes with the output units

❑ DC generation (DAC) with the output units

❑ Tasks like those shown for the continuous mode, but with special treatment due to the changing count direction and short period. The RAM extension TIMAEXT must therefore be taken into account for measurements.

6.3.3.4 The Stop Mode

The stop mode halts the timer register without the change of any control register. The timer actions can then continue on from exactly where they were stopped.

Example 6–24. The Stop Mode

The Timer_A running in up/down mode is stopped. After a certain time, it should continue from exactly where it was halted, including the count direction.

```
BIC     #MUPD,&TACTL    ; Halt Timer_A
...                     ; Proceed without Timer_A
BIS     #MUPD,&TACTL    ; Continue with Up/Down Mode
```

6.3.3.5 Applications of the Timer Modes

Table 6–11 gives an overview of the different applications of the Timer _A modes, together with the capture/compare registers.

Table 6–11. Combinations of Timer_A Modes

COMBINATIONS	CAPTURE/COMPARE REGISTER 0	CAPTURE/COMPARE REGISTER X
Continuous Mode		
Compare register	☐ Interrupt timing ☐ Slow PWM generation ☐ TRIAC timing ☐ SW/HW UART (transmitter) ☐ SW/HW SPI	Same as for capture/compare register 0
Capture register	☐ Capturing of internal and external events ☐ SW/HW UART (receiver) ☐ Revolutions measurement	
Up Mode		
Compare register	Fixed to period register	☐ Interrupt timing ☐ Asymmetric PWM generation ☐ Digital motor control ☐ TRIAC timing ☐ SW/HW UART (transmitter)
Capture register	Not possible due to period register function	☐ Capturing of int. and ext. events ☐ SW/HW UART (receiver) ☐ Revolutions measurement
Up/Down Mode		
Compare register	Fixed to period register	☐ Symmetric PWM generation ☐ Digital motor control
Capture register	Not possible due to period register function	☐ (Capturing of internal and external events is difficult due to up/down counting)

6.3.4 The Timer_A Interrupt Logic

6.3.4.1 Interrupt Sources

Several interrupt sources exist within the Timer_A hardware. An interrupt is requested only if the interrupt of the corresponding timer block is enabled (interrupt enable bit TAIE or CCIEx is set) and the general interrupt enable bit GIE (SR.3) is also set. If more than one interrupt is pending, then the interrupt with the highest priority is first in line for servicing. An interrupt is also requested immediately if any interrupt enable bit (CCIEx or TAIE) is set and the corresponding interrupt flag and GIE (SR.3) were already set.

Timer Register Block — The timer interrupt flag TAIFG requests an interrupt if the timer register reaches 0 and the interrupt enable bit TAIE is set. The TAIFG flag is set, dependent on the actual mode:

❑ **Continuous Mode** — after the overflow from 0FFFFh to 0000h

❑ **Up Mode** — one timer clock after the timer period in CCR0 is reached

❑ **Up/Down Mode** — when the value 0000h is reached during the countdown

Capture/Compare Block x — The capture/compare interrupt Flags CCIFGx are set if one of the following conditions is met. An interrupt is requested only if the corresponding interrupt enable bit CCIEx and GIE are also set.

❑ **Capture Mode** — an input value is captured in register CCRx (the capture condition at the selected input came true)

❑ **Compare Mode** — the timer register counted to the value contained in register CCRx

6.3.4.2 Interrupt Vectors

Two interrupt vectors are associated with the Timer_A module.

❑ The single-source vector for the capture/compare register CCR0 has the highest priority of all Timer_A interrupts. The capture/compare register CCR0 is used to define the timer period during the up mode and the up/down mode. Therefore, it requires the fastest service. This interrupt vector is located at address 0FFF2h.

❑ The multi-source interrupt vector for all other interrupt sources of the Timer_A (capture/compare registers x and Timer Overflow). A 16-bit vector word — the timer vector register (TAIV) — indicates the interrupt with the

highest priority. The register TAIV is normally added to the Program Counter allowing a simple and fast decision without the need for a time consuming *skip chain*. See the section explaining the timer vector register (TAIV) for details. The multi-source interrupt vector is located at address 0FFF0h.

All interrupt flags (CCIFGx and TAIFG) can be accessed by the CPU. The internal priorities of the Timer_A are listed in Table 6–12 (for the MSP430x33x configuration).

Table 6–12.   Timer_A Interrupt Priorities

INTERRUPT PRIORITY	INTERRUPT SOURCE	FLAG NAME	VECTOR ADDRESS
Highest	Capture/Compare Register 0	CCIFG0	0FFF2h
	Capture/Compare Register 1	CCIFG1	0FFF0h
	Capture/Compare Register 2	CCIFG2	0FFF0h
	Capture/Compare Register 3	CCIFG3	0FFF0h
	Capture/Compare Register 4	CCIFG4	0FFF0h
Lowest	Timer Overflow	TAIFG	0FFF0h

Example 6–25. Timer_A Vectors

The following software shows a possible definition for the Timer_A vectors.

```
; Timer_A Interrupt Vectors
;
        .SECT    "TIMVEC",0FFF0h  ; Timer_A Vector Address
        .WORD    TIM_HND          ; Vector for all Blocks except 0
        .WORD    TIMMOD0          ; Vector for Timer Block 0
        .SECT    "INITVEC",0FFFEh ; RESET Vector
        .WORD    INIT             ; Start address
```

6.3.5   The Output Units

Each capture/compare register (CCRx) is connected to an output unit x that controls the corresponding pulse output (TAx). Eight output modes exist and can be selected individually for each capture/compare block by the three output mode bits (OUTMODx) located in the capture/compare control register (CCTLx). For Table 6–13, it is assumed, that the corresponding control signal P3SEL.y is set to 1. See Figure 6–17 for details. The rightmost column of Table 6–13 indicates the behavior of the output TAx if the EQUx and EQU0 signals are valid simultaneously.

Table 6–13.   Output Modes of the Output Units

OUTPUT MODE	MODE NAME	ACTION FOR EQUx	ACTION FOR EQU0	ACTION FOR EQUx .and. EQU0
0	Output only	TAx is set according to bit OUTx (CCTLx.2)		
1	Set	Sets output TAx	No action	Sets output TAx
2	Toggle/Reset	Toggles output TAx	Resets output TAx	Sets output TAx
3	Set/Reset	Sets output TAx	Resets output TAx	Sets output TAx
4	Toggle	Toggles output TAx	No action	Toggles output TAx
5	Reset	Resets output TAx	No action	Resets output TAx
6	Toggle/Set	Toggles output TAx	Sets output TAx	Resets output TAx
7	Reset/Set	Resets output TAx	Sets output TAx	Resets output TAx

The dependence of the output units on the EQU0 signal (shown in Table 6–13) limits the output unit 0 to the following output modes if the up mode or up/down mode is used (the other four output modes output the static signals shown in the rightmost column of Table 6–13).

❑   Output Mode 0

Output Mode — TA0 outputs content of the OUTx bit (CCTLx.2)

❑   Output Mode 1

Set output — TA0 is set from the EQU0 signal

❑   Output Mode 4

Toggle output — TA0 is toggled from the EQU0 signal

❑   Output Mode 5

Reset output — TA0 is reset from the EQU0 signal

If the output mode needs to be changed during the program run (from *Set* to *Reset*, for example), then the output signal TAx will not change its state falsely

if at least one of the three OUTMOD bits retains the 1 state. If this is not the case, (with a change from output mode set to toggle, for example — mode 1 to mode 4), then for a transition time, the output mode 0 may be addressed and will transfer the content of the bit OUTx into the output flip-flop. This may cause glitches at the output terminal.

Figure 6–15 shows the unsafe output mode changes. It indicates that all changes via the output mode 7 are safe.

Mode Transitions							
Output Modes 0	1	2	3	4	5	6	7
Output only	Set	Toggle/Reset	Set/Reset	Toggle	Reset	Toggle/Set	Reset/Set
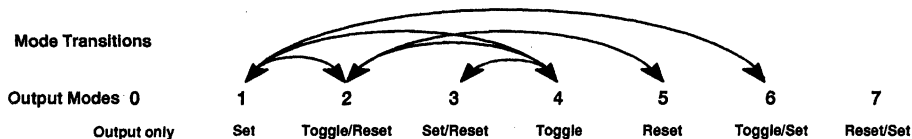
Figure 6–15. Unsafe Output Mode Changes

Example 6–26. Safe Output Mode Changes

The following code may be used for safe changes.

```
; To avoid Output Mode 0, the change is made via Output Mode 7
; Example: Output Mode x to 4
;
        BIS     #OMRS,&CCTL1    ; Set Output Mode 7 (OMRS)
        BIC     #OMSR,&CCTL1    ; Reset LSBs with Output Mode 3
```

If one of the safe changes is possible, then only the different bits are changed:

```
; Change Output Mode from Set to Reset (1 to 5)
;
        BIS     #OMT,&CCTL1     ; Set MSB (OMT) for 1 to 5
;       . . .
; Change Output Mode from Reset to Set (5 to 1)
;
        BIC     #OMT,&CCTL1     ; Reset MSB (OMT) for 5 to 1
```

If, for initialization purposes, a certain state of the output signal TAx is necessary, then the output mode 0 can be used. For the output mode toggle, the output signal OUTx is reset:

```
; Reset output signal TA1 and switch Output Unit 1 to toggle
; mode
```

```
;
        BIC     #OMRS+OUT,&CCTL1  ; OUT1 = 0, Output mode = 0
        BIS     #OMT,&CCTL1       ; Start toggle mode wit OUT1 = 0
```

If the input signals EQU0 and EQUx occur simultaneously, then the output sig-
nal Outx behaves as shown in the rightmost column of Table 6–13.

Figure 6–16 illustrates the simplified structure of the output units. All of the in-
puts that influence the behavior of the output Outx are shown. The reason that
some mode changes are safe and some are not is the NOR gate that decodes
the output mode 0.
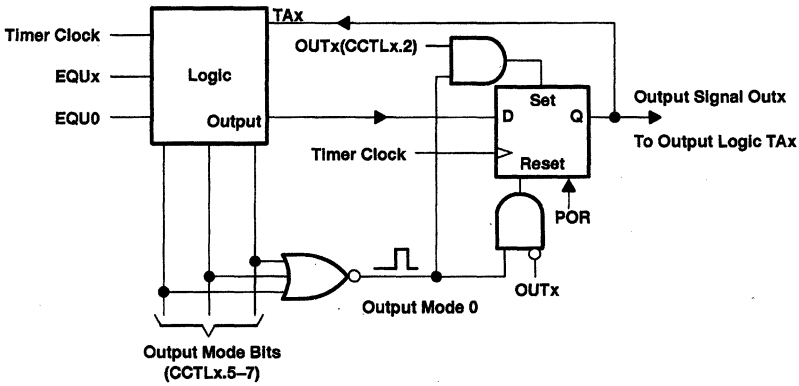
Figure 6–16. Simplified Logic of the Output Units

6.3.5.1 Output Unit I/Os

The bits located in the selection register P3SEL (address 01Bh) and the CAPx bits (CCTLx.8) define the function of the Port3 pins (the MSP430C/P33x configuration is shown — Other family members may use a different implementation, but the principle is the same).

Table 6–14. Timer_A I/O–Port Selection

P3SEL.y = 0	P3SEL.y = 1 CAPx = 0	P3SEL.y = 1 CAPx = 1
Port I/O P3.0	Port I/O P3.0	Port I/O P3.0
Port I/O P3.1	Port I/O P3.1	Port I/O P3.1
Port I/O P3.2	Timer Clock input TACLK	Timer clock input TACLK
Port I/O P3.3	Output TA0	Capture input CCI0A
Port I/O P3.4	Output TA1	Capture input CCI1A
Port I/O P3.5	Output TA2	Capture input CCI2A
Port I/O P3.6	Output TA3	Capture input CCI3A
Port I/O P3.7	Output TA4	Capture input CCI4A

Figure 6–17 illustrates the Timer_A interface to the external world. Six Port3 I/O terminals (MSP430C33x) may be selected individually as normal Port3 I/Os or as Timer_A I/Os. The control bit P3SEL.y selects the function:

❏ P3SEL.y = 0

The I/O pin is connected to the Port3 module (input or output)

❏ P3SEL.y = 1

The I/O pin is connected to the Timer_A module (TAx output or CCIxA input)
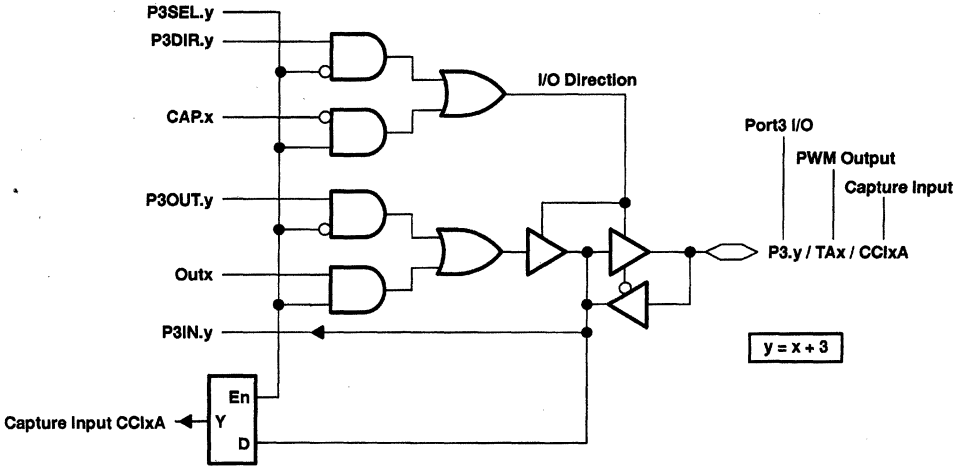
Figure 6–17. Connection of the Port3 Terminals to the Timer_A (MSP430C33x Configuration)

Example 6–27. Port3 Output Control

The initialization for the use of the TA2 and TA1 outputs for PWM is shown. They are disconnected from the Port3 logic by the setting of the bits P3SEL.5 and P3SEL.4.

```
;
; Initialize the Timer_A: MCLK, Stop Mode, INTRPT enabled, /2
;
        MOV     #ISMCLK+D2+CLR+TAIE,&TACTL ; Define Timer_A
        MOV     #200-1,&CCR0       ; Define period 200 cycles
;
; Initialize Control Registers CCTL2 and CCTL1: Reset/set
; mode, INTRPT enabled, Compare Mode, clear flags
;
        MOV     #OMRS+CCIE,&CCTL1 ; CCIFG1 = 0,
        MOV     #OMRS+CCIE,&CCTL2 ; CCIFG2 = 0
;
; Initialize Capture Compare Registers to PWM duty
;
```

```
        MOV     #100,&CCR1      ; 50% PWM

        MOV     #50,&CCR2       ; 25% PWM
;
; Prepare Timer_A Output Units TA2 and TA1 (P3.5 and P3.4)
;
        MOV.B   #TA2+TA1,&P3SEL  ; Connect to Output Units
;
        BIS     #MUP,&TACTL     ; Start Timer_A in Up Mode
```

6.3.5.2 Pulse Width Modulation in the Continuous Mode

The continuous mode is not intended for PWM, but may be used for this purpose in two ways. The timing can be controlled from:

❏ One capture/compare register only

❏ One capture/compare register and additional capture/compare register 0

6.3.5.2.1 One Capture/Compare Register only

The same capture/control register x sets and resets the output TAx. The output modes *toggle* or alternating *set* and *reset* are used. For the second method (Set and Reset), the interrupt handler modifies the output mode in addition to the adding of the time interval to the register CCRx. PWM values near 0% and 100% must be realized with software. See also Section 6.3.6 *The Limitations of Timer_A*.

The output modes and their usability for the first method of PWM in the continuous mode are listed below:

❏ **Set Mode** — used to get the output signal into the set state. It is necessary to alternate with the reset mode to get a PWM output signal

❏ **Toggle/Reset** — not usable due to the influence of capture/compare register 0

❏ **Set/Reset** — not usable due to the influence of capture/compare register 0

❏ **Toggle** — usable, but a defined start position must be initialized. Otherwise, an inverted output signal is generated

❏ **Reset Mode** — used to get the output signal into the reset state. It is necessary to alternate with the set mode to get a PWM output signal

□ **Toggle/Set** — not usable due to the influence of capture/compare register 0

□ **Reset/Set** — not usable due to the influence of capture/compare register 0
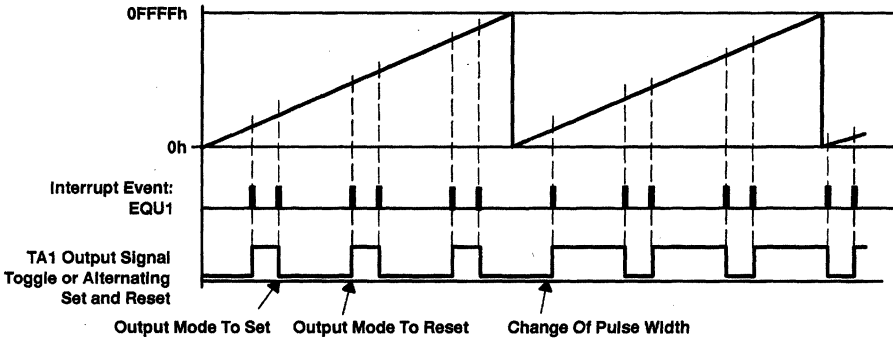
Figure 6–18. PWM Generation in the Continuous Mode (CCR1 only controls TA1)

6.3.5.2.2 One Capture/Compare Register and Additional Capture/Compare Register 0

The capture/compare register CCR0 has the same function as with the other two timer modes: it switches back the PWM output TAx into a defined state. Figure 6–19 shows PWM generation using the *reset/set* mode. This method allows PWM with higher repetition rates than the method described previously. With no pulse width modifications, the time interval between two interrupts are always identical.

The capture/compare register 0 may be used for more than one PWM output used this method. The output frequency of capture/compare register 0 may be chosen in such a way that also supports other purposes — an auxiliary frequency output at TA0, for example. See also Figure 6–13.

The output modes and their usability for the second method of the continuous mode are listed below:

□ **Set** — used to get the output signal TAx into a defined set state initially.

□ **Toggle/Reset** — usable, self-synchronizing PWM

□ **Set/Reset** — usable, self-synchronizing PWM

☐ **Toggle** — not usable due to the missing influence of capture/compare register 0

☐ **Reset Mode** — used to get the output signal TAx into a defined reset state initially

☐ **Toggle/Set** — usable, self-synchronizing PWM

☐ **Reset/Set** — usable, self-synchronizing PWM
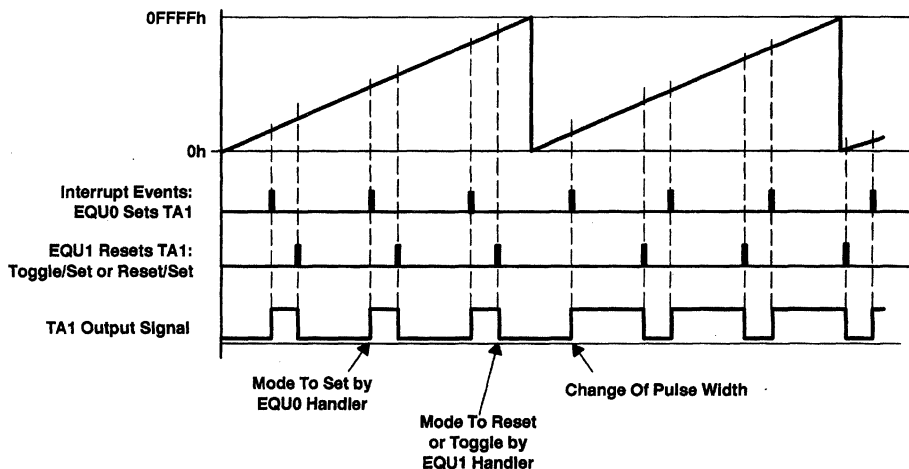
Figure 6–19. PWM Generation in the Continuous Mode (CCR0 and CCR1 control TA1)

Example 6–28. PWM near 0% and 100%

The PWM output values near 0% and 100% must be realized with special software code. A simple way to do this is to use the timer vector register (TAIV) once more after each completed interrupt handler to check to see if another timer interrupt is pending. The software example below shows this solution. It is applicable to all PWM modes. See figure 6–19. It saves 9 to 11 cycles if an additional Timer_A interrupt is pending.

```
PWMper    .EQU    333             ; PWM period (Timer Clock cycles)
;
; Interrupt handler for the Period Register CCR0.
; To handle PWM duties near 0% or 100% a check is made if
; other timer interrupts are pendent: return to TIM_HND
```

```
;
TIMMOD0  .EQU    $                   ; Start of handler
         INC     TIMAEXT             ; Incr. timer extension
         ADD     #PWMper,&CCR0       ; Add period length to CCR0
         ...                         ; Task0 starts here
         ...                         ; Fall through to TIM_HND
;
; Interrupt handlers for Capture/Compare Blocks
;
TIM_HND  .EQU    $                   ; Interrupt latency time
         ADD     &TAIV,PC            ; Add Jump table offset
         RETI                        ; TAIV = 0: No interrupt
         JMP     TIMMOD1             ; TAIV = 2: C/C Block 1
         JMP     TIMMOD2             ; TAIV = 4: C/C Block 2
         JMP     TIMMOD3             ; TAIV = 6: C/C Block 3
         JMP     TIMMOD4             ; TAIV = 8: C/C Block 4
TIMOVH   ...                         ; TAIV = 10: Block 5
;
; C/C Block 1 returns to the timer interrupt handler after
; completion to look for pendent timer interrupts
;
TIMMOD1  .EQU    $                   ; Vector 2: C/C Block 1
         ADD     #PWMper,&CCR1       ; Add period length to CCR1
         ...                         ; Task1 starts here
         JMP     TIM_HND             ; Pendent INTRPTs ?
```

6.3.5.3 Pulse Width Modulation in the Up Mode

The up mode permits all pulse widths from 0% to 100% without any special treatment necessary. The calculation software delivers results ranging from 0 to $n_{CCR0}+1$. Like Figure 6–20 illustrates, the full range of PWM output signals is possible.

The output modes and their usability for the up mode are listed below.

❏ **Set Mode** — used to get the output signal initially into a defined set state.

❏ **Toggle/Reset** — outputs self–synchronizing negative pulses without CPU activity.

❑ **Set/Reset** — outputs self–synchronizing negative pulses without CPU activity.

❑ **Toggle** — this mode cannot be used with the up mode. It outputs a signal with 50% duty and doubled period for all contents of register CCRx, except for CCRx > CCR0. These contents retain the last state of output Outx due to the missing EQUx signal.

❑ **Reset Mode** — used to get the output signal initially into a defined reset state.

❑ **Toggle/Set** — outputs self-synchronizing positive pulses without CPU activity.

❑ **Reset/Set** — outputs self-synchronizing positive pulses without CPU activity.

Figure 6–20 illustrates the four usable output modes for PWM in the up mode.

Note:

No interrupts are generated from the capture/compare blocks x for CCRx = 0 and for CCRx > CCR0. For these two cases, a special treatment is necessary. See the software examples in section *Software Examples for the Up Mode*.
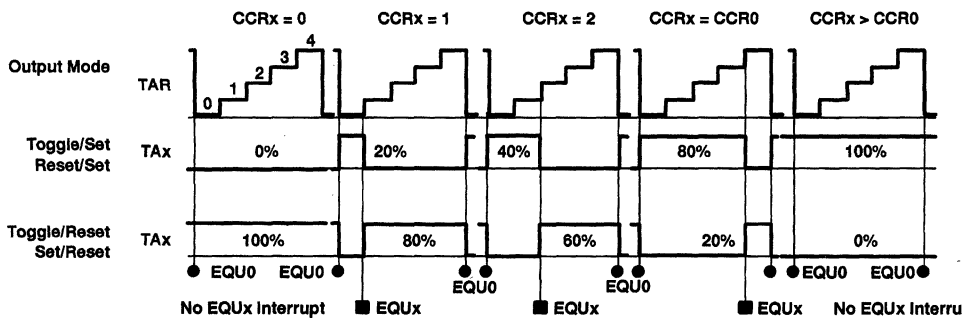
Figure 6–20. PWM Signals at TAx in the Up Mode (CCR0 contains 4)

6.3.5.4 Pulse Width Modulation in the Up/Down Mode

The output modes and their usability for the up/down mode are listed below.

❑ **Set Mode** — used to get the output signal initially into a defined set state.

❑ **Toggle/Reset** — outputs self-synchronizing positive pulses without CPU activity. The Timer_A hardware can produce all of the theoretically possible $n_{CCR0}+1$ states. But special treatment is necessary if register CCRx contains 0. Then the output signal Outx toggles only once per period, which means the output shows a 50% duty and not 0%. See figure 6–21.

❑ **Set/Reset** — cannot be used with up/down mode.

❑ **Toggle** — should not be used with the up/down mode.

❑ **Reset Mode** — used to get the output signal initially into a defined reset state.

❑ **Toggle/Set** — outputs self-synchronizing negative pulses without CPU activity. See *Toggle/Reset*, above, for restrictions.

❑ **Reset/Set** — cannot be used with up/down mode.

As figure 6–21 also shows, the missing PWM values of 0% for toggle/reset and 100% for toggle/set can be output if CCRx contains a greater value than CCR0.

Example 6–29. Pulse Width Modulation in the Up/Down Mode

The checking software for output mode toggle/reset is shown. All PWM values from 1 to n_{CCR0} are valid. The value 0 is emulated by a number greater than n_{CCR0}. R5 contains the calculated PWM value.

```
; PWM value in R5 is checked to be in limits 1 to nCCR0
;
        CMP     R5,&CCR0        ; PWM value =< nCCR0?
        JHS     L$1             ; Yes, proceed
        MOV     &CCR0,R5        ; No, upper limit (100% PWM)
;
; If 0% PWM is needed: 0FFFFh to R5
;
L$1     TST     R5              ; Zero value?
        JNZ     L$2             ; No, proceed
```

```
        MOV      #0FFFFh,R5        ; Use largest, unsigned number
L$2     ...                        ; Result in R5 is in limits
```

The above correction limits the maximum period length n_{CCR0} to 0FFFEh.

Figure 6–21 illustrates the two possible PWM modes for the up/down mode. They correct themselves after one period, max.
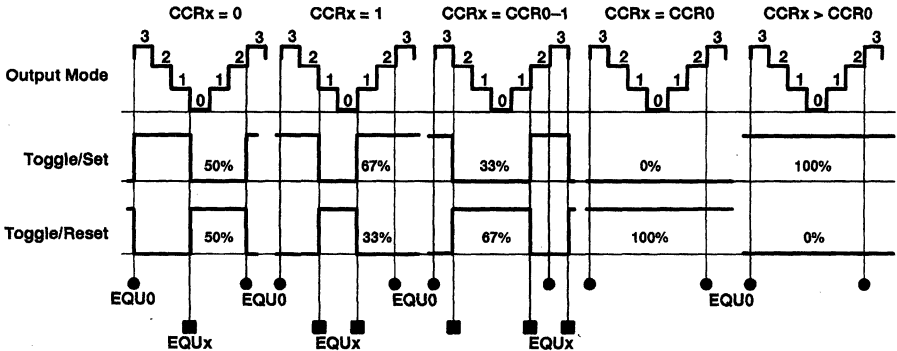
Figure 6–21. PWM Signals at Pin TAx With the Up/Down Mode (CCR0 contains 3)

6.3.6   Limitations of the Timer_A

This section details how to check to see if the limitations imposed by the architecture of the Timer_A are not exceeded. The abbreviations used in this chapter are:

t_{intrpt}	Time for a complete interrupt sequence	[s]
P_{task}	Executed MCLK cycles for the task itself during the interrupt handler (e.g. incrementing of a counter). The necessary cycle count of an instruction depends on the addressing modes used.	[s]
P_{ovhd}	Sum of MCLK cycles for the overhead of an interrupt sequence. See software overhead.	[s]
f_{MCLK}	System clock frequency MCLK	[Hz]
f_{rep}	Repetition rate of an event (e.g. an interrupt request)	[Hz]
u_{CPU}	CPU loading by a given task. Ranges from 0 to 1 (100%)	
t_{ILmax}	Maximum (worst case) of the interrupt latency time due to other enabled interrupts	[s]

The execution time t_{intrpt} for a complete interrupt sequence is the same for all three timer modes:

$$t_{intrpt} = \frac{P_{task} + P_{ovhd}}{f_{MCLK}}$$

The software overhead p_{ovhd} differs slightly for the three possible Timer_A interrupt sources:

❏ Capture/Compare Block CCR0   11 cycles (6 + 0 + 5)

❏ Capture/Compare Blocks CCR1 to CCR4   16 cycles (6 + 5 + 5)

❏ Timer Overflow TAIFG   14 cycles (6 + 3 + 5)

The software overhead p_{ovhd} consists of three parts:

■ Getting to the first instruction of the interrupt handler by the CPU (6 cycles)

■ Decision part: addition of timer vector register (TAIV) to the program counter and execution of the JMP instruction (0 to 5 cycles)

■ Return from interrupt instruction RETI (5 cycles).

These software overhead cycles refer to the minimized software structure shown in all software examples. This structure is valid for all three timer modes.

To get the complete interrupt loading, all execution times of the enabled interrupts are summed up during one period.

To get the loading u_{CPU} (ranging from 0 to 1) of the CPU by the interrupt activity, the following formula is used:

$$u_{CPU} = \sum (t_{intrpt} \times f_{rep})$$

EXAMPLE
Two Timer_A interrupts are active in continuous mode. The system clock frequency f_{MCLK} is 2.097MHz.

1) CCR1: repetition rate 1.2 kHz — 16 cycles for the task, 16 cycles overhead

2) CCR3: repetition rate 2.0 kHz — 22 cycles for the task, 16 cycles overhead

$$u_{CPU} = \sum \left(\frac{16 + 16}{2.097E6} \times 1.2E3 + \frac{22 + 16}{2.097E6} \times 2.0E3 \right) = 0.0545$$

The above result means a CPU loading of approximate 5.5% due to the Timer_A.

6.3.6.1 Limitations of the Continuous Mode

Interrupt Handling — the shortest repetitive time interval, t_{CRmin}, between two similar timer events using a compare register CCRx is:

$$t_{CRmin} = t_{ILmax} + \frac{p_{taskmax} + p_{OVHD}}{f_{MCLK}}$$

The shortest repetitive time interval, t_{CLmin}, between two interrupt events using a capture register CCRx is:

$$t_{CLmin} = t_{ILmax} + \frac{p_{taskmax} + p_{OVHD}}{f_{MCLK}}$$

The time, $t_{taskmax}$, for the capture mode is the time to read the captured time value and to test and reset the COV flag.

☐ **Software Overhead** — the interrupt loading ranges from one interrupt request (request from CCIFGx) up to six interrupt requests (requests from TAIFG, CCIFG0, and all CCIFGx flags).

☐ **Output Units** — for relatively high PWM repetition rates special treatment may be necessary for PWM duties near the limits 0% and 100%.

Maximum Resolution:

$$r = \frac{k}{f_{CLK}}$$

where: r is equivalent to the period of the timer clock

6.3.6.2 Limitations of the Up Mode

☐ **Interrupt Handling** — the worst case sum of all the execution times needed by all interrupts during one timer period must be less than the timer period (defined by the period register, CCR0). Otherwise, the interrupt part will loose the synchronization due to overload.

This means:

$$\frac{(n_{CCR0} + 1) \times k}{f_{CLK}} > \frac{1}{f_{MCLK}} \sum_{t=0}^{t = \frac{(n_{CCR0} + 1) \times k}{f_{CLK}}} p_{taskmax} + p_{ovhd}$$

☐ **Software Overhead** — the overhead ranges from zero (PWM is output automatically after the loading of the timer registers), up to six interrupt requests per period (interrupt requests from TAIFG, CCIFG0, and from all CCIFGx flags).

☐ **Output Units** — all values ranging from 0% to 100% for pulse width modulation (PWM) are possible without special treatment.

☐ **Maximum Resolution** — for a given repetition rate, f_{rep}, of the timer output, a maximum resolution, r, is possible:

$$r = \frac{f_{MCLKmax}}{f_{rep}} = n_{CCR0} + 1$$

This means that with a maximum system clock frequency of 4 MHz and a repetition rate of 20 kHz for a PWM output — due to audibility — a resolution of 200 steps is possible (0.5%).

6.3.6.3 Limitations of the Up/Down Mode

☐ **Interrupt Handling** — the worst case sum of all the execution times needed by all interrupts during one timer period must be less than the doubled period defined by the period register CCR0. Otherwise, the interrupt part will loose the synchronization due to overload.

$$\frac{2 \times n_{CCR0} \times k}{f_{CLK}} > \frac{1}{f_{MCLK}} \sum_{t=0}^{t=\frac{2 \times n_{CCR0} \times k}{f_{CLK}}} p_{taskmax} + p_{ovhd}$$

☐ **Software Overhead** — the overhead ranges from zero (PWM is output automatically after the loading of the timer registers) up to ten interrupt requests per full period (interrupt requests from TAIFG, CCIFG0, and 2 interrupts per CCIFGx).

☐ **Output Units** — the pulse width zero (0%) needs a special software treatment. Without this, the hardware outputs a 50% pulse width instead. This behavior will be changed in future versions.

☐ **Maximum Resolution** — for a given repetition rate, f_{rep}, of the timer output, a maximum resolution, r, is possible:

$$r = \frac{f_{MCLKmax}}{2 \times f_{rep}} = n_{CCR0}$$

This means that with a maximum system clock frequency of 4 MHz and a repetition rate of 20 kHz for a PWM output — due to audibility — a resolution of 100 steps is possible (1.0%). The resolution of the up/down mode is less than it is in the up mode. With the same timer clock, the up mode delivers ($n_{CCR0}+2$) different pulse widths and the up/down mode delivers ($n_{CCR0}+1$) different pulse widths — but with a reduced output frequency due to the up and down counting. This means the resolution is approximately one half the resolution of the up mode.

6.3.7 Miscellaneous

The frequencies generated by the Timer_A may also be used as the timebase for other tasks if defined appropriately:

❑ **Serial Communication Interface (SCI)** — If for an MSP430, a second UART (RS232) is needed, then with a timer frequency of 19.2 kHz (8×2.4 kHz) a software UART with 2400 baud can be implemented. This software UART uses the interrupt generated with the reaching of the content of the period register, CCR0 (CCIFG0 = 1), for the synchronization of the UART software.

❑ **Timing Intervals for Control** — These important control values can also be derived from the timer frequency by an appropriate software prescaling. This timing may be used for calculations, keyboard scan, measurement starts, etc.

6.3.8 Software Examples for the Continuous Mode

This section shows several proven application examples for the Timer_A. Whenever possible, the abbreviations used in the *Architecture Guide and Module Library* are used.

All examples use the value FLLMPY — it defines the master clock frequency fMCLK.

$$f_{MCLK} = FLLMPY \times f_{crystal}$$

If this frequency, fMCLK, is too high for the application (for example: it causes values for the timer registers exceeding the 16-bit range), then the input divider of the Timer_A may be used. It allows a prescaling by 1, 2, 4, and 8. For prescaling by 2, the definitions at the start of each example are simply changed to:

```
FLLMPY    .equ    100             ; FLL multiplier for 3.2768MHz
TCLK      .equ    FLLMPY*32768/2  ; Timer Clock = 1.6384MHz
;
; The Input Divider D2 is used to get MCLK/2 for the TCLK
;
          MOV     #ISMCLK+D2+TAIE+CLR,&TACTL ; Use D2 divider
```

> **Note:**
>
> The software and hardware examples shown here are specific to the MSP430C/P33x family. Other MSP430 family members may use other I/O ports and addresses for the Timer_A registers and signals. The programming principles are unchanged — only address definitions may need to be modified.

The software examples were tested with the software simulator and an EVK330 evaluation kit.

For all examples, the loading of the CPU is given. The terms used are defined below:

❏ **Overhead** — the sum of necessary CPU cycles to get to the first instruction of the interrupt handler and to get back to the interrupted program sequence (wakeup cycles, storing of PC and SR, determination of the interrupt source, and RETI cycles)

❏ **Task** — the CPU cycles used for the interrupt task: incrementing of a counter, calculations, etc.

Advantages of the Continuous Mode:

  ▪ Five complete, independent timings and captures are possible. Any mix is possible

  ▪ No dominance by a period register

Disadvantages of the Continuous Mode:

  ▪ Software update necessary for the capture/compare registers to allow continuous run

  ▪ Speed limit due to the necessary software update

6.3.8.1 Common Initialization Subroutine

The initialization subroutine INITSR is used by all examples. It executes the following tasks:

❏ A check is made if the initialization subroutine is called after applying the supply voltage (the RAM word INITKEY does not contain 0F05Ah) or after an external reset or watchdog reset (INITKEY contains 0F05Ah). If the applying of the supply voltage caused the reset, then the RAM is cleared and the INITKEY is initialized to 0F05Ah.

❏ The system clock oscillator is programmed with the FLL multiplier N. This defines the MCLK frequency f$_{MCLK}$. See above.

❏ The correct DCO switch FN_x for the chosen MCLK frequency (f$_{MCLK}$) is set. These switches allow the system clock oscillator to operate with one of the center taps of the digitally controlled oscillator (DCO). This way the DCO operates always in a nonsaturated condition.

☐ A delay of 30000 clock cycles is included to give the oscillator time to settle at the correct frequency.

```
; Common Initialization Subroutine
; Check the INITKEY value first:
; If value is 0F05Ah: a reset occurred, RAM is not cleared
; otherwise Vcc was switched on: complete initialization
;
INITSR   CMP      #0F05Ah,INITKEY   ; PUC or POR?
         JEQ      IN0               ; Key is ok, continue program
         CALL     #RAMCLR           ; Restart completely: clear RAM
         MOV      #0F05Ah,INITKEY   ; Define "initialized state"
;
IN0      MOV.B    #FLLMPY-1,&SCFQCTL ; Define MCLK frequency
;
         .if      FLLMPY < 48       ; Use the right DCO current:
         MOV.B    #0,&SCFI0         ; MCLK < 1.5MHz: FN_x off
         .else
         .if      FLLMPY < 80       ; 1.5MHz < MCLK < 2.5MHz?
         MOV.B    #FN_2,&SCFI0      ; Yes, FN_2 on
         .else                      ;
         .if      FLLMPY < 112      ; 2.5MHz < MCLK < 3.5MHz?
         MOV.B    #FN_3,&SCFI0      ; Yes, FN_3 on
         .else
         MOV.B    #FN_4,&SCFI0      ; MCLK > 3.5MHz: FN_4 on
         .endif
         .endif
         .endif
;
         MOV      #10000,R5         ; Allow the FLL to settle
IN1      DEC      R5                ; at the correct DCO tap
         JNZ      IN1               ; during 30000 cycles
         RET                        ; Return from initialization
;
; Subroutine for the clearing of the RAM block
;
         .bss     INITKEY,2,0200h   ; 0F05Ah: initialized state
```

```
RAMSTRT  .equ    0200h            ; Start of RAM
RAMEND   .equ    05FEh            ; Highest RAM address (33x)
;
RAMCLR   CLR     R5               ; Prepare index register
RCL      CLR     RAMSTRT(R5)      ; 1st RAM address
         INCD    R5               ; Next word address
         CMP     #RAMEND-RAMSTRT+2,R5 ; RAM cleared?
         JLO     RCL              ; No, once more
         RET                      ; Yes, return
```

6.3.8.2 Generation of Five Independent Timings

The software example explains the use of the timer vector register (TAIV) and the overhead of the interrupt handling. It refers to figure 6–22. The interrupt handler of timer block x adds the appropriate time interval, Δt, to the corresponding compare register, CCRx. The MCLK frequency (3.2768 MHz) is used also for the timer clock. The five timings generated are defined as follows (see also Table 6–15):

❑ **Capture/Compare Block 0** — a positive pulse with a 10 kHz repetition rate is generated and output at terminal TA0. The pulse is reset by the interrupt handler of timer block 0. The pulse is used for the precise triggering of an external analog-to-digital converter. The error of the repetition rate due to the MCLK frequency used is –0.097%

❑ **Capture/Compare Block 1** — an internal interrupt with variable timing is generated. The cycle count is stored in the RAM word TIM1REP. The maximum value of this cycle count is 0FFFFh, the minimum value is 1000. The output terminal TA1 is not used.

❑ **Capture/Compare Block 2** — a square wave with a fixed 1 kHz repetition rate is generated and output at terminal TA2. The pulse is used as a reference for external devices. The error of the repetition rate due to the MCLK frequency used is –244 ppm.

❑ **Capture/Compare Block 3** — an internal interrupt with a fixed 200 Hz repetition rate is generated. The output terminal TA3 is not used. The error of the repetition rate due to the MCLK frequency used is –244 ppm.

❑ **Capture/Compare Block 4** — a square wave with a variable output frequency is generated and output at terminal TA4. The output frequency starts at 409.6 Hz (4000 cycles) and increases up to 1638.4 Hz (1000 cycles). The square wave is used for the control of an external DC/DC converter.

The formula for calculating the value, Δn, that is added to the timer register (TAR) depends on the application. For the internal interrupts (CCR1 and CCR3 in the example) and the external pulse (CCR0 in the example) Δn is:

$$\Delta n = \frac{\Delta t \times f_{CLK}}{k}$$

For the external-generated square wave signals with the frequency fext (CCR2 and CCR4 in the example) Δn is:

$$\Delta n = \frac{f_{CLK}}{k \times 2 \times f_{ext}}$$

Where:

f_{CLK}	Frequency at the input of the input divider	[Hz]
f_{ext}	Frequency to be output with toggle mode	[Hz]
k	Input divider constant (1, 2, 4, 8)	
Δt	Time interval to be generated	[s]

Table 6–15. Short Description of the five independent Timings

CAPTURE/ COMPARE BLOCK	TIME INTERVAL (TIMER CLOCK CYCLES)	SIGNAL TYPE	COMMENT
0	328	External	Pulse: 10kHz ADC repetition rate
1	Variable	Internal	Cycle count stored in TIM1REP (min 1000)
2	1638	External	1 kHz @ 3.2768 MHz (error: –244 ppm)
3	16384	Internal	Fixed frequency 200 Hz
4	4000 to 1000	External	Increasing frequency for ext. DC/DC converter

The software example is written for an fMCLK of 3.276 MHz. If other frequencies are used, the time intervals need to be adapted. Subsequent examples show methods of writing frequency-independent software. Figure 6–22 illustrates the five timings described above:
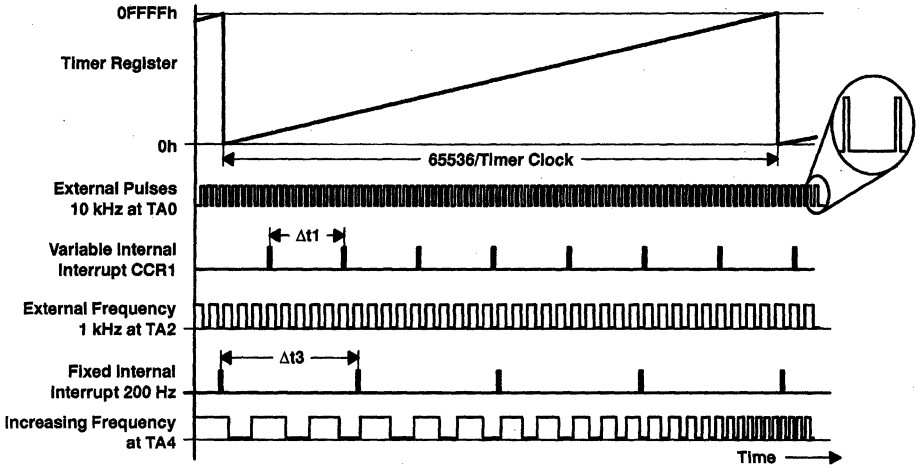
Figure 6–22. Five Independent Timings Generated in the Continuous Mode

The timing of the signals output at the TAx pins (the dedicated I/O pins of the Timer_A) is independent of interrupt latency: the TAx outputs are set, reset or toggled exactly at the programmed time (contained in the capture/compare register x) by the output unit x. The requested interrupt when this occurs is used to update the capture/compare register x and to execute necessary tasks.

Example 6–30. Five independent Timings Generated in the Continuous Mode

The software example also shows how to output the MCLK frequency at the output terminal XBUF for reference purposes. For example, an external ASIC may be driven by this frequency.

```
; Software example: five independent timings using the
; Continuous Mode of the 16-bit Timer_A
;
; Hardware definitions
;
FLLMPY    .equ    100              ; FLL multiplier for 3.2768MHz
TCLK      .equ    FLLMPY*32768     ; TCLK: FLLMPY x fcrystal
STACK     .equ    600h             ; Stack initialization address
;
; RAM definitions
```

```
;
TIM1REP  .equ    202h              ; Repetition rate Block 1
TIM4REP  .equ    204h              ; Repetition rate Block 4
TIMAEXT  .equ    206h              ; Extension for Timer Register
;
         .text 0F000h              ; Software start address
;
INIT     MOV     #STACK,SP         ; Initialize Stack Pointer
         CALL    #INITSR           ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT on
;
         MOV     #ISMCLK+TAIE+CLR,&TACTL
         MOV     #OMSET+CCIE,&CCTL0 ; Set, INTRPT on
         MOV     #OMOO+CCIE,&CCTL1 ; No output, INTRPT on
         MOV     #OMT+CCIE,&CCTL2  ; Toggle, INTRPT on
         MOV     #OMOO+CCIE,&CCTL3 ; No output, INTRPT on
         MOV     #OMT+CCIE,&CCTL4  ; Toggle, INTRPT on
         MOV     #0FFFFh,TIM1REP   ; Start value Block 1
         MOV     #4000,TIM4REP     ; Start value Block 4
         MOV.B   #TA4+TA2+TA0,&P3SEL ; Define TAx outputs
;
         MOV     #1,&CCR0          ; Immediate start
         MOV     #1,&CCR1          ; with defined contents
         MOV     #1,&CCR2          ; for the Capture/Compare
         MOV     #1,&CCR3          ; Registers
         MOV     #1,&CCR4          ;
         CLR     TIMAEXT           ; Clear TAR extension
         MOV.B   #CBMCLK+CBE,&CBCTL ; Output MCLK at XBUF pin
         BIS     #MCONT,&TACTL     ; Start Timer
         EINT                      ; Enable interrupt
MAINLOOP ...                       ; Continue in background
;
; Interrupt handler for Capture/Compare Block 0. An ext. ADC
; is started every 100us (328 cycles @ 3.2768MHz MCLK) with
; a positive pulse at TA0 (set exactly from Output Unit).
```

```
; The interrupt flag CCIFG0 is reset automatically.
;
TIMMOD0   .EQU    $                   ; Start of handler
          ADD     #328,&CCR0          ; Prepare next INTRPT (10kHz)
          BIC     #OMRS+OUT,&CCTL0    ; Reset TA0
          BIS     #OMSET,&CCTL0       ; Back to Set Mode
          RETI                        ; Return from Interrupt
;
; Timer Block 3 generates an internal used 5ms interrupt
; 16384/3.2768MHz = 0.005s
;
TIMMOD3   .EQU    $                   ; Vector 6: Block 3
          ADD     #16384,&CCR3        ; Add time interval (5ms)
          ...                         ; Task3 starts here
                                      ; Fall through to TIM_HND
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
TIM_HND   .EQU    $                          ; Interrupt latency
          ADD     &TAIV,PC            ; Add Jump table offset
          RETI                               ; Vector 0: No interrupt
          JMP     TIMMOD1             ; Vector 2: Block 1
          JMP     TIMMOD2             ; Vector 4: Block 2
          JMP     TIMMOD3             ; Vector 6: Block 3
          JMP     TIMMOD4             ; Vector 8: Block 4
;
; Block 5. Timer Overflow Handler: the Timer Register is
; expanded into the RAM location TIMEXT (MSBs)
;
TIMOVH    .EQU    $                   ; Vector 10: TIMOV Flag
          INC     TIMAEXT             ; Incr. Timer extension
          RETI                        ;
;
; Block 1 uses a variable repetition rate defined in TIM1REP
```

```
; Repetition Rate = 3.2768MHz/(TIM1REP)
;
TIMMOD1  .EQU    $                 ; Vector 2: Block 1
         ADD     TIM1REP,&CCR1     ; Add time interval
         ...                       ; Task1 starts here
         RETI                      ; Back to main program
;
; The used time interval delta t2 is 1638 cycles. This
; delivers an external 1kHz signal (1638/3.2768MHz = 500us)
;
TIMMOD2  .EQU    $                 ; Vector 4: Block 2
         ADD     #1638,&CCR2       ; Add time interval (1/2 period)
         ...                       ; Task2 starts here
         RETI                      ; Back to main program
;
; Block 4 uses a variable repetition rate starting at 4000
; cycles and going down to 1000 cycles. It is used for an
; external DC/DC converter. Toggle Mode is used
;
TIMMOD4  .EQU    $                 ; Vector 8: Block 4
         ADD     TIM4REP,&CCR4     ; Add time interval (1/2 period)
         CMP     #1000,TIM4REP     ; Final value reached?
         JLO     T41               ; Yes, no modification
         SUB     #1,TIM4REP        ; No, modify interval
T41      RETI                      ; Back to main program
;
         .sect   "TIMVEC",0FFF0h   ; Timer_A Interrupt Vectors
         .word   TIM_HND           ; Timer Blocks 1 to 4
         .word   TIMMOD0           ; Vector for Timer Block 0
         .sect   "INITVEC",0FFFEh  ; Reset Vector
         .word   INIT
```

The example above results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:
f_{MCLK} Frequency of the DCO [Hz]
n_{intrpt} Number of cycles executed by the interrupt handler
f_{rep} Repetition rate of the interrupt handler [Hz]

CCR0 — repetition rate 10 kHz   15 cycles for the task, 11 cycles overhead   26 cycles
CCR1 — repetition rate 3.27 kHz   6 cycles for the task, 16 cycles overhead   22 cycles
CCR2 — repetition rate 2.0 kHz   5 cycles for the task, 16 cycles overhead   21 cycles
CCR3 — repetition rate 0.2 kHz   5 cycles for the task, 20 cycles overhead   25 cycles
CCR4 — repetition rate 3.27 kHz   17 cycles for the task, 16 cycles overhead   33 cycles
TIMOV — repetition rate 50 Hz   4 cycles for the task, 14 cycles overhead   18 cycles

$$u_{CPU} = \frac{26 \times 10^4 + 22 \times 3276.8 + 21 \times 2000 + 25 \times 200 + 33 \times 3276.8 + 18 \times 50}{3.2768 \times 10^6} = 0.15$$

The result above means a CPU loading of approximative 15% due to the Timer_A (the tasks of the timer blocks 1, 2, and 3 are not included).

6.3.8.3 DTMF Generation

Modern telephones use dual-tone multi-frequency (DTMF) signaling for the dialing process. A pair of frequencies defines each of the 16 possible numbers and characters, and are selected from the matrix shown in Table 6–16. Two Timer_A outputs (TA2 and TA1) are used to generate the frequency pair. External filters clean up the waveform and mix the two frequencies. The length of the output signals is normally 65 ms to 100 ms.

Table 6–16. DTMF Frequency Pairs

FREQUENCY	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Table 6–17 shows the errors of the generated DTMF frequencies caused by the timer clock frequency used. Rounding is used for the timer values to get the smallest possible errors.

Table 6–17. Errors of the DTMF Frequencies Caused by the MCLK

FLL MULTIPLIER N	32	64	96	116
FREQUENCY	1.048 MHz	2.096 MHz	3.144 MHz	3.801 MHz
697 Hz	+0.027%	+0.027%	+0.027%	+0.027%
770 Hz	−0.015%	−0.016%	+0.033%	−0.016%
852 Hz	+0.059%	−0.023%	+0.005%	+0.031%
941 Hz	+0.029%	+0.029%	+0.029%	+0.035%
1209 Hz	−0.079%	+0.036%	+0.036%	−0.003%
1336 Hz	+0.109%	−0.018%	+0.025%	+0.025%
1477 Hz	−0.009%	−0.009%	−0.009%	−0.009%
1633 Hz	+0.018%	+0.018%	+0.018%	+0.018%

Figure 6–23 shows a proven hardware solution to mix the two output frequencies. A low-pass filter is used for the high output frequency and another one for the low output frequency. The outputs of these low-pass filters are summed by a third operational amplifier. The filter hardware was developed by Robert Siwy/Bavaria.
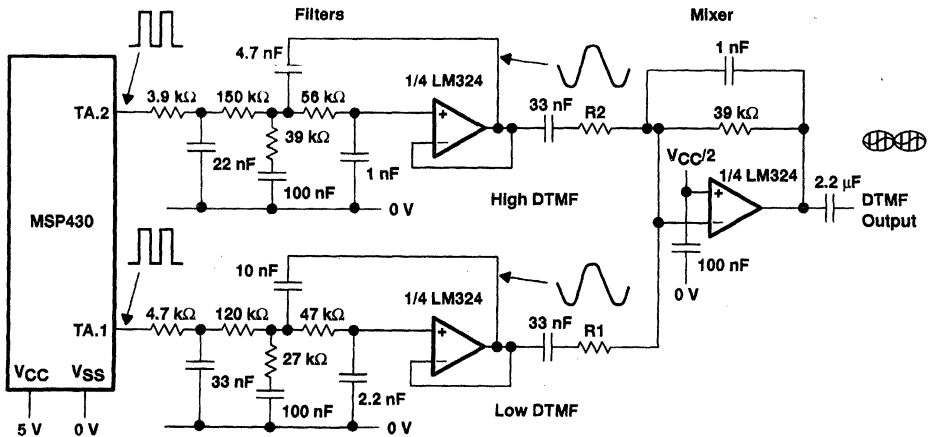
All Components are 10% Tolerance

Figure 6–23. DTMF Filters and Mixer

The two low-pass filters and the mixer are shown in figure 6–23. The symmetrical output pulses at TA2 and TA1 bias the filter amplifiers with $V_{CC}/2$.

The component values are valid for the specification of the German public telephone system. The positive supply voltage for the operational amplifiers is switched by a TP output or an O output.

With the two resistors R1 and R2, the filters can be adapted to the specifications of the telephone systems in other countries. These resistors define the high and low DTMF frequency parts of the DTMF output signal.

Example 6–31. DTMF Software

The following DTMF software routine is independent of the timer clock frequency used. During the assembly, the new timer values are calculated. The length of the DTMF output signal is defined with the value DL — its value is in milliseconds.

```
; Hardware definitions
;
FLLMPY   .equ    32              ; FLL multiplier for 1.048MHz
TCLK     .equ    FLLMPY*32768    ; TCLK: FLLMPY x fcrystal
DL       .equ    82              ; DTMF signal length (65..100ms)
STACK    .equ    600h            ; Stack initialization address
;
; RAM definitions
;
STDTMF   .equ    202h            ; Status Hi and Lo frequency
TIMAEXT  .equ    204             ; Timer Register Extension
LENGTH   .equ    206h            ; DTMF length counter
;
         .text 0F000h            ; Software start address
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT enabled
; Prepare Timer_A Output Units, MCLK = 1.048MHz (autom.)
;
INIT     MOV     #STACK,SP       ; Initialize Stack Pointer SP
         CALL    #INITSR         ; Init. FLL and RAM
         MOV     #ISMCLK+TAIE+CLR,&TACTL ; Define Timer
         MOV.B   #TA2+TA1,&P3SEL ; TA2 and TA1 at P3.5/4
```

```
            CLR      TIMAEXT          ; Clear TAR extension
            BIS      #MCONT,&TACTL    ; Start Timer_A
            EINT                      ; Enable interrupt
MAINLOOP ...                          ; Continue in mainloop
;
; A key was pressed: SDTMF contains the table offset of the
; two frequencies (0..6,0..6) in the high and low bytes
;
            MOV      &TAR,R5          ; For immediate start:
            ADD      FDTMFLO,R5       ; Short time offset
            MOV      R5,&CCR1         ; 1st change after 0.71ms
            MOV      R5,&CCR2         ; 1/(2x697) = 0.71ms
            MOV      #OMT+CCIE,&CCTL1 ; Toggle, INTRPT on
            MOV      #OMT+CCIE,&CCTL2 ; Toggle, INTRPT on
            MOV.B    STDTMF,R5        ; Counter for 82ms
            RRA      R5               ; # of low frequ. changes
            MOV.B    DTMFL(R5),LENGTH ; for the signal length.
            ...                       ; Continue background
;
; CCR0 interrupt handler (not implemented here)
;
TIMMOD0  ...
            RETI
;
; Interrupt handler for Capture/Compare Registers 1 to 4
;
TIM_HND  ADD      &TAIV,PC         ; Serve highest priority request
            RETI                      ; No interrupt pending: RETI
            JMP      HCCR1            ; CCR1 request (low DTMF frequ.)
            JMP      HCCR2            ; CCR2 request (high DTMF fr.)
            JMP      HCCR3            ; CCR3 request
            JMP      HCCR4            ; CCR4 request
;
TIMOVH   INC      TIMAEXT          ; Extension of Timer_A 32 bit
            RETI
;
```

```
; Low DTMF frequencies: TA1 is toggled by Output Unit 1
; Output changes of TA1 are counted to control signal length
;
HCCR1    PUSH    R5                     ; Save used register
         MOV.B   STDTMF,R5              ; Status low DTMF frequency
         ADD     FDTMFLO(R5),&CCR1 ; Add length of half period
         DEC.B   LENGTH                 ; Signal length DL elapsed?
         JNZ     TARET                  ; No
;
; Yes, terminate DTMF signal: disable interrupts, Output only
;
         BIC     #OMRS+OUT+CCIE,&CCTL1 ; Reset TA1
         BIC     #OMRS+OUT+CCIE,&CCTL2 ; Reset TA2
TARET    POP     R5                     ; Restore R5
         RETI                           ; Return from interrupt
;
; High DTMF frequencies: TA2 is toggled by Output Unit 2
;
HCCR2    PUSH    R5                     ; Save used register
         MOV.B   STDTMF+1,R5           ; Status high DTMF frequency
         ADD     FDTMFHI(R5),&CCR2 ; Add length of half period
         POP     R5                     ; Restore R5
         RETI                           ; Return from interrupt
;
HCCR3    ...                            ; Task controlled by CCR3
         RETI
HCCR4    ...                            ; Task controlled by CCR4
         RETI
;
; Table with the DTMF frequencies: the table contains the
; number of MCLK cycles for a half period. The values are
; adapted to the actual MCLK frequency during the assembly
; Rounding assures the smallest possible frequency error
;
FDTMFLO  .word   ((TCLK/697)+1)/2 ; Lo DTMF frequ.   697Hz
         .word   ((TCLK/770)+1)/2 ;          770Hz
```

```
            .word    ((TCLK/852)+1)/2  ;              852Hz
            .word    ((TCLK/941)+1)/2  ;              941Hz
FDTMFHI     .word    ((TCLK/1209)+1)/2 ; Hi DTMF frequ. 1209Hz
            .word    ((TCLK/1336)+1)/2 ;             1336Hz
            .word    ((TCLK/1477)+1)/2 ;             1477Hz
            .word    ((TCLK/1633)+1)/2 ;             1633Hz
;
; Table contains the number of half periods for the signal
; length DL (ms). The low DTMF frequency is used for the timing
;
DTMFL       .byte    2*697*DL/1000     ; Number of half periods
            .byte    2*770*DL/1000     ; per DL ms
            .byte    2*852*DL/1000     ;
            .byte    2*941*DL/1000     ;


            .sect    "TIMVEC",0FFF0h   ; Timer_A Interrupt Vectors
            .word    TIM_HND           ; Timer Block 1..4 Vector
            .word    TIMMOD0           ; Vector for Timer Block 0
            .sect    "INITVEC",0FFFEh  ; Reset Vector
            .word    INIT
```

Example 6–32. DTMF Software — Faster

Another software solution that is faster — but needs more RAM — is shown below. The table containing the length of the half waves is read only once for the two DTMF frequencies and the read values are stored in RAM words DTMFLO and DTMFHI. The Timer_A interrupt routines use these two values. The tables are the same as with the example above.

```
FLLMPY   .equ    32              ; FLL multiplier for 1.048MHz
TCLK     .equ    FLLMPY*32768    ; TCLK: FLLMPY x fcrystal
DL       .equ    82              ; DTMF time ms (65..100ms)
STDTMF   .equ    202h            ; Status Hi and Lo frequency
TIMAEXT  .equ    204             ; Timer Register Extension
LENGTH   .equ    206h            ; DTMF length counter
DTMFLO   .equ    208h            ; Half wave of low frequency
DTMFHI   .equ    20Ah            ; Half wave of high frequency
```

```
STACK    .equ    600h            ; Stack initialization address
         .text 0F000h            ; Software start address
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT enabled
; Prepare Timer_A Output Units, MCLK = 1.048MHz (autom.)
;
INIT     MOV     #STACK,SP       ; Initialize Stack Pointer SP
         CALL    #INITSR         ; Init. FLL and RAM
         MOV     #ISMCLK+TAIE+CLR,&TACTL ; Start Timer
         MOV.B   #TA2+TA1,&P3SEL ; TA2 and TA1 at P3.5/4
         CLR     TIMAEXT         ; Clear TAR extension
         BIS     #MCONT,&TACTL   ; Start Timer_A
         EINT                    ; Enable interrupt
MAINLOOP ...                     ; Continue in mainloop
;
; A key was pressed: STDTMF contains the table offset of the
; two frequencies (0..6,0..6) in the high and low bytes
;
         MOV     &TAR,R5         ; For immediate start:
         ADD     FDTMFLO,R5      ; Short time offset
         MOV     R5,&CCR1        ; 1st change after 0.71ms
         MOV     R5,&CCR2        ; 1/(2x697) = 0.71ms
;
; Fetch the two cycle counts for the DTMF frequencies
;
         MOV.B   STDTMF+1,R5     ; High DTMF frequency
         MOV     FDTMFHI(R5),DTMFHI ; Length of half period
         MOV.B   STDTMF,R5       ; Low DTMF frequency
         MOV     FDTMFLO(R5),DTMFLO ; Length of half period
;
                                 ; Counter for length
         RRA     R5              ; Prepare byte index
         MOV.B   DTMFL(R5),LENGTH ; # of low frequ. changes
         MOV     #OMT+CCIE,&CCTL1 ; Toggle, INTRPT on
         MOV     #OMT+CCIE,&CCTL2 ; Toggle, INTRPT on
         ...                                              ; to
Mainloop
```

```
;
; CCR0 interrupt handler (not implemented here)
;
TIMMOD0   ...
          RETI
;
; Interrupt handler for Capture/Compare Registers 1 to 4
;
TIM_HND   ADD      &TAIV,PC        ; Serve highest priority request
          RETI                     ; No interrupt pending: RETI
          JMP      HCCR1           ; CCR1 request (low DTMF frequ.)
          JMP      HCCR2           ; CCR2 request (high DTMF fr.)
          JMP      HCCR3           ; CCR3 request
          JMP      HCCR4           ; CCR4 request
;
TIMOVH    INC      TIMAEXT         ; Extension of Timer_A 32 bit
          RETI
;
; Low DTMF frequencies: TA1 is toggled by Output Unit 1
;
HCCR1     ADD      DTMFLO,&CCR1    ; Add length of half period
          DEC.B    LENGTH          ; DL ms elapsed?
          JNZ      TARET           ; No
;
; Terminate DTMF output: disable interrupts, Output only
;
          BIC      #OMRS+OUT+CCIE,&CCTL1 ; Reset TA1
          BIC      #OMRS+OUT+CCIE,&CCTL2 ; Reset TA2
TARET     RETI                     ; Return from interrupt
;
; High DTMF frequencies: TA2 is toggled by Output Unit 2
;
HCCR2     ADD      DTMFHI,&CCR2    ; Add length of half period
          RETI                     ; Return from interrupt
;
HCCR3     ...                      ; Task controlled by CCR3
```

```
        RETI
HCCR4   ...                          ; Task controlled by CCR4
        RETI
;
; Tables and interrupt vectors are identical to the previous
; example
```

The second example, with maximum frequencies on both channels, results in a maximum CPU loading, u_{CPU} (ranging from 0 to 1), by the Timer_A activities due to DTMF generation:

$$u_{CPU} = \frac{1}{f_{MCLK}} \; \Sigma \; (n_{intrpt} \times f_{rep})$$

Where:
f_{MCLK}	Frequency of the system clock oscilator (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

CCR1 — repetition rate 2×941 Hz	12 cycles for the task, 16 cycles overhead	28 cycles
CCR2 — repetition rate 2×1633 Hz	6 cycles for the task, 16 cycles overhead	22 cycles

$$u_{CPU} = \frac{28 \times 2 \times 941 + 22 \times 2 \times 1633}{1.048 \times 10^6} = 0.12$$

This result shows a worst case CPU loading of approximate 12% due to the DTMF generation. This loading occurs only during the 82 ms activity.

6.3.8.4 TRIAC Control

TRIAC control for electric motors (DMC) or other loads is a simple task when using the Timer_A. The software loads one of the capture/compare registers (CCR4 with this example), prepares the output unit to change the TAx output after the desired time, and continues with the background task. When the loaded time interval elapses, the output unit fires the TRIAC gate at exactly the programmed time and requests an interrupt. The interrupt handler can use dynamic control (several short pulses to save current) or static control (one long gate pulse), which is used with this example. See figure 6–25 for details.

The TRIAC control software contains some security features. They ensure that no gate triggering of the previous half wave can last into the next half wave and cause gate triggering there also:

- The zero crossing part (P0.0 handler) immediately switches off the gate signal by setting of the TA4 terminal to high

- The P0.0 handler calculates a value, OFFTIME, that defines a time for the actual half wave where the gate signal must be switched off at the latest

- The timer block 4 handler checks before each switch-on of the TRIAC gate to see if the on-time of the gate exceeds the calculated value, OFFTIME, or not. If the value in OFFTIME is exceeded, then it is used for the maximum on time

The TRIAC control software is independent of the ac line frequency. For each full wave of the line voltage, the period is measured and used for the security features. The calculation software also uses the timer clocks value of the half-period stored in RAM location MAINHW.

Figure 6–24 shows the hardware for the TRIAC control in this example. The temperature measurement, the overcurrent detection, and the revolution control are not included in the software example.

After power up, the TA4 terminal is switched to input mode. The base resistor of the PNP transistor switches the gate of the TRIAC off and prevents the motor from running.
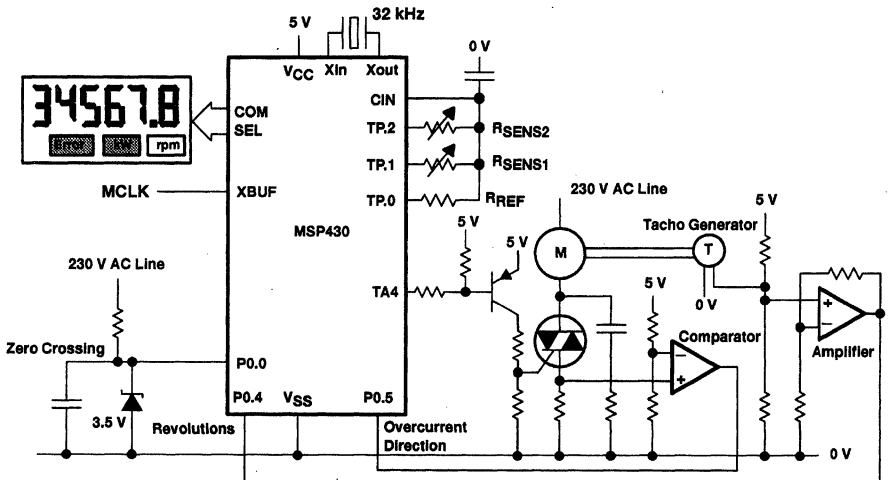
Figure 6–24. TRIAC Control With Timer_A

Figure 6–25 shows a TRIAC control with three different conduction angles. Dynamic control and static control is included.

The software example is written for the static control only, but it is relatively easy to add additional states to the TRIAC handler (timer block 4), which means more than one gate pulse per half wave.
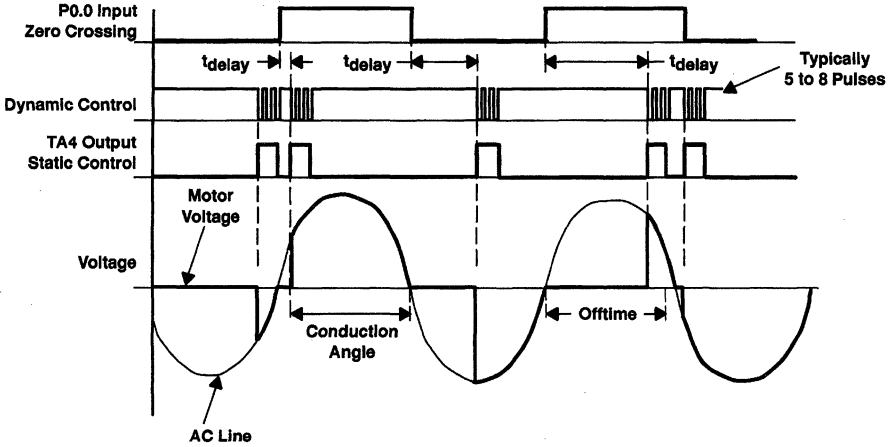
Figure 6–25. Static and Dynamic TRIAC Gate Control

The software shown below works up to a timer clock frequency (fCLK) of MCLK (in this case due to k = 1):

$$f_{CLK} < 2^{16} \times k \times 2 \times f_{LINE}$$

Where:

f_{CLK}	Input frequency at the input divider input of Timer_A	[Hz]
k	Pre-divider constant of the input divider (1, 2, 4 or 8)	
f_{LINE}	AC line frequency used	[Hz]

If fCLK is higher than defined above, then the input divider of Timer_A must be used. This restriction is caused by the 16-bit structure of Timer_A and the RAM.

Example 6–33. Triac Control

The check to see if the gate pulse starts after the security time, SEC, is not included below. It must occur during the calculation.

```
; Definitions for the TRIAC control software
;
FLLMPY  .equ    32                 ; FLL multiplier for 1.048MHz
TCLK    .equ    FLLMPY*32768       ; TCLK (Timer Clock) [Hz]
SEC     .equ    (500*TCLK/1000)/1000 ; Security time (500us)
Gate_On .equ    (1200*TCLK/1000)/1000 ; TRIAC Gate on (1200us)
;
; RAM definitions
;
TIMAEXT .equ    202h               ; Timer Register Extension
OFFTIME .equ    204h               ; Time when gate MUST be off
MAINHW  .equ    206h               ; Length of half wave (TCLK)
PRVTAR  .equ    208h               ; Value of TAR at last pos. edge
FIRANGL .equ    20Ah               ; Half wave - conduction angle
STTRIAC .equ    20Ch               ; Control byte (0 = off)
STACK   .equ    600h               ; Stack initialization address
        .text                      ; Start of ROM code
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT enabled
; Prepare Timer_A Output Units
;
INIT    MOV     #STACK,SP          ; Initialize Stack Pointer SP
        CALL    #INITSR            ; Init. FLL and RAM
        MOV     #ISMCLK+TAIE+CLR,&TACTL ; Init. Timer
        MOV     #OMOO+CCIE+OUT,&CCTL4 ; Set TA4 high
        BIS.B   #TA4,&P3SEL        ; TA4 controls gate transistor
        BIS.B   #P0IE0,&IE1        ; Enable P0.0 interrupt
        CLR     TIMAEXT            ; Clear TAR extension
        CLR.B   STTRIAC            ; TRIAC off status (0)
        BIS     #MCONT,&TACTL      ; Start Timer_A in Cont. Mode
        MOV.B   #CBMCLK+CBE,&CBCTL ; MCLK at XBUF pin
        EINT                       ; Enable interrupt
```

```
MAINLOOP  ...                    ; Continue in mainloop
;
; Some control examples:
; Start electric motor: checked result (TCLK cycles) in R5.
; The result is the time difference from the zero crossing
; to the first gate pulse measured in Timer Clock cycles
;
        MOV     R5,FIRANGL       ; Gate delay to FIRANGL
        MOV.B   #1,STTRIAC       ; Activate TRIAC control
        ...                      ; Continue in background
;
; The motor is running. A new calculation result is available
; in R5. It will be used with the next mains half wave
;
        MOV     R5,FIRANGL       ; Gate delay to FIRANGL
        ...                      ; Continue in background
;
; Stop motor: switch off TRIAC control
;
        CLR.B   STTRIAC          ; Disable TRIAC control
        MOV     #OMOO+CCIE+OUT,&CCTL4 ; TRIAC gate off
        ...                      ; Continue with background
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset when reading
; the Timer Vector Register TAIV
;
TIM_HND EINT                     ; Real time environment
        ADD     &TAIV,PC         ; Add "Jump table" offset
        RETI                     ; Vector 0: No interrupt
        JMP     TIMMOD1          ; Vector 2: Block 1
        JMP     TIMMOD2          ; Vector 4: Block 2
        JMP     TIMMOD3          ; Vector 6: Block 3
        JMP     TIMMOD4          ; Vector 8: Block 4
;
; Block 5. Timer Overflow Handler: the Timer Register is
```

```
; expanded into the RAM location TIMAEXT (16 MSBs)
;
TIMOVH    .EQU    $                ; Vector 10: TIMOV Flag
          INC     TIMAEXT          ; Incr. Timer extension
          JMP     TIM_HND          ; Another Timer_A interrupt?
;
; The interrupt handlers for the Timer Blocks 0 to 3 follow
; They are not implemented here
;
TIMMOD0   .equ    $                ; Handler for Timer Block 0
TIMMOD1   .equ    $                ; Handler for Timer Block 1
TIMMOD2   .equ    $                ; Handler for Timer Block 2
TIMMOD3   .equ    $                ; Handler for Timer Block 3
          RETI
;
; Timer Block 4: interrupt handler for the TRIAC control
;
TIMMOD4   PUSH    R5               ; Save help register R5
          MOV.B   STTRIAC,R5       ; Status of TRIAC control
          MOV.B   CC4TAB(R5),R5    ; Fetch offset to status handler
          ADD     R5,PC            ; Branch to status handler
CC4TAB    .byte   STATE0-CC4TAB    ; Status 0: No TRIAC activity
          .byte   STATE0-CC4TAB    ; Status 1: activition made
          .byte   STATE2-CC4TAB    ; Status 2: 1st gate pulse
          .byte   STATE3-CC4TAB    ; Status 3: TRIAC gate off
          .even
;
; TRIAC status 2: gate is switched on for "Gate_ON" time
; The On time is shortened to the OFFTIME value if the
; OFFTIME is before the Gate_On time
;
STATE2    MOV     &CCR4,R5         ; Copy time of interrupt
          ADD     #Gate_On,&CCR4   ; Set end of ON state
          INV     R5               ; Negate last INTRPT time
          INC     R5
          ADD     OFFTIME,R5       ; OFFTIME - last INTRPT time
```

```
          CMP      #Gate_On,R5        ; OFFTIME later than next INTRPT?
          JHS      ST20               ; Yes, ok
;
; The calculated ON time ends after OFFTIME: OFFTIME is used
;
          MOV      OFFTIME,&CCR4
ST20      MOV      #OMSET+CCIE,&CCTL4 ; Prepare for gate off
          INC.B    STTRIAC            ; TRIAC status + 1
;
; TRIAC status 0: No activity. TRIAC is off always
;
STATE0    POP      R5                 ; Restore help register
          RETI                        ; Return from interrupt
;
; TRIAC status 3: gate pulse  is output.
; No activity until next half wave.
STATE3    MOV      #OMOO+CCIE+OUT,&CCTL4 ; Gate off (TA4 high)
          MOV.B    #1,STTRIAC         ; TRIAC status: wait for 0-cross.
          JMP      STATE0
;
; P0.0 Handler: the mains voltage causes interrupt with each
; zero crossing. The TRIAC gate is switched off first, to
; avoid the ignition of the coming half wave. Hardware debounce
; is necessary for the mains signal! See schematic
;
P00_HNDLR MOV      #OMOO+CCIE+OUT,&CCTL4 ; Switch off TRIAC
          PUSH     R5                 ; Save used register
          XOR.B    #1,&P0IES          ; Change interrupt edge of P0.0
          MOV      &TAR,R5            ; 0-crossing time to R5
;
; The shorter positive halfwave is measured (TCLK cycles)
;
          BIT.B    #1,&P0IN           ; Positive edge of mains?
          JZ       P01                ; No,
          MOV      R5,PRVTAR          ; Yes, for next HW calculation
          JMP      P03                ; Save time of 0-crossing
```

```
P01     MOV     R5,MAINHW       ; Measure pos. mains half wave
        SUB     PRVTAR,MAINHW   ; Difference is length of pos. HW
;
; If STTRIAC is not 0 ( 0 = inactivity) then the next gate
; firing is prepared
;
P03     TST.B   STTRIAC         ;
        JZ      P02             ; STTRIAC = 0: no activity
        MOV.B   #2,STTRIAC      ; STTRIAC > 0: prep. next firing
;
; The TRIAC firing time is calculated: Timer Reg. + FIRANGL
;
        MOV     R5,&CCR4        ; TAR to CCR4
        ADD     FIRANGL,&CCR4   ; TAR + delay -> CCR4
        MOV     #OMR+CCIE+OUT,&CCTL4 ; TA4 is reset by INTRPT
;
; The worst case switch-off time for the TRIAC is calculated:
; Zero crossing time + half period - security time
; This calculation ensures a safe distance to the next zero
; crossing of the mains
;
        ADD     MAINHW,R5       ; TAR + MAINHW
        SUB     #SEC,R5         ; Subtract security time
        MOV     R5,OFFTIME      ; worst case switch-off time
P02     POP     R5              ; Restore R5
        RETI
        .sect   "TIMVEC",0FFF0h ; Timer_A Interrupt Vectors
        .word   TIM_HND         ; Timer Blocks 1..4 Vector
        .word   TIMMOD0         ; Vector for Timer Block 0
        .sect   "P00VEC",0FFFAh ; P0.0 Vector
        .word   P00_HNDLR
        .sect   "INITVEC",0FFFEh ; Reset Vector
        .word   INIT
```

The TRIAC control example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1):

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma\,(n_{intrpt} \times f_{rep})$$

Where:

f_{MCLK} Frequency of the system clock generator (DCO)   [Hz]

n_{intrpt} Number of cycles executed by the interrupt handler

f_{rep} Repetition rate of the interrupt handler   [Hz]

CCR4 — repetition rate 100Hz	79 cycles for the task, 16 cycles overhead	95 cycles
P0.0 — repetition rate 100Hz	60 cycles for the task, 11 cycles overhead	71 cycles

$$u_{CPU} = \frac{100 \times 95 + 100 \times 71}{1.048 \times 10^6} = 0.016$$

This shows a CPU loading of approximately 1.6% due to the static TRIAC control.

6.3.8.5 Mixture of Capture and Compare Modes

Any mix of capture and compare mode is possible with the Timer_A. The following software example shows two timer blocks using the capture mode and three timer blocks using the compare mode. For formulas, see Section 6.3.8.2.

❑ **Capture/Compare Block 0** — a short negative pulse with a 1 kHz repetition rate is generated and output at the terminal TA0. The pulse is reset to high by the interrupt handler of timer block 0. The pulse is used for the precise triggering of an external peripheral. The error of the repetition rate due to the MCLK frequency used is –0.055%.

❑ **Capture/Compare Block 1** — the period of the input signal at the CCI1A input terminal is measured in timer clock cycles. The period is measured from leading edge to leading edge of the input signal. The last measured value is stored in the RAM word PERIOD. The maximum period length that can be measured this way is $k \times 2^{16}/f_{CLK}$.

❑ **Capture/Compare Block 2** — a square wave with a variable repetition rate is generated and output at the terminal TA2. The actual cycle count for one half-wave is stored in the RAM word TIM2REP.

❑ **Capture/Compare Block 3** — the event time of the trailing edge of the input signal at the CCI3A input terminal is captured. The last captured value is stored in the RAM word STOR3.

❑ **Capture/Compare Block 4** — a square wave with a variable output frequency is generated and output at the TA4 terminal. The output frequency starts at 4 kHz and decreases to 1 kHz. The square wave is used for the control of an external peripheral.

The software routine is independent of the MCLK frequency used. Only the FLL multiplier constant, FLLMPY, needs to be redefined if another MCLK frequency is selected.

Table 6–18. Short Description of the Capture and Compare Mix

TIMER BLOCK	TIME INTERVAL	OUTPUT UNIT	COMMENT
0	1 ms	Outputs frequency	Negative pulses: 1 kHz
1	External	Not used	Measures period of signal at input CCI1A, leading edge to leading edge. Minimum signal length: 2 ms
2	Variable	Outputs frequency	Length of a half-period stored in TIM2REP. (2 kHz max)
3	External .	Not used	Captures event time of the trailing edge of the signal input at CCI3A. Maximum signal = 500 Hz
4	250 μs to 1 ms	Outputs frequency	Decreasing frequency from 4 kHz to 1 kHz

The maximum frequencies and minimum signal length shown do not indicate the limits of the Timer_A. They are given for the calculation of the loading of the CPU only.

Figure 6–26 illustrates the above described five tasks:
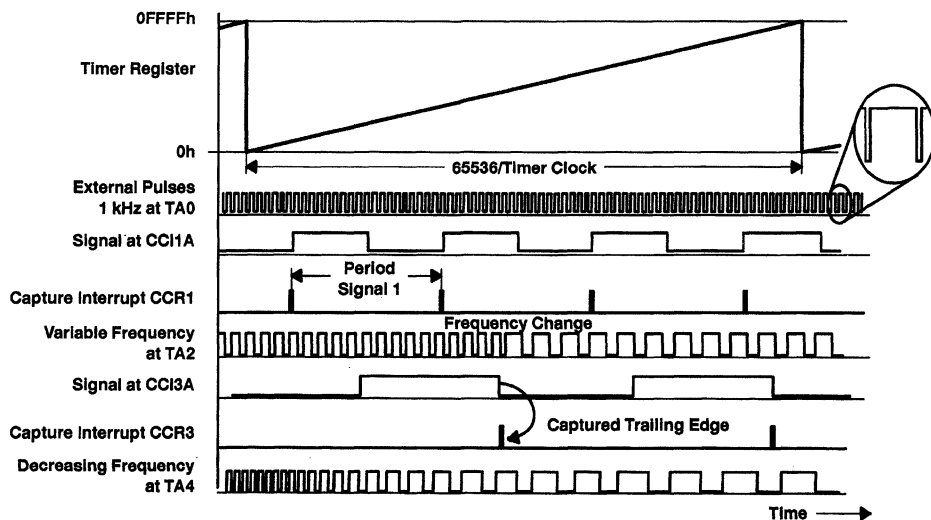
Figure 6–26. Mixture of Capture Mode and Compare Mode With the Continuous Mode

The software example also shows how to output the ACLK frequency at output terminal XBUF for reference purposes. An external device may be driven by this stable and precise crystal-controlled frequency.

A special method is used for the return from interrupt. The interrupt handlers of the five timer blocks do not return normally with a RETI instruction but jump back to the start of the timer handler for a test to see if another Timer_A interrupt is pending. This makes it necessary to enable the interrupt at the start of the timer handler. Otherwise, the interrupt latency time will get too long for other interrupts.

Example 6–34. Mixed Capture and Compare Modes

```
; Software example: three independent timings and two inputs
; with capturing. The Continuous Mode of Timer_A is used
;
FLLMPY   .equ      64                ; FLL multiplier for 2.096MHz
TCLK     .equ      FLLMPY*32768      ; TCLK: FLLMPY x fcrystal
OLDRE    .equ      202h              ; Time of last edge at CCI1A
PERIOD   .equ      204h              ; Calc. period of CCI1A event
TIM2REP  .equ      206h              ; Repetition rate Block 2
STOR3    .equ      208h              ; Last neg. edge at CCI3A
TIM4REP  .equ      20Ah              ; Repetition rate Block 4
TIMAEXT  .equ      20Ch              ; Extension for Timer Register
STACK    .equ      600h              ; Stack initialization address
         .text 0F000h               ; Software start address
INIT     MOV       #STACK,SP         ; Initialize Stack Pointer
         CALL      #INITSR           ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT on
; Inputs (CCIxA) and outputs (Tax) of Timer_A are defined
;
         MOV       #ISMCLK+TAIE+CLR,&TACTL
         MOV       #OMR+CCIE,&CCTL0  ; Reset Mode, INTRPT on
         MOV       #CMPE+ISCCIA+SCS+CAP+CCIE,&CCTL1 ;
         MOV       #OMT+CCIE,&CCTL2  ; Toggle, INTRPT on
         MOV       #CMNE+ISCCIA+SCS+CAP+CCIE,&CCTL3 ;
         MOV       #OMT+CCIE,&CCTL4  ; Toggle, INTRPT on
         MOV       #0FFFFh,TIM2REP   ; Start value Block 2
```

```
          MOV       #((TCLK/4000)+1)/2,TIM4REP ; 4kHz start frequ.
          MOV.B     #TA4+TA3+TA2+TA1+TA0,&P3SEL ; Define I/Os
;
          MOV       #1,&CCR0          ; Immediate start
          MOV       #1,&CCR2          ; for the Capture/Compare
          MOV       #1,&CCR4          ; Registers
          CLR       TIMAEXT           ; Clear TAR extension
          MOV.B     #CBACLK+CBE,&CBCTL ; Output ACLK at XBUF pin
          BIS       #MCONT,&TACTL     ; Start Timer
          EINT                        ; Enable interrupt
MAINLOOP ...                          ; Continue in background
; Interrupt handler for Capture/Compare Block 0. An ext.
; peripheral is started every 1ms with a negative pulse at
; TA0 (set exactly in time by Output Unit 0). The handler
; resets the negative signal.
;
TIMMOD0   .EQU      $                 ; Start of handler
          ADD       #((2*TCLK/1000)+1)/2,&CCR0 ; For next INTRPT
          MOV       #OMOO+CCIE+OUT,&CCTL0; Set TA0: pulse off
          BIS       #OMR,&CCTL0       ; Back to Reset Mode
                                      ; Fall through to TIM_HND
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
TIM_HND   .EQU      $                 ; Start of Timer_A handler
          EINT                        ; Allow interrupt nesting
TH0       ADD       &TAIV,PC          ; Add Jump table offset
          RETI                        ; Vector 0: No interrupt
          JMP       TIMMOD1           ; Vector 2: Block 1
          JMP       TIMMOD2           ; Vector 4: Block 2
          JMP       TIMMOD3           ; Vector 6: Block 3
          JMP       TIMMOD4           ; Vector 8: Block 4
;
; Block 5. Timer Overflow Handler: the Timer Register is
```

```
; expanded into the RAM location TIMEXT (MSBs)
;
TIMOVH   .EQU    $               ; Vector 10: TIMOV Flag
         INC     TIMAEXT         ; Incr. Timer extension
         JMP     THO             ; Test for other interrupts
;
; Timer Block 1 measures the period of an input signal at
; pin CCI1A. The interval between two rising edges is measured
;
TIMMOD1  .EQU    $               ; Vector 2: Block 1
         MOV     &CCR1,PERIOD    ; Time of captured rising edge
         SUB     OLDRE,PERIOD    ; Calculate period (difference)
         MOV     &CCR1,OLDRE     ; Store actual edge time
         JMP     THO             ; Test for another interrupts
;
; The used time interval delta t2 is stored in TIM2REP.
;
TIMMOD2  .EQU    $               ; Vector 4: Block 2
         ADD     TIM2REP,&CCR2   ; Add time interval
         ...                     ; Task2 starts here
         JMP     THO             ; Test for another interrupts
;
; Timer Block 3stores the time for a trailing edge at CCI3A
; STOR3 contains the time of the latest trailing edge
;
TIMMOD3  .EQU    $               ; Vector 6: Block 3
         MOV     &CCR3,STOR3     ; Store event time
         JMP     THO             ; Test for another interrupts
;
; Block 4 uses a variable repetition rate starting at 4kHz
; cycles and going down to 1kHz.
;
TIMMOD4  .EQU    $               ; Vector 8: Block 4
         ADD     TIM4REP,&CCR4   ; Add time interval
         CMP     #((TCLK/1000)+1)/2,TIM4REP ; Final value?
         JHS     THO             ; Yes, no modification
```

```
        INC       TIM4REP           ; No, modify interval
        JMP       TH0               ; Test for other interrupts
;
        .sect     "TIMVEC",0FFF0h   ; Timer_A Interrupt Vectors
        .word     TIM_HND           ; Timer Blocks 1..4 Vector
        .word     TIMMOD0           ; Vector for Timer Block 0
        .sect     "INITVEC",0FFFEh  ; Reset Vector
        .word     INIT
```

The software example above results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:

f_{MCLK}	Frequency of the system clock generator (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

CCR0 — repetition rate 1 kHz	15 cycles for the task, 15 cycles overhead	30 cycles
CCR1— rep. rate max. 0.5 kHz	18 cycles for the task, 22 cycles overhead	40 cycles
CCR2 — repetition rate 2 kHz	6 cycles for the task, 22 cycles overhead	28 cycles
CCR3 — repetition rate 0.5 kHz	6 cycles for the task, 22 cycles overhead	28 cycles
CCR4 — repetition rate 8 kHz	17 cycles for the task, 22 cycles overhead	39 cycles
TIMOV — rep. rate 32 Hz@2 MHz	4 cycles for the task, 20 cycles overhead	24 cycles

$$u_{CPU} = \frac{30 \times 10^3 + 40 \times 500 + 28 \times 2000 + 28 \times 500 + 39 \times 8000 + 24 \times 32}{2.096 \times 10^6} = 0.21$$

This shows a worst case CPU loading of approximate 21% due to the Timer_A (the task of the timer block 2 is not included). If f_{MCLK} is chosen to be 3.8 MHz, then the CPU loading is only 11.5%, max.

Any pending Timer_A interrupt during the return phase saves 6 cycles because of the code in this example.

6.3.8.6 Applications Exceeding the 16-Bit Range of the Timer_A

If the periods of the internal interrupt timings or the time intervals to be captured are longer than one period of the timer register, then a special method is necessary to take care of the larger time periods. The same is true if a half period of a generated output frequency is larger than the period of the Timer_A.

This special method, using extension registers for the capture/compare registers is necessary if:

$$t_{SIGNAL} > \frac{2^{16} \times k}{f_{CLK}}$$

Where:

t_{SIGNAL}	Time interval to be measured or generated	[Hz]
f_{CLK}	Input frequency at the input divider input of Timer_A	[Hz]
k	Predivider constant of the input divider (1, 2, 4 or 8)	

Figure 6–27 illustrates the hardware and RAM registers used with the *compare mode* if the compared values exceed the range of 16 bits (values are greater than 65535):
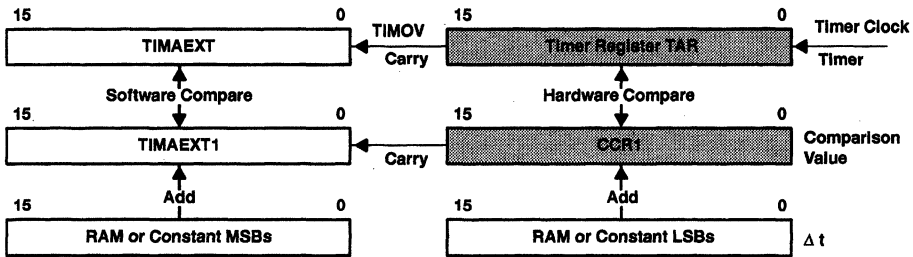
Figure 6–27. Compare Mode with Timer Values Greater than 16 Bit (shown for CCR1)

Figure 6–28 illustrates the hardware and RAM registers used with the *capture mode* if the captured values exceed the range of 16 bits (values are greater than 65535):
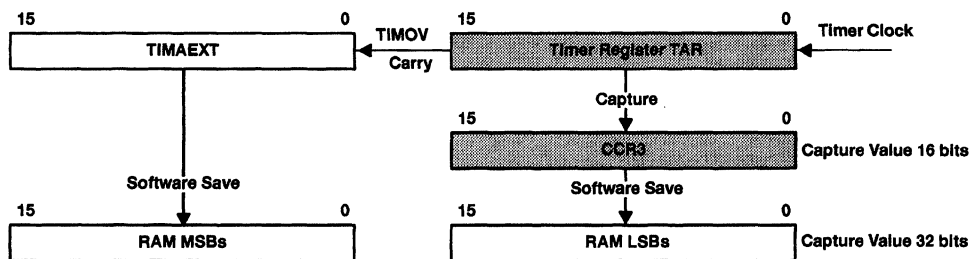
Figure 6–28. Capture Mode With Timer Values Greater than 16 Bit (shown for CCR3)

Figure 6–29 illustrates five examples. The tasks are defined as follows:

❑ **Capture/Compare Block 0** — a symmetric 1 kHz signal is generated and output at terminal TA0. It is used for the control of external peripherals (e.g. ADCs).

❑ **Capture/Compare Block 1** — an internal interrupt with a period $\Delta t1 = 1s$ (considerably longer than the timer register period) is generated.

❑ **Capture/Compare Block 2** — the length, $\Delta t2$, of the high part of the input signal at the CCI2A input terminal is measured and stored in the RAM words PP2MSB and PP2LSB. The captured time of the leading edge is stored in the RAM words TIM2MSB and TIM2LSB.

❑ **Capture/Compare Block 3** — the event time of the leading edge of the signal at the CCI3A input pin is captured. The captured value is stored in the RAM words TIM3MSB and TIM3LSB.

❑ **Capture/Compare Block 4** — A symmetrical, external signal is output at terminal TA4. The time interval, $\Delta t4$, between two output signal edges is defined in TIM4MSB and TIM4LSB.

The RAM extension of the timer register TIMAEXT is used for all applications exceeding the 16-bit range of the Timer_A. Due to the low priority of the TIMOV interrupt, however, checks are necessary in the application software to see if the RAM extension is updated yet or not.

The software routine is independent of the MCLK frequency used. Only the FLL multiplier constant, FLLMPY, needs to be redefined if another MCLK frequency is selected. For the example, 3.801 MHz is used.

The task of capture/compare block 0 shows that tasks extending the 16-bit range of the Timer_A may be mixed with normal tasks that fit into the 16-bit range.

Table 6–19. Short Description of the Capture and Compare Mix

TIMER BLOCK	TIME INTERVAL	OUTPUT UNIT	COMMENT
0	1 ms	Outputs frequency	Pulses: 1 kHz @ 3.801 MHz
1	1 s	Not used	Generation of an internal reference frequency: 1s for time and date
2	External event	Input pin CCI2A is used	Measures high signal part.Stored in PP2MSB and PP2LSB
3	External event	Input pin CCI3A is used	Captures event time of the leading edge of the input signal — stored in TIM3 MSB and TIM3 LSB
4	Variable	Outputs frequency	Symmetric output signal — half period is defined by TIM4 MSB and TIM4 LSB

Figure 6–29 illustrates the four tasks described above (not to scale):
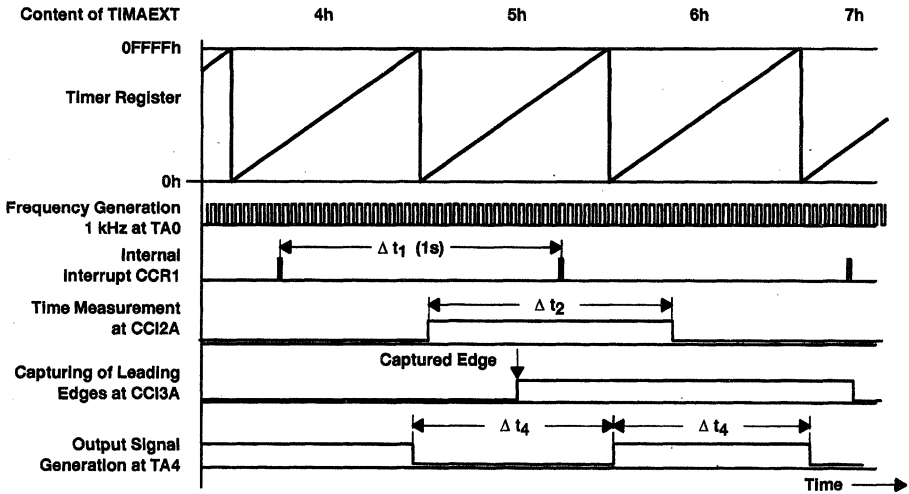
Figure 6–29. Five Different Timings Extending the Normal Timer_A Range

Example 6–35. Extending the Normal Timer_A Range

The assembler definitions for T1 (V1sMSB and V1sLSB) show a way to define times exceeding the range of one word.

```
FLLMPY    .equ    116                    ; 3.801MHz
TCLK      .equ    FLLMPY*32768           ; TCLK: FLLMPY x fcrystal
T1        .equ    1                      ; T1 is 1 second
;
V1sMSB    .equ    T1*FLLMPY*32768/65536 ; MSBs of 1s value
V1sLSB .equ (T1*FLLMPY*32768)-((T1*FLLMPY*32768/65536)*65536)
TIM2MSB   .equ    202h                   ; Time of leading edge at CCI2A
TIM2LSB   .equ    204h                   ;
PP2MSB    .equ    206h                   ; Length of high signal at CCI2A
PP2LSB    .equ    208h                   ;
TIM3MSB   .equ    20Ah                   ; Time of leading edge at CCI3A
TIM3LSB   .equ    20Ch                   ;
TIM4MSB   .equ    20Eh                   ; Time interval between TA4 edges
TIM4LSB   .equ    210h                   ;
TIMAEXT   .equ    212h                   ; Extension for Timer Register
TIMAEXT1 .equ     214h                   ; Extension for Timer Block 1
TIMAEXT4 .equ     216h                   ; Extension for Timer Block 4
STACK     .equ    600h                   ; Stack initialization address
          .text 0F000h                   ; Software start address
INIT      MOV     #STACK,SP              ; Initialize Stack Pointer
          CALL    #INITSR               ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT on
;
          MOV     #ISMCLK+TAIE+CLR,&TACTL
          MOV     #OMT+CCIE,&CCTL0  ; Toggle Mode, INTRPT on
          MOV     #OMOO+CCIE,&CCTL1 ; No output, INTRPT on
          MOV     #CMBE+ISCCIA+SCS+CAP+CCIE,&CCTL2 ; Both edges
          MOV     #CMPE+ISCCIA+SCS+CAP+CCIE,&CCTL3 ; + edge
          MOV     #OMT+CCIE,&CCTL4  ; Toggle Mode, INTRPT on
          MOV.B   #TA4+TA3+TA2+TA0,&P3SEL ; Define timer I/Os
;
```

```
        MOV     #1,&CCR0            ; Immediate start for TA0
    ⸴   CLR     TIMAEXT            ; Clear Timer Register extension
        MOV     #V1sLSB,&CCR1       ; Next INTRPT time block 1
        MOV     #V1sMSB,TIMAEXT1
        MOV     TIM4LSB,&CCR4       ; Next INTRPT time block 4
        MOV     TIM4MSB,TIMAEXT4
;
        MOV.B   #CBMCLK+CBE,&CBCTL ; Output MCLK at XBUF pin
        BIS     #MCONT,&TACTL      ; Start Timer
        EINT                       ; Enable interrupt
MAINLOOP ...                       ; Continue in background
; Interrupt handler for Capture/Compare Block 0. An external
; peripheral is started every 1ms with the negative edge of
; TA0 (set exactly from Output Unit 0).
;
TIMMOD0  .EQU   $                  ; Start of handler
         ADD    #((TCLK/1000)+1)/2,&CCR0 ; For next INTRPT
         RETI
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
TIM_HND  .EQU   $                  ; Start of Timer_A handler
         ADD    &TAIV,PC           ; Add Jump table offset
         RETI                      ; Vector 0: No interrupt
         JMP    TIMMOD1            ; Vector 2: Block 1
         JMP    TIMMOD2            ; Vector 4: Block 2
         JMP    TIMMOD3            ; Vector 6: Block 3
         JMP    TIMMOD4            ; Vector 8: Block 4
;
; Block 5. Timer Overflow Handler: the Timer Register is
; expanded into the RAM location TIMAEXT (MSBs 16 to 31)
;
TIMOVH   .EQU   $                  ; Vector 10: TIMOV Flag
         INC    TIMAEXT            ; Incr. Timer extension
```

```
        RETI                    ;
;
; Timer Block 1 gen. the 1s reference used for date and time
;
TIMMOD1  .EQU    $               ; Vector 2: Block 1
         BIT     #TAIFG,&TACTL   ; TIMOV pending?
         JNZ     TM11            ; Yes, checks necessary
TM12     CMP     TIMAEXT,TIMAEXT1 ; MSBs also equal?
         JEQ     T13             ; Yes
         RETI
;
TM11     TST     &CCR1           ; TAIFG = 1: check CCR1
         JN      TM12            ; CCR1 > 7FFFh: correct values
         PUSH    TIMAEXT         ; TIMAEXT not yet updated
         INC     0(SP)           ; Updated value of TIMAEXT
         CMP     @SP+,TIMAEXT1   ; MSBs equal?
         JNE     TM1R            ; No, return
T13      ADD     #V1sLSB,&CCR1   ; Yes, prepare next INTRPT (1s)
         ADDC    #V1sMSB,TIMAEXT1 ; MSBs of 1 second
         CALL    #RTCLK          ; Increment time by 1s
         JNC     TM1R            ; if C = 1: incr. date
         CALL    #DATE           ; 00.00 o'clock: next day
TM1R     RETI
;
; Capture Mode: the high part of the CCI2A input signal is
; measured. The result is stored in PP2MSB and PP2LSB.
;
TIMMOD2  .EQU    $               ; Vector 4: Block 2
         BIT     #CCI,&CCTL2     ; Input signal high?
         JZ      TM21            ; No,calculation necessary
         MOV     &CCR2,TIM2LSB   ; Store LSBs of capt. time
         MOV     TIMAEXT,TIM2MSB ; MSBs of capt. time
         BIT     #TAIFG,&TACTL   ; TIMOV pending?
         JZ      TM2RET          ; No, values are correct
         TST     &CCR2           ; Yes, check CCR2
         JN      TM2RET          ; CCR2 > 7FFFh: correct values
```

```
          INC      TIM2MSB            ; MSBs not yet updated
TM2RET    RETI
;                                     ; High part is calculated
TM21      MOV      &CCR2,PP2LSB       ; Store LSBs of capt. time
          MOV      TIMAEXT,PP2MSB     ; MSBs of capt. time
          BIT      #TAIFG,&TACTL      ; TIMOV pending?
          JZ       TM22               ; No, values are correct
          TST      &CCR2              ; Yes, check CCR2
          JN       TM22               ; CCR2 > 7FFFh: correct values
          INC      TIM2MSB            ; MSBs not yet updated
TM22      SUB      TIM2LSB,PP2LSB     ; Build difference
          SUBC     TIM2MSB,PP2MSB     ;
          ...                         ; Task 2 to do
          RETI
;
; Timer Block 3 captures the time of a leading edge at CCI3A
; TIM3MSB and TIM3LSB contain the time of the actual edge
;
TIMMOD3   .EQU     $                  ; Vector 6: Block 3
          MOV      &CCR3,TIM3LSB      ; Store LSBs of event time
          MOV      TIMAEXT,TIM3MSB    ; MSBs of event time
          BIT      #TAIFG,&TACTL      ; TIMOV pending?
          JZ       TM31               ; No, values are correct
          TST      &CCR3              ; Yes, check CCR3
          JN       TM31               ; CCR3 > 7FFFh: correct values
          INC      TIM3MSB            ; MSBs not yet updated
TM31      ...                         ; Task 3 to do
          RETI
;
; Timer Block 4 gen. a symmetric pulse at pin TA4
; f = 0.5 x TCLK/TIM4xSB
;
TIMMOD4   .EQU     $                  ; Vector 8: Block 4
          BIT      #TAIFG,&TACTL      ; TIMOV pending?
          JNZ      TM41               ; Yes, checks necessary
TM42      CMP      TIMAEXT,TIMAEXT4   ; MSBs also equal?
```

```
        JEQ     T43             ; Interval is reached
        RETI
;
TM41    TST     &CCR4           ; TAIFG = 1: check CCR4
        JN      TM42            ; CCR4 > 7FFFh: correct values
        PUSH    TIMAEXT         ; TIMAEXT not yet updated
        INC     0(SP)           ; Updated value of TIMAEXT
        CMP     @SP+,TIMAEXT4   ; MSBs equal?
        JNE     TM4R            ; No, return
T43     ADD     TIM4LSB,&CCR4   ; LSBs of interval
        ADDC    TIM4MSB,TIMAEXT4 ; MSBs of interval
        XOR     #OUT,&CCTL4     ; Toggle TA4 without Output Unit
        ...                     ; Task 4
TM4R    RETI
;
        .sect   "TIMVEC",0FFF0h ; Timer_A Interrupt Vectors
        .word   TIM_HND         ; Timer Blocks 1..4 Vector
        .word   TIMMOD0         ; Vector for Timer Block 0
        .sect   "INITVEC",0FFFEh ; Reset Vector
        .word   INIT
```

The example above results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

CCR0 — repetition rate 1 kHz	5 cycles for the task, 11 cycles overhead	16 cycles
CCR1 — repetition rate 58 Hz	16 cycles for the task, 16 cycles overhead	32 cycles
CCR2 — rep. rate max. 58 Hz	30 cycles for the task, 16 cycles overhead	46 cycles
CCR3 — rep. rate max. 58 Hz	25 cycles for the task, 16 cycles overhead	41 cycles
CCR4 — rep. rate max. 58 Hz	32 cycles for the task, 16 cycles overhead	48 cycles
TIMOV — 58 Hz with 3.8 MHz	4 cycles for the task, 14 cycles overhead	18 cycles

$$u_{CPU} = \frac{16 \times 10^3 + 32 \times 58 + 46 \times 58 + 41 \times 58 + 48 \times 58 + 18 \times 58}{3.801 \times 10^6} = 0.007$$

This results in a nominal CPU loading of approximate 0.7% due to the Timer_A activities (the tasks of the timer blocks 2, 3, and 4 are not included). If f_{MCLK} is chosen to be 1 MHz, then the CPU loading is 2.6%, max.

6.3.8.7 MSP430 Operation Without a Crystal

The MSP430 may be used without a 32 kHz crystal. The FLL loop is opened and a DCO tap with a frequency near the desired frequency is used (the dependence of the MCLK frequency on the DCO tap used is shown in the *MSP430 Architecture Guide and Module Library*). If an application requires a relatively stable MCLK frequency, DCO control by software is possible. The MCLK frequency is compared to the AC line frequency or another external stable reference frequency. No capture/compare register is necessary, but if one is used, LCD operation is also possible.
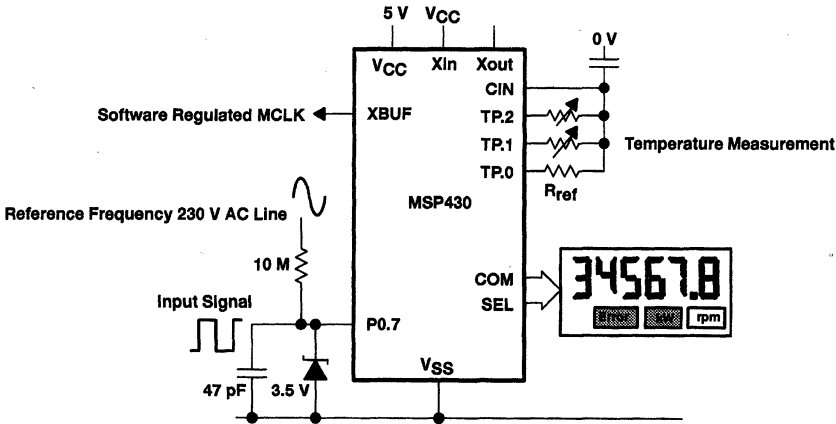
Figure 6–30. MSP430 Operation Without Crystal

Any external reference frequency, f_{REF}, may be used if it is in the range:

$$\frac{k \times 2^{16}}{f_{CLK}} < f_{REF} < \frac{f_{MCLK}}{100}$$

The lower limit prevents overflow of the result, the upper (arbitrary) limit prevents overloading of the CPU (the control task needs approximately 50 cycles per reference period).

If the reference frequency is far above the main frequency (kilo Hertz range), then it is recommended to use the P0.0 input due to its dedicated interrupt vector.

If the reference frequency disappears, then the MCLK frequency continues to work with the actual DCO value until the reference frequency appears again. The frequency of the system clock generator does not step down to the lowest DCO frequency due to the open control loop.

Example 6–36. Operation Without Crystal

An ac line-powered system works without a crystal. The line frequency, f_{MAINS} is connected to P0.7, causing interrupt for each positive edge. The P0.7 interrupt handler calculates the cycle difference between two interrupts — which is the number of MCLK cycles during one mains period — and compares this difference to a maximum value, HID, and a minimum value, LOWD. If the difference is out of these limits, the DCO is corrected in small steps (2^2 of n_{DCOmod}). See Section 6.5 *The System Clock Generator* for an explanation of n_{DCOmod}. The nominal value of the frequency f_{MCLK} is chosen to 2 MHz. The hardware is shown in Figure 6–30.

The software example below works up to a maximum frequency f_{CLK}:

$$f_{CLK} < 2^{16} \times k \times f_{MAINS}$$

Where:

f_{CLK}	Input frequency at the input divider input of Timer_A	[Hz]
k	Predivider constant of the input divider (1, 2, 4 or 8)	
f_{MAINS}	AC Line frequency used	[Hz]

If no LCD is used, then the value LCD is set to 0:

```
LCD      .equ     0                    ; No LCD used
```

The value f_{LCD} used with the example is calculated:

$$f_{LCD} = f_{FRAME} \times 2 \times MUX$$

Where:

f_{FRAME}	Recommended frame frequency for the used LCD	[Hz]
MUX	Driving method for the used LCD (1, 2, 3, or 4 MUX)	

```
; Hardware definitions
;
FLLMPY   .equ    64                  ; Only for FN_x selection
TCLK     .equ    40*50000            ; Timer input clock 2.0MHz = MCLK
FMAIN    .equ    50                  ; Mains frequency 50Hz
FLCD     .equ    256                 ; 4MUX: fFRAME = 256/8 = 32Hz
LOWD     .equ    TCLK/FMAIN*99/100   ; Lower MCLK limit 99% TCLK
HID      .equ    TCLK/FMAIN*101/100  ; Upper MCLK limit 101%
TCLK LCD .equ    1                   ; 1: LCD drive implemented too
;
; RAM definitions
;
TARSTOR  .equ    202h                ; Last TAR content for delta
CNTMAINS .equ    204h                ; Mains frequency counter
STACK    .equ    300h                ; Stack address
         .text 0F000h                ; Software start address
INIT     MOV     #STACK,SP
         CALL    #INITSR             ; Initialize RAM, set FN_2
;
; Prepare System Clock Generator for operation without crystal
;
         BIS.B   #SCG0,SR            ; FLL: loop Control off
         CLR.B   &SCFQCTL            ; Modulation on
         MOV.B   #050h,&SCFI1        ; Tap 10: 2MHz (nom. with FN_2)
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT on
;
         MOV     #ISMCLK+MCONT+TAIE,&TACTL
         MOV     #OMOO+CCIE,&CCTL4   ; TA4 not used, Intrpt on
         BIS.B   #080h,&P0IE         ; Enable INTRPT for P0.7 (mains)
         BIC.B   #080h,&P0IES        ; INTRUPT for leading edge
         .if     LCD=1               ; If LCD is needed too
         MOV.B   #0,&BTCTL           ; Prepare Basic Timer. Use fLCD
         .endif
         MOV.B   #CBMCLK+CBE,&CBCTL  ; MCLK at XBUF pin
         EINT
```

```
MAINLOOP ...
;
; Interrupt handler Port0: P0.2 to P0.7. The mains input
; is at pin P0.7
;
P072_HNDL PUSH    R5                   ; Save R5
          BIT.B   #080h,&P0FG          ; P0.7 (mains) INTRPT?
          JZ      P062                 ; No, check P0.6 to P0.2
          MOV     &TAR,R5              ; Act. Timer Register
          SUB     TARSTOR,R5           ; Build delta MCLK cycles
          MOV     &TAR,TARSTOR         ; For next MCLK measurement
          CMP     #LOWD,R5             ; fMCLK < lower MCLK limit?
          JHS     P07T1                ; No, check upper limit
          INC.B   &SCFI1               ; Yes, increase DCO frequency
P07T1     CMP     #HID,R5              ; fMCLK > upper MCLK limit?
          JLO     P07T2                ; No, return
          DEC     &SCFI1               ; Yes, decrease DCO frequency
P07T2     INC     CNTMAINS             ; Mains counter + 1 (time base)
;
P062      MOV.B   &P0IFG,R5            ; Read P0 flags
          BIC.B   R5,&P0IFG            ; Reset read flags (P0.7 to P0.2)
          ....                         ; Process inputs P0.6 to P0.2
P072RET   POP     R5                   ; All done, return
          RETI
;
          .if     LCD=1                ; If LCD is needed too
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
;
TIM_HND   ADD     &TAIV,PC             ; Add Jump table offset
          RETI                         ; Vector 0: No interrupt
          JMP     TIMMOD1              ; Vector 2: Block 1
          JMP     TIMMOD2              ; Vector 4: Block 2
          JMP     TIMMOD3              ; Vector 6: Block 3
          JMP     TIMMOD4              ; Vector 8: Block 4
          RETI                              ; TIMOV not used here
```

```
;
; The interrupt handlers for the Timer Blocks 0 to 3 follow
; They are not implemented here
;
TIMMOD0  .equ    $               ; Handler for Timer Block 0
TIMMOD1  .equ    $               ; Handler for Timer Block 1
TIMMOD2  .equ    $               ; Handler for Timer Block 2
TIMMOD3  .equ    $               ; Handler for Timer Block 3
         RETI
;
; Timer Block 4: interrupt handler for the LCD drive. The
; BTCNT1 register - which generates fLCD - is incremented
; twice with the fLCD period to generate both edges
;
TIMMOD4  ADD.B   #010h,&BTCNT1    ; Toggle BTCNT1.4
         ADD     #TCLK/(2*FLCD),&CCR4 ; Add 1/(2*fLCD)
         RETI
         .endif                   ; End of LCD drive part
;
.sect    "TIMVEC",0FFF0h          ; Timer_A Interrupt Vectors
         .word   TIM_HND          ; Timer Blocks 1..4 Vector
         .word   TIMMOD0          ; Vector for Timer Block 0
         .sect   "P072VEC",0FFE0h ; P0.x Vector
         .word   P072_HNDLR       ; Handler for P0.7 to P0.2
         .sect   "INITVEC",0FFFEh ; Reset Vector
         .word   INIT
```

The example results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

LCD timing — 2×256 Hz	10 cycles for the task, 16 cycles overhead	26 cycles
MCLK frequency control 50 Hz	49 cycles for the task, 11 cycles overhead	60 cycles

$$u_{CPU} = \frac{26 \times 512 + 60 \times 50}{2.0 \times 10^6} = 0.008$$

This results in a CPU loading of approximate 0.8% due to the Timer_A tasks, the LCD timing, and the MCLK control.

6.3.8.8 RF Timing Generation

Different modulation methods for RF timing generation are shown in Figure 6–32. All are used with metering devices (electric meter, water meter, gas meter, heat allocation meters, etc.) for the long-distance readout of the consumption.

For the generation of the modulated RF, normally a regulated 6-V supply voltage is used. If this voltage is not available, the step-up power supply shown in figure 6–31 may be used. An existing supply voltage (here 3 V) is transformed by the step-up circuit to 8 V and regulated down to the desired 6 V. The step-up frequency is delivered by the MSP430. The XBUF output, with its four possible output frequencies, is used. The sequence starts with the ACLK frequency (32.768 kHz) and then lowers to ACLK/2 and ACLK/4. In this way, the CPU is not loaded with the frequency generation at all. Figure 6–31 illustrates the connection of an RF interface module to an MSP430.
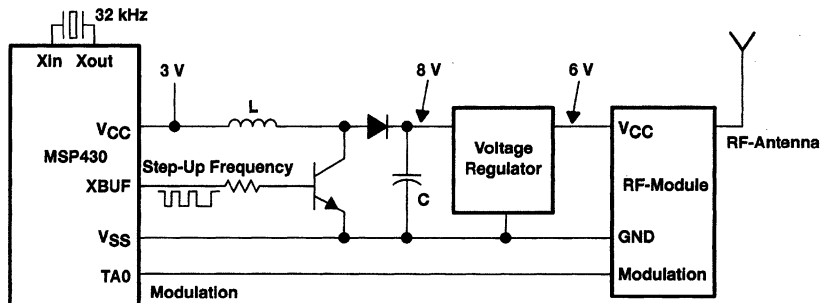
Figure 6–31. RF Interface Module Connection to the MSP430

Modulation modes used are:

❏ **Amplitude Modulation** — the RF oscillator is switched on for a logical 1 and switched off for a logical 0 (100% modulation).

❏ **Biphase Code** — the information is represented by a bit time consisting of one half bit without modulation and one half bit with full modulation. A logical 1 starts with 100% modulation, a logical 0 starts with no modulation.

❏ **Biphase Space** — a logical 1 (space) is represented by a constant signal during the complete bit time. A logical 0 (mark) changes the signal in the middle of the bit time. The signal changes after each transmitted bit.

The last two modulation modes do not have a dc part. Figure 6–32 shows all three modulations modes. If the LSBs are transmitted first, the information sent is 096h.
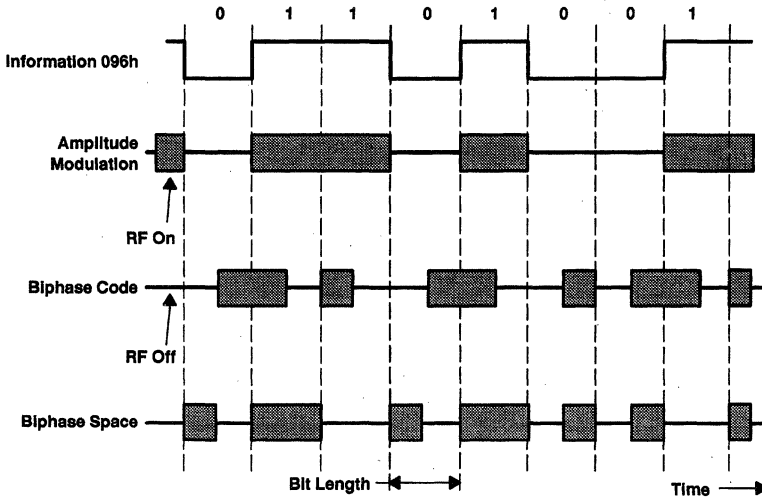
Figure 6–32. RF Modulation Modes

The timer block 0 is used with the software examples for all three modes due to the following two reasons:

❏ The fastest possible response. The decision making with the timer vector register is not necessary for the timer block 0 — it uses its own, dedicated interrupt vector.

❏ The capture/compare register 0 interrupts not only if the timer register and CCR0 are equal (like with the other CCRs), but also if the timer register contains a higher value. This prevents the loss of synchronity due to other interrupts during the transmission.

The software of the other four timer blocks is not shown with the following software routines. Many examples of their use are given in the previous examples. The software examples also show how to output the ACLK/2 frequency at output terminal XBUF. This accurate frequency may be used for the clocking of external peripherals.

6.3.8.8.1 RF Amplitude Modulation

This is the simplest method — a set data bit (1) switches on the RF, a zero data bit switches off the RF.
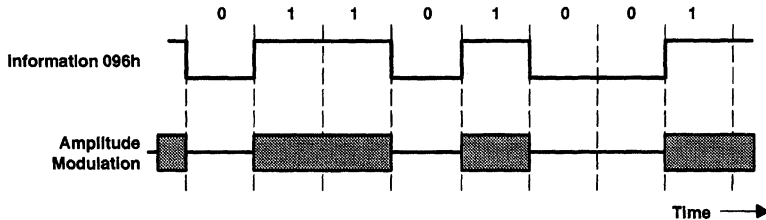
Figure 6–33. Amplitude Modulation

❏ If the speed of the software is not sufficient, dedicated registers (R4 to R15) may be used for RFDATA and RFCOUNT. This register method is used with the biphase code and biphase space software. See sections 8.8.2 and 8.8.3.

❏ If the MSB needs to be output first, then the instruction `RRA RFDATA` (after label `TM01`) is simply replaced by `RLA RFDATA`

Example 6–37. Amplitude Modulation Methods

```
; Software example: Amplitude Modulation methods.
;
; Hardware definitions
;
FLLMPY    .equ    48              ; FLL multiplier for 1.572MHz
TCLK      .equ    FLLMPY*32768    ; TCLK: FLLMPY x fcrystal
fRF       .equ    19200           ; Bit rep. frequency (Baud Rate)
Bit_Length .equ   ((2*TCLK/fRF)+1)/2 ; Bit length (TCLK cycles)
STACK     .equ    600h            ; Stack initialization address
;
; RAM definitions. Use dedicated CPU registers (R4 to R15)
; if the speed is not sufficient
;
RFDATA    .equ    202h            ; 16 bit data to be sent
TIMAEXT   .equ    204h            ; 32 bit extension Timer_A
RFCOUNT   .equ    206h            ; Counter for 16 bits (byte)
```

```
;
        .text 0F000h              ; Software start address
;
INIT    MOV     #STACK,SP         ; Initialize Stack Pointer
        CALL    #INITSR           ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Cont. Mode, no INTRPT
;
        MOV     #ISMCLK+CLR,&TACTL
        MOV     #OMOO,&CCTL0       ; Reset TA0, INTRPT off
        MOV.B   #TA0,&P3SEL        ; Define TA0 output
;
        CLR     TIMAEXT            ; Clear TAR extension
        MOV.B   #3,&CBCTL          ; Output ACLK/2 at XBUF pin
        BIS     #MCONT,&TACTL      ; Start Timer
        EINT                       ; Enable interrupt
MAINLOOP ...                       ; Continue in background
;
; A 16 bit value is to be output. R5 contains data
;
        MOV     R5,RFDATA          ; Value into data word
        MOV.B   #16+2,RFCOUNT      ; Bit count+2 to RFCOUNT
        MOV     &TAR,&CCR0         ; For fast response:
        ADD     #100,&CCR0         ; Time of 1st bit test
        MOV     #OMOO+CCIE,&CCTL0 ; Enable interrupt for CCR0
        ...                        ; Continue in background
;
; Test in background if 16 bits are output: RFCOUNT = 0
;
        TST.B   RFCOUNT            ; Output completed?
        JZ      BPC_MADE           ; Yes, interrupt bit is reset
        ...                        ; No continue
;
; Interrupt handler for Capture/Compare Block 0.
; Data in RFDATA is output: LSB first
;
```

```
TIMMOD0   .EQU    $                    ; Start of CCR0 handler
          ADD     #Bit_Length,&CCR0 ; Time of next bit change
          DEC.B   RFCOUNT              ; Bit count - 1
          JNZ     TM01                 ; Not zero: continue
          MOV     #OMR,&CCTL0          ; Finish output: reset TA0
          RETI                         ; INTRPT off
;
TM01      RRA     RFDATA               ; Next bit of RFDATA
          JC      TM02                 ; Bit is one
          MOV     #OMR+CCIE,&CCTL0  ; Bit is 0: prepare reset
          RETI
;
TM02      MOV     #OMSET+CCIE,&CCTL0 ; Bit is 1: prepare set
          RETI                         ;
;
          .sect   "TIMVEC",0FFF2h   ; Timer_A Interrupt Vector
          .word   TIMMOD0              ; Vector for Timer Block 0
          .sect   "INITVEC",0FFFEh  ; Reset Vector
          .word   INIT
```

The example results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}}\ (n_{intrpt} \times f_{rep}) = \frac{33 \times 19200}{1.572E6} = 0.44$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

The RF amplitude modulation loads the CPU to 44% of its capacity when running with 1.572 MHz.

6.3.8.8.2 RF Biphase Code Modulation

The biphase code modulation represents each data bit by a change of the information in the middle of the sent data bit:

☐ Data bit is 0: the information starts with 0 (RF off) and in the middle of the info bit the RF is switched on for the remaining half of the bit time.

☐ Data bit is 1: the information starts with 1 (RF on) and in the middle of the info bit the RF is switched off for the remaining half of the bit time.
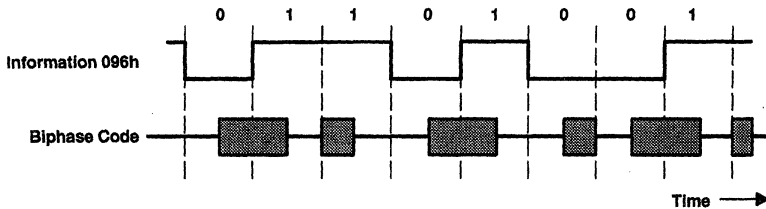
Figure 6–34. Biphase Code Modulation

Due to the information change in the middle of the data bit, biphase code modulation needs twice the repetition rate of amplitude modulation — 38400 bits/ second for a baud rate of 19200. Therefore, a system clock frequency of 1.048 MHz is not sufficient for this modulation. Instead, 1.606 MHz is selected for the MCLK frequency for biphase modulation. All members of the MSP430 family can use this frequency.

The information is not converted in real time due to the high transmission rate of 38400 bits/second. The conversion is made before the transmission — bytes from eight arbitrary addresses (ADDRESS0 to ADDRESS7) are converted and the bit pattern stored in a RAM block of 128 bits in length. This 128-bit buffer is output in real time.

Example 6–38. Biphase Code Modulation

```
; Software example: Bi-Phase Code Modulation
;               Input in R6          Output in R5
; Some examples:  096h       ->      06996h          Input -> Output
;                 000h       ->      0AAAAh
;                 0FFh       ->      05555h
;                 069h       ->      09669h
;                 011h       ->      0A9A9h
;
```

```
; Hardware definitions
;
FLLMPY    .equ    49                          ; FLL multiplier for 1.606MHz
TCLK      .equ    FLLMPY*32768      ; TCLK: FLLMPY x fcrystal
fRF       .equ    38400             ; Bit rep. frequency
Bit_Length .equ   ((2*TCLK/fRF)+1)/2 ; Bit length (TCLK cycles)
STACK     .equ    600h              ; Stack initialization address
;
; RAM Definitions
;
RF_BLK    .equ    202h              ; Converted data 16 bytes
TIMAEXT   .equ    212h              ; 32 bit extension Timer_A
          .text 0F000h             ; Software start address
;
INIT      MOV     #STACK,SP         ; Initialize Stack Pointer
          CALL    #INITSR           ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Cont. Mode, INTRPT on
;
          MOV     #ISMCLK+TAIE+CLR,&TACTL
          MOV     #OMOO,&CCTL0      ; Reset TA0, INTRPT off
          MOV.B   #TA0,&P3SEL       ; Define TA0 output
;
          CLR     TIMAEXT           ; Clear TAR extension
          MOV.B   #3,&CBCTL         ; Output ACLK/2 at XBUF pin
          BIS     #MCONT,&TACTL     ; Start Timer
          EINT                      ; Enable interrupt
MAINLOOP ...                        ; Continue in background
;
; 64 bits of data is to be output with Bi-Phase Code Modul.
; The data is converted into a 128-bit RAM block with the
; Bi-Phase Code sequence
;
          CLR     R5                ; For Bi-Phase Space necessary
          MOV     #RF_BLK,R8        ; Address 8 word send block
          MOV.B   ADDRESS0,R6       ; 1st data byte to R6
```

```
        CALL    #BI_PHASE_CODE    ; Convert it to 16 bits
        MOV     R5,0(R8)          ; Converted data to RF-Block
        ...                       ; Convert next 6 bytes same way
        MOV.B   ADDRESS7,R6       ; 8th data byte to convert to R6
        CALL    #BI_PHASE_CODE    ; Convert to 16 bits
        MOV     R5,14(R8)         ; Converted data to RF-Block
;
        MOV     #RF_BLK+16,R9     ; 1st word after RF_BLK
        MOV     #16+1,R6          ; Bit count for 1st 16 bits
        MOV     @R8+,R5           ; 1st 16 bits for output
;
; Switch off all interrupts to allow exact RF timing. This is
; not necessary if ALL OTHER interrupt handlers start with
; an EINT instruction
;
        ...
;
        MOV     &TAR,&CCR0        ; For fast response:
        ADD     #100,&CCR0        ; Time of 1st bit test
        MOV     #OMOO+CCIE,&CCTL0 ; Enable interrupt for CCR0
        ...                                 ; Continue in background
;
; Test in background if 128 bits are output: INTRPT of Timer
; Block 0 is switched off by the INTRPT handler
;
        BIT     #CCIE,&CCTL0      ; Output completed?
        JZ      BPC_MADE          ; Yes
        ...                       ; No continue
;
; Interrupt handler for Capture/Compare Block 0
; Data in RF_BLK is output: LSB first
;
TIMMOD0 .EQU    $                 ; Start of CCR0 handler
        ADD     #Bit_Length,&CCR0 ; For next INTRPT
        DEC     R6                ; Bit count - 1
        JNZ     TM01              ; Not zero: continue
```

```
        MOV     @R8+,R5              ; Next 16 bits for output
        MOV     #16,R6              ; Bit count
        CMP     R9,R8              ; End of buffer reached?
        JHS     TM03              ; Yes, finish output
;
TM01    RRC     R5              ; Next data bit to carry
        JC      TM02              ; Bit is one
        MOV     #OMR+CCIE,&CCTL0  ; Bit is 0: prepare reset
        RETI
;
TM02    MOV     #OMSET+CCIE,&CCTL0 ; Bit is 1: prepare set
        RETI
;
TM03    MOV     #OMR,&CCTL0        ; Output complete:
        RETI                       ; Reset TA0, INTRPT off
;
; Subroutine transforms the data byte in R6 (8 bits) to
; Bi-Phase Code in R5 (16 bits). CALL + 86 cycles/byte
;
BI_PHASE_CODE .equ $              ; Conversion routine
        MOV     #8,R7              ; Convert 8 bits
BIPL    RRA     R6              ; LSB to Carry
        RRC     R5              ; Bit to R5 MSB
        BIT     #8000h,R5          ; Copy bit once more
        RRC     R5              ; to R5
        XOR     #8000h,R5          ; and invert it
        DEC     R7              ; Bit count - 1
        JNZ     BIPL              ; 8 bits not yet converted
        RET                        ; 16 info bits in R5
;
        .sect   "TIMVEC",0FFF2h   ; Timer_A Interrupt Vector
        .word   TIMMOD0            ; Vector for Timer Block 0
        .sect   "INITVEC",0FFFEh  ; Reset Vector
        .word   INIT
```

The example results in a CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \; \Sigma \left(n_{intrpt} \times f_{rep} \right) = \frac{\left(27 \times 15/16 + 34 \times 1/16\right) \times 2 \times 19200}{1.606E6} = 0.66$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

This results in a MSP430 CPU load of 66% when outputting Biphase Code Modulation at 19200 baud with an MCLK frequency of 1.606 MHz.

RF Biphase Space Modulation

The realtime software — that outputs the 128-bit block — is exactly the same as the biphase code modulation shown in Section 8.8.2. Only the subroutine that converts the binary data to the biphase space code is different — the actual bit depends also on the previous bit. Therefore, only the different conversion subroutine is shown below. The CPU loading is, due to the equal real time part, is also 66%, like it is for the biphase code modulation.
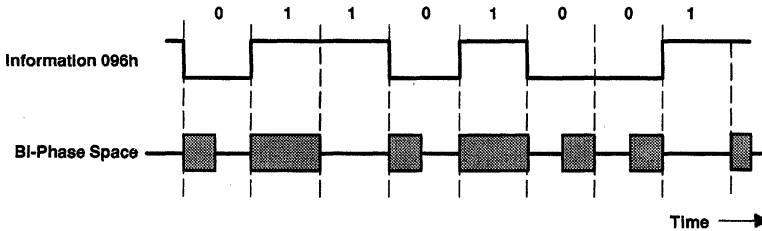
Figure 6–35. Biphase Space Modulation

Example 6–39. Biphase Space Modulation

The information cannot be converted in real time due to the high transmission speed of 38400 bits/second. The conversion is made before the transmission: bytes from eight arbitrary addresses (ADDRESS0 to ADDRESS7) are converted and the bit pattern is stored in a RAM block with 128 bits in length. This 128-bit buffer is output in real time by the timer block 0. See Section 8.8.2.

```
;
; Subroutine converts the data byte in R6 (8 bits) to
; Bi-Phase Space Code in R5 (16 bits). CALL + 162 cycles/byte
; R5 contains the MSB (2nd half bit) of the last conversion.
```

```
;
;               Input in R6        Output in R5
; Some examples: 096h      ->      02B4Dh  Prev. 2nd half bit = 0
;                000h      ->      05555h
;                0FFh      ->      03333h
;                069h      ->      04D2Bh
;                011h      ->      054abh
;                016h      ->      0AB4Dh
;                000h      ->      0AAAAh  Prev. 2nd half bit = 1
;                0FFh      ->      0CCCCh
;
BI_PHASE_SPACE .equ $              ; Conversion routine
          MOV    #8,R7             ; Number of bits
BPSL      CLR    R9                ; Table Pointer
          BIT    #8000h,R5         ; Test last half bit (MSB R5)
          RLC    R9                ; Bit to LSB
          RRC    R6                ; Next info bit
          RLC    R9                ; 2 bit table address in R9
          MOV.B  BPSTAB(R9),R9     ; Data for 2 bits to be sent
          SWPB   R9                ; 00x0 -> x000
          CLRC                     ; Free two bits for new data
          RRC    R5                ; in R5
          RRA    R5
          ADD    R9,R5             ; Insert new data to MSBs
          DEC    R7                ; Bit count - 1
          JNZ    BPSL              ; 8 bits not yet converted
          RET                      ; 16 info bits in R5
;
; Table with 2-bit info for all possible four bit combinations
;                            Prev Half-Bit    Curr. Bit          Info Bits
BPSTAB    .byte  040h        ;  0              0                  10
          .byte  0C0h        ;  0              1                  11
          .byte  080h        ;  1              0                  01
          .byte  000h        ;  1              1                  00
```

The example results in a constant CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities of 66%, as with the biphase code modulation due to the equal RF part.

6.3.8.9 Real Time Clock

The Timer_A can also be used as a real time clock (RTC), especially in the low power mode 3 (LPM3). The ACLK is summed up and when one of the capture/compare registers is equal to the timer register (TAR), then an interrupt wakes up the CPU. The interrupt handler adds the time interval to the capture/compare register and returns to LPM3. Due to the available five capture/compare registers, up to five independent wake up frequencies may be programmed. Their handlers have the same structure as shown here for timer block 1.

The timer overflow delivers an additional 0.5 Hz wake up frequency ($2^{16}/32768$ Hz = 2 s). If this timing is sufficient, no interrupt by a capture/compare register is necessary.
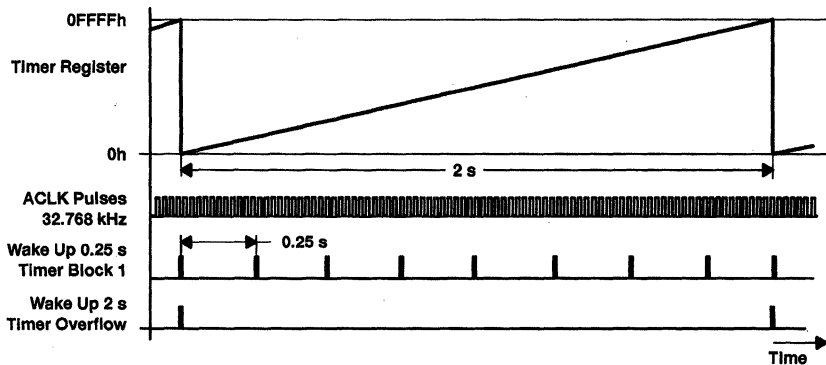
Figure 6–36. Real Time Clock Application of the Timer_A

The software example shows a real time application with a wake up every 0.25 s initiated by timer block 1 (the software for the other timer blocks is not shown). The 0.25 s interrupt increments a RAM counter (RTC_CNT) and updates the time and date if a full second has elapsed.

Example 6–40. Real Time Clock Application of the Timer_A

```
; Software example: Real Time Clock application of the
; Timer_A running in the Continuous Mode
;
; Hardware definitions
;
```

```
FLLMPY     .equ    32                 ; FLL multiplier for 1.048MHz
TCLK       .equ    FLLMPY*32768       ; TCLK: FLLMPY x fcrystal
RTC_DELTA  .equ    32768/4            ; ACLK delta for 4Hz wake-up
STACK      .equ    600h               ; Stack initialization address
;
; RAM definitions
;
RTC_CNT    .equ    202h               ; RTC counter for the 4Hz
TIMAEXT    .equ    204h               ; TAR extension
;
           .text   0F000h             ; Software start address
;
INIT       MOV     #STACK,SP          ; Initialize Stack Pointer
           CALL    #INITSR            ; Init. FLL and RAM
;
; Initialize the Timer_A: ACLK, Cont. Mode, INTRPT on
;
           MOV     #ISACLK+TAIE+CLR,&TACTL
           CLR     &CCR0              ; Defined start value
           CLR     TIMAEXT            ; Clear TAR extension
           MOV     #OMOO+CCIE,&CCTL1  ; CCR1 used for RTC
           MOV.B   #CBACLK+CBE,&CBCTL ; Output ACLK at XBUF pin
           BIS     #MCONT,&TACTL      ; Start Timer
           EINT                       ; Enable interrupt
MAINLOOP   ...                        ; Continue in background
;
; Enter LPM3. The watchdog must be held (ACLK continues)
;
           MOV     #05A00h+HOLD+CNTCL,&WDTCTL ; Hold watchdog
           BIS     #CPUOFF+GIE+SCG1+SCG0,SR ; Enter LPM3
           ...
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
```

```
TIM_HND  ADD      &TAIV,PC          ; Add Jump table offset
         RETI                       ; Vector 0: No interrupt
         JMP      TIMMOD1           ; Vector 2: Block 1
         JMP      TIMMOD2           ; Vector 4: Block 2
         JMP      TIMMOD3           ; Vector 6: Block 3
         JMP      TIMMOD4           ; Vector 8: Block 4
;
; Block 5. Timer Overflow Handler: the Timer Register is
; expanded into the RAM location TIMAEXT (MSBs). TIMAEXT is
; incremented every 2s
;
TIMOVH   .EQU     $                 ; Vector 10: TIMOV Flag
         INC      TIMAEXT           ; Incr. Timer extension
         ...                        ; 0.5Hz task starts here
         RETI                       ;
;
; Timer Block 1 is used for the Real Time Clock
; Repetition Rate = 4Hz (0.25s)
;
TIMMOD1  .EQU     $                 ; Vector 2: Block 1
         ADD      #RTC_DELTA,&CCR1  ; Add time interval (0.25s)
         ...                        ; RTC Task 4Hz starts here
         INC      RTC_CNT           ; Increment 4Hz counter
         BIT      #3,RTC_CNT        ; one second elapsed?
         JZ       TASK              ; Yes
         RETI                       ; No, back to LPM3
;
TASK     CALL     #RTCLK            ; Time + 1s
         JNC      TM1               ; If carry: 00.00 o'clock
         CALL     #DATE             ; Next day
TM1      RETI                       ; Return to LPM3
;
         .sect    "TIMVEC",0FFF0h   ; Timer_A Interrupt Vectors
         .word    TIM_HND           ; Vector for blocks 1 to 4
         .word    TIMMOD0           ; Vector for Timer Block 0
         .sect    "INITVEC",0FFFEh
         .word    INIT              ; Reset Vector
```

The example results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

CCR1 — repetition rate 4 Hz   16 cycles for the task, 16 cycles overhead   32 cycles

TIMOV — repetition rate 0.5 Hz   4 cycles for the task, 14 cycles overhead   18 cycles

$$u_{CPU} = \frac{1}{f_{MCLK}} \; \sum \left(n_{intrpt} \times f_{rep} \right) = \frac{32 \times 4 + 18 \times 0.5}{1.048E6} = 0.00013$$

Where:

   f_{MCLK} Frequency of the system clock (DCO)   [Hz]

   n_{intrpt} Number of cycles executed by the interrupt handler

   f_{rep} Repetition rate of the interrupt handler   [Hz]

This result means that the MSP430 CPU uses the low power mode 3 during 99.987% of the time when running the RTC software shown (time and date tasks are not included).

6.3.8.10 Conclusion

This section demonstrated the many and versatile possibilities of the Timer_A running in the continuous mode. Any mixture of capture and compare modes is possible with the five capture/compare registers (CCRx). It is also possible to change the mode of a capture/compare register during the run: a capture/compare register used in capture mode during the calibration process may be used as a compare register during the normal run and vice versa.

Also worth mentioning is the absolute synchronization of the generated timings. This is a result of the single timer register used for all capture/compare registers. This feature is very important for digital motor control applications. The read/write feature of all the Timer_A registers additionally offers possibilities beyond the scope of this discussion.

6.3.9 Software Examples for the Up Mode

This section shows several proven application examples for the Timer_A running in the up mode. The software definitions appear near the beginning of this section. Whenever possible, the abbreviations defined in the *MSP430 Architecture Guide and Module Library* are used with the software examples.

The software examples are written to be independent of the MCLK frequency in use. Only the FLL multiplier constant, FLLMPY, and the period for the period register need to be redefined if another combination is needed. The source lines for the definition of these important values are:

```
FLLMPY    .equ    122                    ; 1. FLL multiplier
fper      .equ    19200                  ; 2. PWM Repetition rate
TCLK      .equ    FLLMPY*32768/4         ; 3. FLLMPY x fcrystal/4
PERIOD    .equ    ((2*TCLK/fper)+1)/2    ; 4. Period of the PWM
```

❑ **Definition of the CPU frequency fMCLK.** The multiplier FLLMPY for the digitally controlled oscillator (DCO) is defined. The value for the actual frequency f_{MCLK} is (FLLMPY $\times 2^{15}$). The value 122 stands for f_{MCLK} = 122 $\times 2^{15}$ = 3.9977 MHz.

❑ **Definition of the desired repetition rate.** The value 19200 stands for f_{per} = 19.2 kHz.

❑ **Definition of the input frequency for the Timer Register (TAR).** The expression /4 indicates that the input divider is set to the *Divide-by-Four* mode. The shown value stands for TCLK = 3.9977 MHz/4 = 999.424 kHz. Only the selected predivider for the input divider (here /4) needs to be defined.

❑ **Calculation of the TCLK cycles for the defined period.** This expression is used for the rounding of the result. No change is necessary for this line.

6.3.9.1 Common Remarks

The up mode is designed primarily for pulse width modulation (PWM) or DC generation applications. If none of these applications is needed, then the continuous mode, with its five independent timings, should be used.

Advantages of the Up Mode:

❑ Free run without CPU load for stable PWM values (e.g. DAC, PWM)

❑ High PWM frequency is possible due to the pure hardware control

❑ Clever selected timings are usable for more than one real time job

Disadvantages of the Up Mode:

❑ Dominance of the period register — it defines the time frame for all other capture/compare blocks (C/C Blocks)

❑ Current switching occurs at the same time for all PWM outputs — this is the case when the timer register (TAR) equals the period register CCR0

6.3.9.1.1 Initialization

The initialization subroutine, INITSR, is used in all examples. This subroutine was described in Section 6.3.8.1. It includes the following tasks:

❑ Checks the reason for the initialization (switch on of the supply voltage, watchdog interrupt, or activation of the RESET input)

❑ Clears the RAM — or not — depending on the result of the above check.

❑ Programs the system clock oscillator (multiplication factor N and optimum current switch FN_2, FN_3, or FN_4)

❑ Allows the digitally controlled oscillator to settle at the appropriate tap, providing the correct MCLK frequency

6.3.9.1.2 Timer Clock

The information in this section is also valid for the continuous mode and the up/down mode.

All software examples use the value FLLMPY — it defines the master clock frequency, f_{MCLK}.

$$f_{MCLK} = FLLMPY \times f_{crystal}$$

If this frequency, f_{MCLK} is too high for the application (it causes values for the timer registers exceeding the range from 0 to 65535, for example), then the input divider of the Timer_A may be used. This allows a prescaling of 1, 2, 4, or 8 for the timer input frequencies (f_{MCLK}, f_{ACLK} or f_{TACLK})

Example 6–41. Prescaling Factor of 2

For a required prescaling of 2, the definitions at the start of each example are simply changed to:

```
FLLMPY    .equ      100              ; FLL multiplier for 3.2768MHz

TCLK      .equ      FLLMPY*32768/2   ; Timer Clock = 1.6384MHz

;

; Input Divider D2 is used to get MCLK/2 for the TCLK

;

          MOV       #ISMCLK+D2+MUP+TAIE+CLR,&TACTL ; Use /2 divider
```

The examples normally use an internally generated timer clock — the MCLK or the ACLK. It is also possible to use an external clock. This clock signal is connected to the TACLK terminal and selected with the following code sequence during the initialization:

```
; Ext. clock, Up Mode, Interrupt enabled, Timer Reg. cleared

;

          MOV       #ISTACLK+MUP+TAIE+CLR,&TACTL
          BIS.B     #TACLK,&P3SEL      ; External clock to Timer_A
```

6.3.9.1.3 Timing Considerations

The five independent timings provided by the continuous mode are not possible anymore in the noncontinuous modes because the period register (CCR0) dictates the timing frame for all other capture/compare blocks. Therefore, the period of the timer must be chosen very carefully to allow all the necessary timings. For example, a period chosen for PWM with 19.2 kHz also allows the timing for a software UART running at 2400 baud (19200/8) or 4800 baud (19200/4).

To allow comparison and capturing also with the noncontinuous modes, it is a good practice to have not only a register that counts the overflows (period counter) — like TIMAEXT used with the continuous mode — but also a 16-bit or 32-bit register that counts the TCLK cycles. This allows the use of simple additions for the calculation of a time point. Otherwise, a multiplication is necessary (period counter × period length) to get the elapsed time. The examples given use both registers:

TIMACNT	Period counter	counts the number of full periods
TIMACYCx	Cycle counter	counts the cycles of the timing (one or more words)

See also figure 6–43; the contents of these two registers are shown there for an example.

Frequencies used by the CPU and the Timer_A. The following software examples are (nearly) independent of the MCLK and timer clock frequency in use. During the assembly, the new values for the period register and the timer clock frequency are calculated. A worst case calculation is necessary if a f_{MCLK} that is too low is used.

Update of Extension Registers. Unlike the case with the continuous mode, the update of these extension registers is made with the interrupt handler of the period register (CCR0). This has three reasons:

❑ The interrupt of the period register occurs one cycle before the TIMOV interrupt

❑ The period register (CCR0) has the highest interrupt priority of all Timer_A interrupts

❑ A dedicated interrupt vector (address FFF2h) allows the fastest response to interrupt requests

Real Time Environment. For all applications of the Timer_A running in one of the noncontinuous modes and using interrupt frequencies in the kilohertz range, it is recommended that strict *real time environment* programming be used. Otherwise, interrupt handlers are delayed and information may be lost. To achieve a *real time environment*, the following simple rules should be applied to all interrupt handlers:

❑ The first instruction after the processing of time critical data — Timer_A related data for the Timer_A handlers, for example — should be the EINT (Enable Interrupt) instruction. This allows nested interrupts, a feature possible due to the stack architecture of the MSP430 family.

❑ Interrupt handlers should be as short as possible. Only the absolutely necessary tasks should be executed (incrementing of counters, update of the

status bytes, etc.). The time consuming main tasks should be shifted to the background, where the software executes them according to the status byte information.

Output Units. The PWM examples shown all use the set/reset mode or the reset/set mode of the output units. This has the advantage — compared to the use of toggling — that no incorrect pulse widths can be generated during the change of the pulse width.

6.3.9.1.4 Interrupt Overhead

The calculations for the CPU loading that are appended to the software examples split the necessary cycles for each capture/compare block into two parts:

❏ **Overhead** — This part sums the cycles that are necessary for the CPU to execute the interrupt (saving of the program counter and the status register, decision on which interrupt needs to be served, restoring of the CPU registers).

❏ **Update or Task** — This part actually does the work that needs to be done (incrementing of counters, changing of status bytes, etc.).

The number of overhead cycles shown with the examples are derived from the following sequences:

■ Interrupt of the period register CCR0 (or other interrupt sources with a dedicated vector):

Cycles from interrupt request to 1st instruction of the interrupt handler:		6 cycles
Return from Interrupt instruction:	`RETI`	5 cycles
Sum of overhead		11 cycles

■ Interrupt of capture/compare registers CCR1 to CCR4:

Cycles from interrupt request to 1st instruction of the interrupt handler:		6 cycles
Decision which source caused the interrupt:	`ADD  &TAIV,PC`	3 cycles
Addressed jump instruction:	`JMP  TIMMODx`	2 cycles
Return from Interrupt instruction	`RETI`	5 cycles
Sum of overhead		16 cycles

■ Interrupt of the timer register TIMOV:

This interrupt needs the same number of cycles as the interrupt of the capture/compare registers, but without the JMP TIMMODx instruction. This results in 14 cycles overhead.

6.3.9.2 Update of the Capture/Compare Registers

If the capture/compare registers are updated asynchronously with the periodic timing of the Timer_A, the output pulses may become too long or may be missing. Therefore, a synchronous update should be used, which means the PWM value is written into a buffer, read out from this buffer at the correct time, and then written into the capture/compare register. Three possibilities exist for the synchronous update:

1) . Frequent update by the appropriate Interrupt Handler

2) Infrequent update by the appropriate Interrupt Handler

3) Update by the interrupt handler of capture/compare block 0

The three possibilities are described in the following paragraphs. To find the appropriate solution for a given timing problem, the following decision path may be used:

❑ Is an individual interrupt task necessary for one ore more than one of the capture/compare blocks? If yes, use solution 1, otherwise continue.

❑ Is a very fast update of the capture/compare registers necessary? If yes, use solution 1, otherwise solution 2.

Frequent Update by the Appropriate Interrupt Handler

The interrupt handler of capture/compare block x updates the capture/ compare register CCRx with the repetition rate defined by the period register (CCR0). This method is necessary if an additional task is to be executed by the interrupt handler — medium preparation effort in the background, fast C/C register change.

This method is used with *Generation of Asymmetric Pulse Width Modulation* and *RF Timing Generation*. The following software examples refer to the first application.

If the range for the PWM output values is limited from 1 cycle to (period–1) cycles, then the following simple update sequence may be used:

```
; R6 contains new PWM info for CCR2. Range: 1 to (period-1).

;

        MOV.B    R6,TA2PWM        ; Actualize PWM buffer
```

If the PWM output values 0% or 100% are actually used (CCRx = 0 resp. CCRx ≥ period), then a special treatment is necessary due to the not-generated interrupt request of the capture/compare block x under these circumstances. The

new CCRx value is then written immediately. To determine these special
cases, the following update sequence may be used:

```
; R6 contains new PWM info for CCR2. Range: 0 to full period.
; Check if an immediate update is necessary:
; This is the case for CCR2 = 0 .or. CCR2 >= period
; Software is written for a constant Period Register CCR0
;
        CMP     #PERIOD,&CCR2   ; CCR2 actually >= period?
        JHS     L$21            ; Yes, update CCR2 immediately
        TST     &CCR2           ; No, CCR2 = 0?
        JNZ     L$22            ; CCR2 > 0: normal procedure
L$21    MOV     R6,&CCR2        ; No interrupt: immed. update
L$22    MOV.B   R6,TA2PWM       ; Actualize TA2PWM buffer
        ...                     ; Continue
```

Infrequent Update by the Appropriate Interrupt Handler

The interrupt handler of capture/compare block x updates the capture/
compare register (CCRx) with a repetition rate given by the calculation speed
of the background program. If a new PWM value is calculated for a capture/
compare block, then an individual flag is set and the interrupt for this capture/
compare block is enabled. The first asynchronous interrupt is rejected and the
second one (synchronous) is used for the update of the capture/compare reg-
ister x. An interrupt task is possible only with the update repetition rate. This
method is used if the PWM values for the update are not available at the same
time — minimum interrupt overhead, individual C/C register change.

This method is used with *Digital–to–Analog Conversion* and *TRIAC Control*.
The following software examples refer to the first application.

If the range for the PWM output values is limited from 1 cycle to (period–1)
cycles, then the following simple update sequence may be used:

```
; R6 contains new PWM info for CCR2. Range: 1 to (period-1).
;
        MOV     R6,DAC0OV       ; Actualize DAC0 pulse length
        BIS.B   #1,FLAG         ; Set update flag for DAC0
        BIS     #CCIE,&CCTL2    ; Enable interrupt for DAC0
        ....                    ; Continue in background
```

If the output values 0% or 100% are actually used (CCRx = 0 resp. CCRx ≥. period), then a special treatment is necessary. The interrupt of the capture/ compare block x is not generated in these cases. To determine these special cases, the following update sequence may be used:

```
; R6 contains the new PWM info for CCR2. Range: 0 to period.
; The interrupt is enabled individually for the update.
; A check is made if a special treatment is necessary:
; CCR2 = 0 .or. CCR2 >= period
; Software is written for a variable Period Register CCR0
;
        MOV     R6,DAC0OV       ; Actualize DAC0 pulse length
        CMP     &CCR2,&CCR0     ; CCR2 >= period actually?
        JLO     L$21            ; Yes, update CCR2 immediat.
        TST     &CCR2           ; No, is CCR2 = 0 actually?
        JNZ     L$22            ; No, proceed normally
L$21    MOV     R6,&CCR2        ; Update CCR2 immediately
        JMP     L$23            ; Update made, no interrupt
L$22    BIS.B   #1,FLAG         ; Set update flag for DAC0
        BIS     #CCIE,&CCTL2    ; Enable interrupt for DAC0
L$23    ....                    ; Continue in background
```

For a constant period register (CCR0), the sequence is:

```
; Software is written for a constant period register CCR0
;
        MOV     R6,DAC0OV       ; Actualize DAC0 pulse length
        CMP     #PERIOD,&CCR2   ; CCR2 >= period actually?
        JHS     L$21            ; Yes, update CCR2 immediat.
        TST     &CCR2           ; No, is CCR2 = 0 actually?
        ....                    ; Same as above;
```

Update by the Interrupt Handler of Capture/Compare Block 0

The interrupt handler of capture/compare block 0 updates the capture/compare registers (CCRx) with the repetition rate given by the period register (CCR0). No additional tasks are possible for the other capture/compare blocks — their interrupts are disabled. The output units control the TAx outputs without software overhead — minimum interrupt overhead, fastest C/C register change. If an update is made from relatively large CCRx values to small ones (approximately interrupt latency time), then 100% pulses may occur. Therefore, this method is only recommended for small changes of the PWM value. This method can be used only if:

❑ A very fast update is necessary

❑ Only a minimum overhead can be tolerated (no additional handler is needed, the CCR0 handler is only slightly longer due to this operation)

❑ Erroneous output pulses with 100% length can be tolerated

An example for this update method is given in section *Capturing with the Up Mode* for capture/compare block 4.

These three possibilities may be mixed if it is advantageous. The examples of this section apply the same solution for all capture/compare blocks.

Table 6–20 shows the overhead calculation and the percentage of the update overhead for the three different update methods. The calculation results are based on:

Where:

f_{MCLK}	Frequency of the DCO (MCLK)	4.0 MHz
fupdate	Update frequency for the capture/compare registers	1.0kHz
fper	Timer_A repetition rate (defined by the period register CCR0)	19.2 kHz
n	Number of C/C blocks used for the PWM generation	3

Table 6–20. Interrupt Overhead for the three different Update Methods

UPDATE METHOD	OVERHEAD FORMULA (CPU CYCLES)	OVERHEAD PERCENTAGE
Frequent Update with appropriate Handler	$n \times fper \times 22$	31.7%
Infrequent Update with appropriate Handler	$n \times fupdate \times 44$	3.3%
Update by Capture/Compare Block 0	$n \times fper \times 6$	8.6%

Note:

No interrupts are generated — and therefore no interrupt overhead — for capture/compare registers containing 0 or a value greater than or equal to the period register (CCR0).

6.3.9.3 Generation of Asymmetric Pulse Width Modulation (PWM)

The medium output voltage V_{PWM} at the pin TAx resp. the necessary register content nccrx for a given voltage V_{PWM} is:

$$V_{PWM} = Vcc \times \frac{n_{CCRx}}{n_{CCR0} + 1} = Vcc \times \frac{tpw}{tper} \rightarrow n_{CCRx} = \frac{V_{PWM}}{Vcc} \times (n_{CCR0} + 1)$$

Where:

V_{PWM}	Medium output voltage at the TAx pin	[V]
V_{CC}	Supply voltage of the system	[V]
n_{CCR0}	Content of the period register CCR0	
n_{CCRx}	Content of the capture/compare register CCRx	
t_{pw}	Time generated by the capture/compare register	[s]
t_{per}	Period generated by the period register CCR0	[s]

Table 6–21 shows the necessary content of a capture/compare register CCRx to get some defined unsigned output values for V_{PWM}:

Table 6–21. Output Voltages for unsigned PWM

OUTPUT VOLTAGE (VPWM)	CONTENT OF CCRx n_{CCRx}
0 V	0
$0.25 \times V_{CC}$	$(n_{CCR0} + 1) \times 0.25$
$0.5 \times V_{CC}$	$(n_{CCR0} + 1) \times 0.5$
$0.75 \times V_{CC}$	$(n_{CCR0} + 1) \times 0.75$
V_{CC}	$(n_{CCR0} + 1)$

If the output voltage is seen as a signed voltage — like for 3-phase digital motor control — then the voltage $0.5 \times V_{CC}$ is seen as the 0 point. The signed output voltage V_{PWM} gets:

$$V_{PWM} = Vcc \times \left(\frac{n_{CCRx}}{n_{CCR0} + 1} - 0.5\right) \rightarrow n_{CCRx} = \left(\frac{V_{PWM}}{Vcc} + 0.5\right) \times (n_{CCR0} + 1)$$

Table 6–22 shows the contents of a capture/compare register (CCRx) required to get some defined values for a signed output voltage V_{PWM}:

Table 6–22. Output Voltages for Signed PWM

OUTPUT VOLTAGE (VPWM)	CONTENT OF CCRx n_{CCRx}	COMMENT
$-0.5 \times V_{CC}$	0	Most negative output voltage
$-0.25 \times V_{CC}$	$(n_{CCR0} + 1) \times 0.25$	Half negative output voltage
0 V	$(n_{CCR0} + 1) \times 0.5$	0 voltage
$0.25 \times V_{CC}$	$(n_{CCR0} + 1) \times 0.75$	Half positive output voltage
$0.5 \times V_{CC}$	$n_{CCR0} + 1$	Most positive output voltage

Example 6–42. Generation of Two PWM Output Signals

The software example shows the generation of two PWM output signals at the output terminals TA1 and TA2:

❏ **Output Pin TA1** — a positive PWM signal. The length of the active high part is defined in the RAM location TA1PWM (TCLK cycles).

❏ **Output Pin TA2** — a negative PWM signal. The length of the active low part is defined in the RAM location TA2PWM (TCLK cycles).

Additional tasks need to be executed by the interrupt handlers of the capture/compare blocks 1 and 2, therefore an individual handler is used for both of them.

The system clock frequency in use is 4 MHz (exactly $f_{MCLK} = 122 \times 32768 = 3.9977$ MHz), and the pulse repetition frequency is 19.2 kHz to allow the use of this frequency for other timings as well (a software UART with 4800 baud = 19.2 kHz/4, for example). For this application, bytes are sufficient for TA1PWM and TA2PWM because the maximum possible value of its content is 209. (4.0 MHz/19200 = 208.33).

The output unit 0 outputs 9600 Hz without any overhead. This signal may be used for peripherals or for synchronization — the signal is always present, even if the signals at TA1 and TA2 disappear due to an output signal with 0% or 100% pulse width.

The example uses the *Frequent Update by the Appropriate Interrupt Handler*. See Section 6.3.9.2 for details.
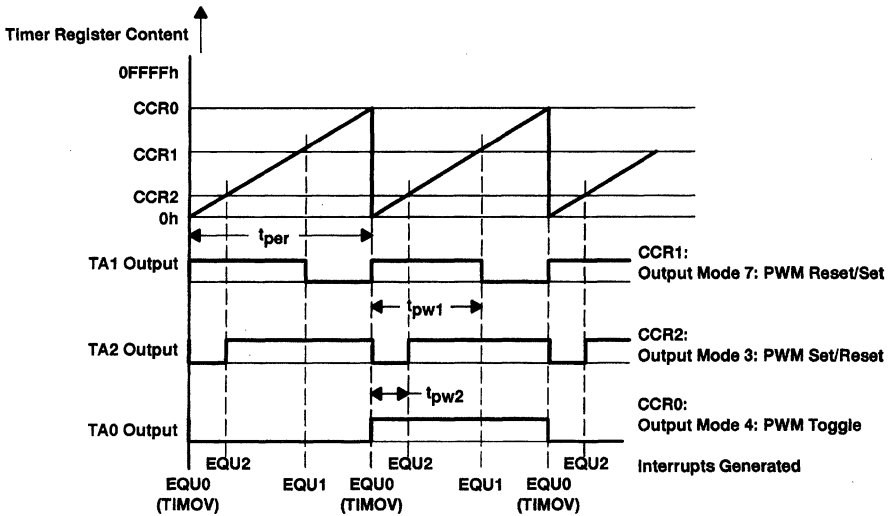
Figure 6–37. Three Different Asymmetric PWM-Timings Generated With the Up Mode

```
; Software example:
; TA0: symmetric output signal  9.6kHz
; TA1: positive PWM signal     19.2kHz. Length in TA1PWM
; TA2: negative PWM signal     19.2kHz. Length in TA2PWM
;
; Hardware definitions
;
FLLMPY   .equ    122                 ; FLL multiplier for 3.9977MHz
fper     .equ    19200               ; 19.2kHz repetition rate
TCLK     .equ    FLLMPY*32768        ; TCLK: FLLMPY x fcrystal
PERIOD   .equ    ((2*TCLK/fper)+1)/2 ; Period of output signals
STACK    .equ    600h                ; Stack initialization address
;
; RAM definitions
;
TA1PWM   .equ    202h                ; Pulse length Block 1 (0..209)
TA2PWM   .equ    203h                ; Pulse length Block 2 (0..209)
```

```
TIMACYC0 .equ      204h               ; Low cycle counter (bits 15..0)
TIMACYC1 .equ      206h               ; High cycle counter (31..16)
TIMACNT  .equ      208h               ; Counts # of periods
;
         .text                        ; Software start address
;
INIT     MOV       #STACK,SP          ; Initialize Stack Pointer
         CALL      #INITSR            ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Up Mode, INTRPTs on
;
         MOV       #ISMCLK+CLR,&TACTL ; Define Timer_A
         MOV       #PERIOD-1,&CCR0    ; Period to Period Register
         MOV       #0,&CCR1           ; TA1: pulse width = 0
         MOV       #0,&CCR2           ; TA2: pulse width = 0
         MOV       #OMT+CCIE,&CCTL0   ; TA0: Toggle Mode
         MOV       #OMRS+CCIE,&CCTL1  ; TA1: Reset/Set Mode
         MOV       #OMSR+CCIE,&CCTL2  ; TA2: Set/Reset Mode
         MOV.B     #TA2+TA1+TA0,&P3SEL ; Define Timer_A I/Os
         MOV.B     #CBMCLK+CBE,&CBCTL ; Output MCLK at XBUF pin
;
         CLR.B     TA1PWM             ; Start value Block 1: 0V
         CLR.B     TA2PWM             ; Start value Block 2: 0V
         CLR       TIMACYC0           ; Clear low cycle counter
         CLR       TIMACYC1           ; Clear high cycle counter
         CLR       TIMACNT            ; Clear period counter
         BIS       #MUP,&TACTL        ; Start Timer in Up Mode
         EINT                         ; Enable interrupts
MAINLOOP ...                          ; Continue in background
;
; Calculations resulted in new PWM values. The new results
; are stored in R6 (C/C Block 1) and R7.(C/C Block 2)
; Check if immediate update is necessary:
; CCRx = 0 .or. >= period.
;
         CMP       #PERIOD,&CCR1      ; CCR1 actually >= period?
```

```
          JHS     L$11            ; Yes, update CCR1 immediately
          TST     &CCR1           ; No, is CCR1 = 0?
          JNZ     L$12            ; CCR1 > 0: normal procedure
L$11      MOV     R6,&CCR1        ; No interrupt: immed. update
L$12      MOV.B   R6,TA1PWM       ; Actualize TA1PWM buffer
;
          CMP     #PERIOD,&CCR2   ; CCR2 actually >= period?
          JHS     L$21            ; Yes, update CCR2 immediately
          TST     &CCR2           ; No, CCR2 = 0?
          JNZ     L$22            ; CCR2 > 0: normal procedure
L$21      MOV     R7,&CCR2        ; No interrupt: immed. update
L$22      MOV.B   R7,TA2PWM       ; Actualize TA2PWM buffer
          ...                     ; Continue in background
;
; Interrupt handler for CCR0: the Period Register. The cycle
; counters and the period counter are updated.
; A symmetric 9.6kHz signal is output by the Output Unit 0
; Return from interrupt via the handler of C/C Blocks 1 to 4.
;
TIMMOD0   ADD     #PERIOD,TIMACYC0 ; Add (fixed) period to
          ADC     TIMACYC1        ; cycle counters
          INC     TIMACNT         ; Period counter +1
          ...                     ; Task0 (if any)
                                  ; Fall through to TIM_HND
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The actual interrupt flag CCIFGx is reset by the
; reading of the Timer Vector Register TAIV
;
TIM_HND   ADD     &TAIV,PC        ; Add Jump table offset
          RETI                    ; Vector 0: No interrupt pending
          JMP     TIMMOD1         ; Vector 2: Block 1
          JMP     TIMMOD2         ; Vector 4: Block 2
          JMP     TIMMOD3         ; Vector 6: Block 3 (not shown)
          JMP     TIMMOD4         ; Vector 8: Block 4 (not shown)
          RETI                    ; Vector 10: TIMOV not used
```

```
;
; Capture/Compare Block 1 outputs a positive PWM signal at TA1
; The pulse width is defined in TA1PWM (0..PERIOD)
;
TIMMOD1  MOV.B    TA1PWM,&CCR1     ; Pulse width to CCR1
         EINT                      ; Allow nested interrupts
         ...                       ; Task1 starts here
         RETI                      ; Back to main program
;
; Capture/Compare Block 2 outputs a negative PWM signal at TA2
; The pulse width is defined in TA2PWM (0..PERIOD)
;
TIMMOD2  MOV.B    TA2PWM,&CCR2     ; Pulse width to CCR2
         EINT                      ; Allow nested interrupts
         ...                       ; Task2 starts here
         RETI                      ; Back to main program
;
; The tasks for the C/C Blocks 3 and 4 are not shown
;
TIMMOD3  ...                       ; Handler for C/C Block 3
         RETI
;
TIMMOD4  ...                       ; Handler for C/C Block 4
         RETI
;
         .sect   "TIMVEC",0FFF0h  ; Timer_A Interrupt Vectors
         .word   TIM_HND          ; C/C Blocks 1 to 4
         .word   TIMMOD0          ; Capture/Compare Block 0
         .sect   "INITVEC",0FFFEh ; Reset Vector
         .word   INIT
```

The example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \ \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

Note:

The formula and the definitions given above are also valid for all subsequent software examples. They are therefore not repeated.

CCR0 — repetition rate 19.2kHz   13 cycles for the task, 14 cycles overhead   27 cycles

CCR1 — repetition rate 19.2kHz   6 cycles for the update, 17 cycles overhead   23 cycles

CCR2 — repetition rate 19.2kHz   6 cycles for the update, 17 cycles overhead   23 cycles

$$u_{CPU} = \frac{19200 \times (27 + 23 + 23)}{3.9977 \times 10^6} = 0.35$$

This result shows a CPU loading of 35% due to the Timer_A (the tasks of the capture/compare blocks 1 and 2 are not included).

6.3.9.4 Digital-to-Analog Conversion (DC Generation)

With the Timer_A running in the up mode, a maximum of four digital–to–analog converters (DACs) can be created. With appropriate external filters, dc output voltages are available.

The Figure 6–38 shows simple hardware solutions for cleaning up the output dc voltage. The ripple shown on the dc output voltages is exaggerated for explanation purposes.
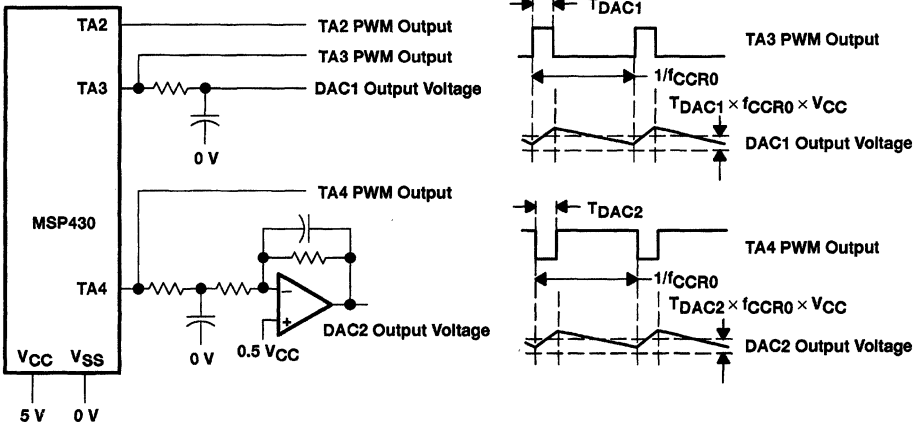
Figure 6–38. Digital-to-Analog Conversion

Example 6–43. Digital-to-Analog Conversion

This software example creates three DACs that are updated at individual times and relatively infrequently compared to the repetition rate defined by the period register (CCR0):

❏ **DAC0** — output TA2, positive output signal, output value stored in DAC0OV.

❏ **DAC1** — output TA3, positive output signal, output value stored in DAC1OV.

❏ **DAC2** — output TA4, negative output signal, output value stored in DAC2OV.

The higher the selected output frequency at the TAx outputs, the better the suppression of the ac part of the output signal is.

The interrupt is used only after a new PWM value is calculated and needs to be transferred to the capture/compare register. The update rate is approximately 500 Hz.

The repetition frequency for all three DAC outputs is 3.072 kHz, the system clock frequency selected is 3.1457 MHz. This results in 1024 different steps (10 bits resolution) for the DAC output voltages.

This example uses the *Infrequent Update by the Appropriate Interrupt Handler.* See Section 6.3.9.2 for details. The software is written for a variable period register.

```
; Software example: three independent DACs at TA2, TA3 and TA4
; Hardware definitions
;
FLLMPY    .equ    96                ; FLL multiplier for 3.1457MHz
fper      .equ    3072              ; 3.072kHz repetition rate
TCLK      .equ    FLLMPY*32768      ; TCLK: FLLMPY x fcrystal
PERIOD    .equ    ((2*TCLK/fper)+1)/2 ; Period of output signal
STACK     .equ    600h              ; Stack initialization address
;
; RAM definitions
;
DAC0OV    .equ    202h              ; Output value DAC0 (10 bits)
DAC1OV    .equ    204h              ; Output value DAC1 (10 bits)
```

```
DAC2OV   .equ    206h              ; Output value DAC2 (10 bits)
TIMACYC0 .equ    208h              ; Cycle counter low (bits 15..0)
TIMACYC1 .equ    20Ah              ; Cycle counter high (bits 31..16)
TIMACNT  .equ    20Ch              ; Period counter
FLAG     .equ    20Eh              ; Flag register for DACs
;
         .text                     ; Software start address
;
; Initialize the Timer_A: MCLK, Up Mode, CCR0 INTRPT enabled
; Prepare Timer_A Output Units, MCLK = 3.1457MHz
;
INIT     MOV     #STACK,SP         ; Initialize Stack Pointer SP
         CALL    #INITSR           ; Init. FLL and RAM
         MOV     #ISMCLK+CLR,&TACTL ; Define Timer_A
         MOV     #OMOO+CCIE,&CCTL0 ; Enable INTRPT Per. Reg.
         MOV     #OMRS,&CCTL2      ; DAC0: Reset/Set
         MOV     #OMRS,&CCTL3      ; DAC1: Reset/Set
         MOV     #OMSR,&CCTL4      ; DAC2: Set/Reset
         MOV     #PERIOD-1,&CCR0   ; Load Period Register
         CLR     &CCR2             ; DAC0: 0% output
         CLR     &CCR3             ; DAC1
         CLR     &CCR4             ; DAC2
         MOV.B   #TA4+TA3+TA2,&P3SEL ; Output Unit I/Os
         CLR     TIMACNT           ; Clear period counter
         CLR     TIMACYC0          ; Clear cycle counters
         CLR     TIMACYC1
         CLR.B   FLAG              ; Disable update of DACs
         BIS     #MUP,&TACTL       ; Start Timer_A with Up Mode
         EINT                      ; Enable interrupts
MAINLOOP ...                       ; Continue in background
;
; Calculations for the new DAC values start.
; The new results in R6 are written to DACxOV after completion
; The interrupt is enabled individually for the update. A check
; is made if special treatment is necessary:
; CCRx = 0 .or. >= period
```

```
; Software is written for a variable period register CCR0
;
                ...                  ; Calculate DAC0 value to R6
        MOV     R6,DAC0OV            ; Actualize DAC0 pulse length
        CMP     &CCR2,&CCR0          ; CCR2 >= period actually?
        JLO     L$21                 ; Yes, update CCR2 immediat.
        TST     &CCR2                ; No, is CCR2 = 0 actually?
        JNZ     L$22                 ; No, proceed normally
L$21    MOV     R6,&CCR2             ; Update CCR2 immediately
        JMP     L$23                 ; Update made, calc. CCR3 PWM
L$22    BIS.B   #1,FLAG              ; Set update flag for DAC0
        BIS     #CCIE,&CCTL2         ; Enable interrupt for DAC0
L$23    .equ    $
;
                ...                  ; Calculate DAC1 value to R6
        MOV     R6,DAC1OV            ; Actualize DAC1 pulse length
        CMP     &CCR3,&CCR0          ; See comment for DAC0
        JLO     L$31
        TST     &CCR3
        JNZ     L$32
L$31    MOV     R6,&CCR3
        JMP     L$33
L$32    BIS.B   #2,FLAG
        BIS     #CCIE,&CCTL3
L$33    .equ    $
;
                ...                  ; Calculate DAC2 value to R6
        MOV     R6,DAC2OV            ; Actualize DAC2 pulse length
        CMP     &CCR4,&CCR0          ; See comment for DAC0
        JLO     L$41
        TST     &CCR4
        JNZ     L$42
L$41    MOV     R6,&CCR4
        JMP     L$43
L$42    BIS.B   #4,FLAG
        BIS     #CCIE,&CCTL4
```

```
L$43     ....                   ; Continue in background
;
; Interrupt handler of the Period Register CCR0.
; A way is shown how to update the cycle counters if the
; timer period is variable during the program flow
;
TIMMOD0  SETC                   ; Period = (CCR0)+1
         ADDC   &CCR0,TIMACYC0  ; Add actual period to
         ADC    TIMACYC1        ; cycle counters TIMACYCx
         INC    TIMACNT         ; Period counter +1
         EINT                   ; Allow nested interrupts
         ...                    ; Task0 (if any)
         RETI
;
; Interrupt handler for Capture/Compare Registers 1 to 4
;
TIM_HND  ADD    &TAIV,PC        ; Serve highest priority requ.
         RETI                   ; No interrupt pending
         JMP    TIMMOD1         ; CCR1 request
         JMP    TIMMOD2         ; DAC0 request
         JMP    TIMMOD3         ; DAC1
         JMP    TIMMOD4         ; DAC2
         RETI                   ; Timer overflow disabled
;
; Capture/Compare Block 1 interrupt handler. May be used for
; comparison or capturing. Not implemented here.
;
TIMMOD1  ...                    ; Handler start
         RETI
;
; DACx updates. Interrupt is used only if a new result is
; calculated. Update frequency is 0.5kHz. The 1st interrupt
; is rejected, due to the always set interrupt flag CCIFGx.
; The 2nd synchronous interrupt updates the C/C Block.
;
TIMMOD2  BIT.B   #1,FLAG        ; Update possible? (flag = 0)
```

```
          JNZ      T20              ; No, asynchronous interrupt
          MOV      DAC0OV,&CCR2     ; Yes, update DAC0
          BIC      #CCIE,&CCTL2     ; Disable interrupt
          RETI
T20       BIC.B    #1,FLAG          ; Indicate update readiness
          RETI                      ; Return from interrupt
;
; DAC1 update. Same as above for DAC0
;
TIMMOD3   BIT.B    #2,FLAG          ; Update possible? (flag = 0)
          JNZ      T30              ; No, asynchronous interrupt
          MOV      DAC1OV,&CCR3     ; Yes, update DAC1
          BIC      #CCIE,&CCTL3     ; Disable interrupt
          RETI
T30       BIC.B    #2,FLAG          ; Indicate readiness
          RETI                      ; Return from interrupt
;
; DAC2 update. Same as above for DAC0
;
TIMMOD4   BIT.B    #4,FLAG          ; Update possible? (flag = 0)
          JNZ      T40              ; No, asynchronous interrupt
          MOV      DAC2OV,&CCR4     ; Yes, update DAC2
          BIC      #CCIE,&CCTL4     ; Disable interrupt
          RETI
T40       BIC.B    #4,FLAG          ; Indicate readiness
          RETI                      ; Return from interrupt

          .sect    "TIMVEC",0FFF0h  ; Timer_A Interrupt Vectors
          .word    TIM_HND          ; Vector for C/C Block 1..4
          .word    TIMMOD0          ; Vector for C/C Block 0
          .sect    "INITVEC",0FFFEh ; Reset Vector
          .word    INIT
```

This example results in a maximum CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities (maximum update frequencies on all three DAC channels):

CCR0 — repetition rate 3.072 kHz	16 cycles for the task, 11 cycles overhead	27 cycles
CCR2 — repetition rate 0.5 kHz	27 cycles for the update, 32 cycles overhead	59 cycles
CCR3 — repetition rate 0.5 kHz	27 cycles for the update, 32 cycles overhead	59 cycles
CCR4 — repetition rate 0.5 kHz	27 cycles for the update, 32 cycles overhead	59 cycles

$$u_{CPU} = \frac{3072 \times 27 + 500 \times (59 + 59 + 59)}{3.1457 \times 10^6} = 0.055$$

Note that an update for the capture/compare blocks 2 to 4 needs two interrupt services. The above result means a worst case CPU loading of approximate 5.5% due to the three DACs.

If all three tasks are updated with a 100 Hz update rate, then the CPU is loaded with only 3.2%.

6.3.9.5 TRIAC Control

TRIAC control for electric motors (DMC) or other loads is also possible with the up mode of the Timer_A. But the time frame, defined by the period register, does not allow the same resolution as with the continuous mode. The control software now counts the number of periods and fires the TRIAC after the reaching of the programmed number.

The medium resolution pmed is:

$$pmed = \frac{1}{2 \times f_{MAINS} \times tper}$$

Where:
f_{MAINS}	AC Line frequency	[Hz]
tper	Period of the Timer_A, defined by CCR0	[s]

The integrated energy E of a sine half wave dependent on the time t is described by the equation:

$$E \approx 1 - \cos \omega \ t = 1 - \cos 2\pi f t \qquad t = 0 \ ... \ \frac{1}{2f}$$

Due to this nonlinear energy increase, the worst case resolution p_{min} — near the angle $\pi/4$ (90°) — is reduced by a factor of $\pi/2$ (1.57) compared to the medium resolution p_{med}:

$$pmin = \frac{1}{2 \times f_{MAINS} \times tper \times \pi \ / \ 2}$$

The TRIAC control software contains fewer security features than the version shown for the continuous mode:

❑ The zero crossing part (P0.0 handler) immediately switches the gate signal off by setting terminal TA0 to high. This prevents the firing for the next half wave.

This means, the background software has to check to see if the calculated time for the firing of the TRIAC — the number of timer periods after the zero crossing of the ac line voltage (FIRANGL) — is not too near to the next zero crossing.

No capture/compare register is needed for the TRIAC control — only the period register with its interrupt and output unit 0 is used. This frees the remaining capture/compare blocks for other tasks.

Figure 6–39 shows the hardware for the TRIAC control of this example. After power up, the TA0 terminal is switched to input mode — the base resistor of the PNP transistor switches the gate of the TRIAC off and prevents a run of the motor. The necessary hardware debounce for the zero crossing signal at P0.0 is made with the internal capacity, Cz, of the zener diode.
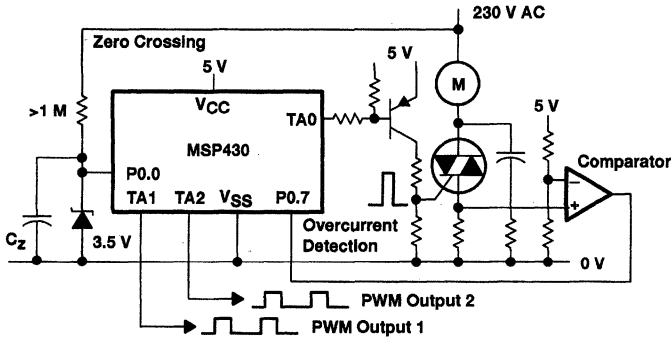
Figure 6–39. TRIAC Control with Timer_A

Figure 6–40 illustrates the software example given below. The period of CCR0 is not shown in to scale — 160 steps make one half wave of the 50 Hz line.
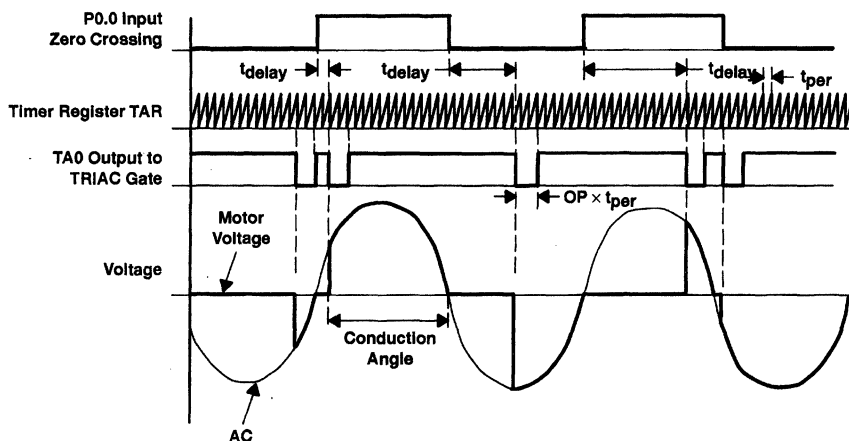
Figure 6–40. Signals for the TRIAC Gate Control With Up Mode

Example 6–44. Static TRIAC Control

A static TRIAC control software example is shown. The calculated number of periods until the TRIAC gate is fired after the zero crossing of the ac line voltage, is contained in the RAM word FIRANGL.

The medium resolution p_{med} is 160 steps per 50 Hz line half wave (16 kHz/100 Hz = 160). The minimum resolution, p_{min}, is 102 steps ($160 \times 2/\pi$ = 102), which means approx. 1% resolution. See the equations above.

At the TA1 and TA2 terminals, positive PWM signals are output. The period is defined by the Period register CCR0, the pulse length (TCLK cycles) is contained in the RAM bytes TA1PWM and TA2PWM. The update is made with 1 kHz.

The example uses the *Infrequent Update by the Appropriate Interrupt Handler*. See Section 6.3.9.2 for details.

```
; Definitions for the TRIAC control software
;
FLLMPY   .equ   64                 ; FLL multiplier for 2.096MHz
fper     .equ   16000              ; 16.000kHz repetition rate
TCLK     .equ   FLLMPY*32768       ; TCLK (Timer Clock) [Hz]
PERIOD   .equ   ((2*TCLK/fper)+1)/2 ; Period in Timer clocks
OP       .equ   2                  ; TRIAC gate pulse length (per.)
```

```
;
; RAM definitions
;
TIMACYC0 .equ    202h            ; Timer Register Extensions:
TIMACYC1 .equ    204h            ; Cycle counters
TIMACNT  .equ    206h            ; Counter of periods
FIRANGL  .equ    208h            ; Half wave - conduction angle
FIRTIM   .equ    20Ah            ; Fire time rel. to TIMACNT
TA1PWM   .equ    20Ch            ; PWM cycle count for Block 1
TA2PWM   .equ    20Dh            ; PWM cycle count for Block 2
STTRIAC  .equ    20Eh            ; Control byte (0 = off) Status
FLAG     .equ    20Fh            ; 1: update for PWM necessary
STACK    .equ    600h            ; Stack initialization address
         .text                   ; Start of ROM code
;
; Initialize the Timer_A: MCLK, Up Mode, INTRPT enabled
; Prepare Timer_A Output Units
;
INIT     MOV     #STACK,SP       ; Initialize Stack Pointer SP
         CALL    #INITSR         ; Init. FLL and RAM
         MOV     #ISMCLK+CLR,&TACTL ; Init. Timer
         MOV     #PERIOD-1,&CCR0  ; Period to CCR0
         MOV     #OMOO+CCIE+OUT,&CCTL0 ; Set TA0 high
         MOV     #OMRS,&CCTL1    ; TA1: pos. PWM pulses
         MOV     #OMRS,&CCTL2    ; TA2: pos. PWM pulses
         BIS.B   #TA2+TA1+TA0,&P3SEL ; Define timer outputs
         BIS.B   #P0IE0,&IE1     ; Enable P0.0 interrupt (mains)
         CLR     TIMACYC0        ; Clear low cycle counter
         CLR     TIMACYC1        ; Clear high cycle counter
         CLR     TIMACNT         ; Clear period counter
         CLR.B   STTRIAC         ; TRIAC off status (0)
         CLR.B   FLAG            ; No update
         CLR.B   TA1PWM          ; TA1: no output (0% duty cycle)
         CLR.B   TA2PWM          ; TA2: no output (0% duty cycle)
         BIS     #MUP,&TACTL     ; Start Timer_A in Up Mode
         MOV.B   #CBMCLK+CBE,&CBCTL      ; MCLK at XBUF pin
```

```
        EINT                    ; Enable interrupts
MAINLOOP  ...                   ; Continue in mainloop
;
; Some TRIAC control examples:
; Start electric motor: checked result (Timer_A periods) in R5
; The result is the time difference from the zero crossing
; of the mains voltage (P0.0) to the first gate pulse
; (measured in Timer_A periods)
;
        MOV     R5,FIRANGL      ; Delay (periods) to FIRANGL
        MOV.B   #2,STTRIAC      ; Activate TRIAC control
        ...                     ; Continue in background
;
; The motor is running. A new calculation result is available
; in R5. It will be used with the next mains half wave
;
        MOV     R5,FIRANGL      ; Delay (periods) to FIRANGL
        ...                     ; Continue in background
;
; Stop motor: switch off TRIAC control
;
        CLR.B   STTRIAC         ; Disable TRIAC control
        BIC     #OMRS,&CCTL0    ; TRIAC gate off
        BIS     #CCIE+OUT,CCTL0 ; TA0 high, Output only Mode
        ...                     ; Continue with background
;
; Calculations for the new PWM values start.
; The new results in R6 are written to TAxPWM after completion
; The interrupt is enabled individually for the update. A check
; is made if special treatment is necessary:
; Actual CCRx = 0 .or. >= period
; Software is written for a constant period register CCR0
;
        ...                     ; Calculate TA1PWM value to R6
        MOV.B   R6,TA1PWM       ; Actualize pulse length
        CMP     #PERIOD,&CCR1   ; CCR1 >= period actually?
```

```
        JHS     L$11            ; Yes, update CCR1 immediat.
        TST     &CCR1           ; No, is CCR1 = 0 actually?
        JNZ     L$12            ; No, proceed normally
L$11    MOV     R6,&CCR1        ; Update CCR1 immediately
        JMP     L$13            ; Update made, calc. next PWM
L$12    BIS.B   #1,FLAG         ; Set update flag for TA1PWM
        BIS     #CCIE,&CCTL1    ; Enable interrupt
L$13    .equ    $
;

        ...                     ; Calculate TA2PWM value to R6
        MOV.B   R6,TA2PWM       ; Actualize pulse length
        CMP     #PERIOD,&CCR2   ; CCR2 >= period actually?
        JHS     L$21            ; Yes, update CCR2 immediat.
        TST     &CCR2           ; No, is CCR2 = 0 actually?
        JNZ     L$22            ; No, proceed normally
L$21    MOV     R6,&CCR2        ; Update CCR2 immediately
        JMP     L$23            ; Update made, continue
L$22    BIS.B   #2,FLAG         ; Set update flag for TA2PWM
        BIS     #CCIE,&CCTL2    ; Enable interrupt for DAC0
L$23    ....                    ; Continue in background
;
;
; Interrupt handler for CCR0: the Period Register:
; - The cycle counters and the period counter are updated:
; - The TRIAC control task is executed
;
TIMMOD0 ADD     #PERIOD,TIMACYC0 ; Add (fixed) period to
        ADC     TIMACYC1        ; Cycle counters
        INC     TIMACNT         ; Increment period counter
;
; Interrupt handler for the TRIAC control
;
        EINT                    ; Allow nested interrupts
        PUSH    R5              ; Save help register R5
        MOV.B   STTRIAC,R5      ; Status of TRIAC control
        MOV     STTAB(R5),PC    ; Branch to status handler
```

```
STTAB    .word    STATE0           ; Status 0: No TRIAC activity
         .word    STATE0           ; Status 2: activation possible
         .word    STATE4           ; Status 4: wait for gate pulse
         .word    STATE6           ; Status 6: wait for gate off
;
; TRIAC status 4: gate is switched on for OP periods after the
; value in FIRTIM is reached
;
STATE4   CMP      FIRTIM,TIMACNT   ; TRIAC gate time reached?
         JNE      STATE0           ; No
         BIS      #OMR+CCIE,&CCTL0 ; Prepare for gate on pulse
         ADD.B    #2,STTRIAC       ; Next TRIAC status (6)
;
; TRIAC status 0: No activity. TRIAC is off always
;
STATE0   POP      R5               ; Restore help register
         RETI                      ; Return from interrupt
;
; TRIAC status 6: gate pulse is active. Check if it's time
; to switch the gate off.
STATE6   MOV      FIRTIM,R5
         ADD      #OP,R5           ; Gate-on time (periods)
         CMP      R5,TIMACNT       ; On-time terminated?
         JLO      STATE0           ; No
         BIC      #OMRS,&CCTL0     ; Yes, prepare TRIAC Gate off
         BIS      #OMSET+CCIE,&CCTL0;
         MOV.B    #2,STTRIAC       ; TRIAC status:
         JMP      STATE0           ; Wait for next zero crossing
;
; Interrupt handler for Capture/Compare Blocks 1 to 4
;
TIM_HND  ADD      &TAIV,PC         ; Serve highest priority requ.
         RETI                      ; No interrupt pending
         JMP      TIMMOD1          ; PWM 1 request
         JMP      TIMMOD2          ; PWM 2 request
         RETI                      ; Not used
```

```
        RETI                        ; Not used
        RETI                        ; Timer overflow disabled
;
; C/C Block updates. Interrupt is used only if a new result is
; calculated. Update frequency is 1.0kHz. The 1st interrupt
; is rejected, due to the always set interrupt flag CCIFGx.
; The 2nd synchronous interrupt updates the C/C Register.
;
TIMMOD1 BIT.B   #1,FLAG             ; Update possible? (flag = 0)
        JNZ     T10                 ; No, asynchronous interrupt
        MOV.B   TA1PWM,&CCR1        ; Yes, update C/C Block 1
        BIC     #CCIE,&CCTL1        ; Disable interrupt
        RETI
T10     BIC.B   #1,FLAG             ; Indicate: ready for update
        RETI                        ; Return from interrupt
TIMMOD2 BIT.B   #2,FLAG             ; Update possible? (flag = 0)
        JNZ     T20                 ; No, asynchronous interrupt
        MOV.B   TA2PWM,&CCR2        ; Yes, update C/C Block 2
        BIC     #CCIE,&CCTL2        ; Disable interrupt
        RETI
T20     BIC.B   #2,FLAG             ; Indicate: ready for update
        RETI                        ; Return from interrupt
;
; P0.0 Handler: the mains voltage causes interrupt with each
; zero crossing. The TRIAC gate is switched off first, to
; avoid the ignition for the actual half wave.
; Hardware debounce is necessary for the mains signal!
;
P00_HNDLR BIC   #OMRS,&CCTL0        ; Switch off TRIAC gate
        BIS     #CCIE+OUT,&CCTL0
        EINT                        ; Allow nested interrupts
        XOR.B   #1,&P0IES           ; Change interrupt edge of P0.0
;
; If STTRIAC is not 0 ( 0 = inactivity) then the next gate
; firing is prepared: STTRIAC is set to 4
;
```

```
            TST.B    STTRIAC
            JZ       P00           ; STTRIAC = 0: no activity
            MOV.B    #4,STTRIAC    ; STTRIAC > 0: prep. next firing
;
; The TRIAC firing time is calculated: TIMACNT + FIRANGL
;
            MOV      TIMACNT,FIRTIM   ; Period counter
            ADD      FIRANGL,FIRTIM   ; TIMACNT + delay -> FIRTIM
P00         RETI
;
            .sect    "TIMVEC",0FFF0h  ; Timer_A Interrupt Vectors
            .word    TIM_HND          ; C/C Blocks 1..4 Vector
            .word    TIMMOD0          ; Vector for C/C Block 0
            .sect    "P00VEC",0FFFAh  ; P0.0 Vector
            .word    P00_HNDLR
            .sect    "INITVEC",0FFFEh ; Reset Vector
            .word    INIT
```

The TRIAC control example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) for the active TRIAC control (STTRIAC = 4):

CCR0 — repetition rate 16 kHz	32cycles for the task, 11 cycles overhead	43 cycles
CCR1 — repetition rate 1 kHz	27 cycles for the update, 32 cycles overhead	59 cycles
CCR2 — repetition rate 1 kHz	27 cycles for the update, 32 cycles overhead	59 cycles
P0.0 — repetition rate 100 Hz	32 cycles for the task, 11 cycles overhead	43 cycles

$$u_{CPU} = \frac{16.0 \times 10^3 \times 43 + 1.0 \times 10^3 \times (59+59) + 100 \times 47}{2.096 \times 10^6} = 0.39$$

The above result means a CPU loading of approximate 39% due to the static TRIAC control. The necessary tasks for the update of the period counter and the cycle counters are included. The PWM activities alone load the CPU with less than 6% using this method (f_{update} = 1 kHz).

6.3.9.6 RF Timing Generation

The repetition rate used in the up mode must be a multiple of the data change frequency. The three different modulation methods and its conversion subroutines were explained in depth in the section RF generation.

The RF modulation modes described earlier were:

☐ **Amplitude Modulation** — the RF oscillator is switched on for a logical *1* and switched off for a logical *0* (100% modulation).

☐ **Biphase Code** — the information is represented by a bit time consisting of one half bit without modulation and one half bit with full modulation. A logical *1* starts with 100% modulation, a logical *0* starts with no modulation.

☐ **Biphase Space** — a logical *1* (space) is represented by a constant signal (100% or 0% modulation) during the complete bit time. A logical *0* (mark) changes the signal in the middle of the bit time. The signal changes after each transmitted bit. This means, the previous bit influences the current bit.

Figure 6–41 shows the three different modulation modes for an input byte containing the value 96h.
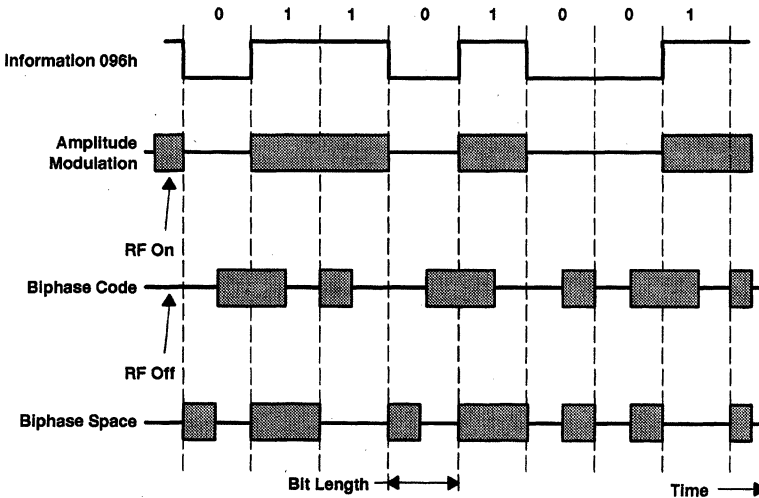
Figure 6–41. RF Modulation Modes

The capture/compare block 0 is used with the software example due to two facts:

❏ The fastest possible response. Decision making with the timer vector register is not necessary for the capture/compare block 0 — it uses its own, dedicated interrupt vector. The vector address is 0FFF2h.

❏ The capture/compare block 0 delivers the necessary timing anyway. The use of the period register therefore frees the remaining capture/compare registers for other tasks.

Example 6–45. RF Modulation Modes

The real time task common to all three modulation modes is given below. The background software prepares a 128-bit block starting at address RF_BLK, containing the information to be output in the desired coding format. This 128-bit buffer is output in real time with the same handler for all three modulation modes.

The selected half-bit repetition frequency is 19200 Hz, because 38400 Hz is too high a PWM frequency (increased switching losses, too few resolution steps).

The capture/compare blocks 1 to 3 are used for the PWM generation. The table processing used allows (nearly) simultaneous update of all three capture/compare blocks. The method used is the fastest one for updating. The number range for the PWM is from 1 to (period–1), therefore, the fast update — without range checks — is possible.

The CPU registers R5 and R8 are reserved for the RF timing. R5 contains the data to be output currently, R8 points to the next data word. They must not be overwritten by other tasks.

The conversion subroutines for the biphase code and biphase space modulation are described in the section *RF Generation.*

The example uses the *Frequent Update by the Appropriate Interrupt Handler.* See Section 6.3.9.2 for details.

```
; Hardware definitions
;
FLLMPY   .equ    122                  ; FLL multiplier for 3.998MHz
fRF      .equ    19200                ; Half-bit rep. frequency
TCLK     .equ    FLLMPY*32768         ; TCLK: FLLMPY x fcrystal
PERIOD   .equ    ((2*TCLK/fRF)+1)/2   ; Bit length (TCLK cycles)
```

```
STACK     .equ    600h            ; Stack initialization address
;
; RAM Definitions
;
RF_BLK    .equ    202h            ; Converted data 128 bits
TIMACYC   .equ    212h            ; Cycle counter Timer_A
TIMACNT   .equ    214h            ; Period counter Timer_A
RFSTAT    .equ    216h            ; Status of RF transmission
TA1PWM    .equ    217h            ; Value for C/C Block 1
TA2PWM    .equ    218h            ; Value for C/C Block 2
TA3PWM    .equ    219h            ; Value for C/C Block 3
          .text                         ; Software start address
;
INIT      MOV     #STACK,SP       ; Initialize Stack Pointer
          CALL    #INITSR         ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Up Mode, INTRPT on for CCRx
;
          MOV     #ISMCLK+CLR,&TACTL ; MCLK, TIMOV off
          MOV     #OMOO+CCIE,&CCTL0 ; Reset TA0, INTRPT on
          MOV     #OMSR+CCIE,&CCTL1 ; TA1: Set/Reset
          MOV     #OMSR+CCIE,&CCTL2 ; TA2: Set/Reset
          MOV     #OMSR+CCIE,&CCTL3 ; TA3: Set/Reset
          MOV     #PERIOD-1,&CCR0  ; 19.2kHz period
          MOV.B   #1,&CCR1          ; Minimum PWM length
          MOV.B   #1,&CCR2
          MOV.B   #1,&CCR3
          MOV.B   #TA3+TA2+TA1+TA0,&P3SEL ; Define timer outputs
;
          CLR     TIMACYC         ; Clear cycle counter
          CLR     TIMACNT         ; Clear period counter
          MOV.B   #1,TA1PWM       ; Minimum PWM output
          MOV.B   #1,TA2PWM       ;
          MOV.B   #1,TA3PWM       ;
          MOV.B   #CBMCLK,&CBCTL  ; Output MCLK at XBUF pin
          CLR.B   RFSTAT          ; RF status = 0
```

```
          BIS     #MUP,&TACTL      ; Start Timer in Up Mode
          EINT                     ; Enable interrupts
MAINLOOP ...                       ; Continue in background
;
; The data to be transmitted by RF is converted into a
; 128-bit RAM block starting at address RF_BLK with the
; appropriate conversion routine. The subroutines described
; in Part III are used. See there for explanation.
; R5 and R8 are reserved for the RF transmission!
;
          MOV.B   ADDRESS0,R6      ; 1st data byte to R6
          CALL    #BI_PHASE_xxx    ; Convert it to 16 bits
          MOV     R5,0(R8)         ; Converted data to RF-Block
          ...                      ; Continue with converting
;
; Initialize transmission of the converted data (128-bit)
;
          MOV     #RF_BLK,R8       ; Start of 128-bit block
          MOV     @R8+,R5          ; 1st 16 bits for output to R5
          MOV.B   #16+1,RFSTAT     ; Bit count for 1st 16 bits
          ...                      ; Continue in background
;
; Test in background if 128 bits are output: RFSTAT = 0
;
          TST.B   RFSTAT           ; Output completed?
          JZ      BPC_MADE         ; Yes, RFSTAT = 0
          ...                      ; No, continue
;
; New values for the three PWM channels are read from a table
;
          MOV     ANGLE,R15        ; Actual angle for DMC
          MOV.B   TABLE+00(R15),TA1PWM ; Update PWM channels
          MOV.B   TABLE+12(R15),TA2PWM ; out of a sine table
          MOV.B   TABLE+24(R15),TA3PWM ;
          ...                      ; Continue in background
;
```

```
; A second example is given for a 64 bit block:
; Initialize transmission of only 64 bits: the start address
; differs, the end address is again RF_BLK+16
;
         MOV     #RF_BLK+8,R8      ; Start of 64-bit block
         MOV     @R8+,R5          ; 1st 16 bits for output to R5
         MOV.B   #16+1,RFSTAT     ; Bit count for 1st 16 bits
         ...                      ; Continue in background
;
TABLE    .byte   1,15,29,43...PERIOD-1 ; PWM table
;
; Interrupt handler for Capture/Compare Block 0 (CCR0)
; Data in RF_BLK is output: LSB first.
; The Output Unit outputs the data bit prepared during the last
; period. The data bit for the next period is prepared now.
; Output is completed, when (last word +4) is addressed by R8.
;
TIMMOD0  ADD     #PERIOD,TIMACYC ; Add period to cycle counter
         INC     TIMACNT          ; Increment period counter
;
         EINT                     ; Allow nested interrupts
         TST.B   RFSTAT           ; RF transmission underway?
         JZ      TM03             ; No, return from interrupt
         DEC.B   RFSTAT           ; Yes, bit count - 1
         JNZ     TM01             ; Not zero: continue
         MOV     @R8+,R5          ; Next 16 bits for output
         CMP     #RF_BLK+18,R8    ; End of buffer+2 reached?
         JHS     TM04             ; Yes, finish output (RFSTAT=0)
         MOV.B   #16,RFSTAT       ; Bit count for next word
;
TM01     RRC     R5               ; Next data bit to carry
         JC      TM02             ; Bit is one
TM04     MOV     #OMR+CCIE,&CCTL0 ; Bit is 0: prepare reset
         RETI
;
TM02     MOV     #OMSET+CCIE,&CCTL0 ; Bit is 1: prepare set
```

```
TM03      RETI
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The actual interrupt flag CCIFGx is reset by the
; reading of the Timer Vector Register TAIV
;
TIM_HND   ADD     &TAIV,PC        ; Add Jump table offset
          RETI                    ; Vector 0: No interrupt pending
          JMP     TIMMOD1         ; Vector 2: Block 1
          JMP     TIMMOD2         ; Vector 4: Block 2
          JMP     TIMMOD3         ; Vector 6: Block 3
          RETI                    ; Vector 8: Block 4 (not used)
          .RETI                   ; Vector 10: TIMOV not used
;
; Capture/Compare Block 1 outputs a positive PWM signal at TA1
; The pulse width is defined in TA1PWM (1..PERIOD-1)
;
TIMMOD1   MOV.B   TA1PWM,&CCR1    ; Pulse width to CCR1
          RETI                    ; Back to main program
;
; Capture/Compare Block 2 outputs a positive PWM signal at TA2
; The pulse width is defined in TA2PWM (1..PERIOD-1)
;
TIMMOD2   MOV.B   TA2PWM,&CCR2    ; Pulse width to CCR2
          RETI                    ; Back to main program
;
; Capture/Compare Block 3 outputs a positive PWM signal at TA3
; The pulse width is defined in TA3PWM (1..PERIOD-1)
;
TIMMOD3   MOV.B   TA3PWM,&CCR3    ; Pulse width to CCR3
          RETI                    ; Back to main program
          .sect   "TIMVEC",0FFF0h ; Timer_A Interrupt Vector
          .word   TIM_HND         ; Vector C/C Blocks 1 to 3
          .word   TIMMOD0         ; Vector for C/C Block 0
          .sect   "INITVEC",0FFFEh ; Reset Vector
          .word   INIT
```

The RF timing generation example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) for the active transmit (RFSTAT > 0):

CCR0 — repetition rate 19.2 kHz	30 cycles for the task, 11 cycles overhead	41 cycles
CCR1 — repetition rate 19.2 kHz	6 cycles for the update, 16 cycles overhead	22 cycles
CCR2 — repetition rate 19.2 kHz	6 cycles for the update, 16 cycles overhead	22 cycles
CCR3 — repetition rate 19.2 kHz	6 cycles for the update, 16 cycles overhead	22 cycles

The above example results in a medium CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \sum \left(n_{intrpt} \times f_{rep} \right) = \frac{19.2 \times 10^{3} \times \left(41 + 22 + 22 + 22 \right)}{3.998 \times 10^{6}} = 0.51$$

The result means that the MSP430 CPU is loaded 20% when outputting the RF buffer with 19200 baud and an MCLK frequency of 4 MHz. The updates of the cycle counters and the period counter are included. The update of the PWM registers adds 31%, if used.

6.3.9.7 Software UART

With a carefully chosen timer period, a software UART can be implemented relatively simply. The complete software, a status-controlled handler, will be the topic of an external application report. This report will describe a full-duplex UART controlled by the timing of Timer_A.

6.3.9.8 Comparison With the Up Mode

Comparison with the up mode is made the same way as described in the section *Applications exceeding the 16-Bit Range of the Timer_A* for the continuous mode. As in that case, the timings to be created exceed the period of the timer register and external RAM extensions are therefore necessary.

6.3.9.9 Capturing With the Up Mode

If the periods of the internal interrupt timings or the time intervals to be captured are longer than one period of the timer register, then a special method is necessary to take care of the longer time periods. The same is true if a half period of a generated output frequency is longer than the period of the Timer_A.

This special method, with the use of extension registers for the capture/compare registers, is necessary if:

$$t_{SIGNAL} > \frac{\left(n_{ccr0} + 1 \right) \times k}{f_{CLK}}$$

Where:

t_{SIGNAL}	Time interval to be captured	[s]
f_{CLK}	Input frequency at the input divider input of Timer_A	[Hz]
k	Pre–divider constant of the input divider (1, 2, 4 or 8)	
n_{CCR0}	Content of period register CCR0	

Figure 6–42 illustrates the hardware and RAM registers used with the capture mode if the captured values are greater than one period of the Timer_A.
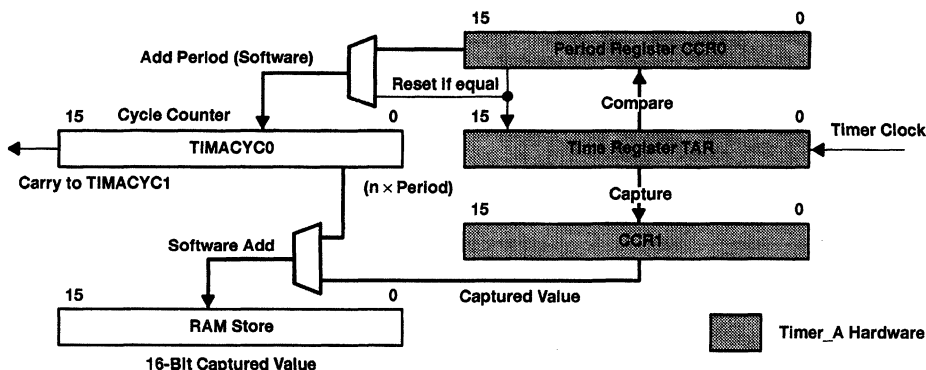
Figure 6–42. Capture Mode with the Up Mode (shown for CCR1)

Figure 6–43 illustrates five examples. The tasks are defined as follows:

❏ **Capture/Compare Block 0** — outputs a symmetrical 9.6 kHz signal. The edges contain the information for the period generated by the period register (CCR0). This signal is always available (the PWM signals of the capture/compare blocks disappear for pulse widths of 0% and 100%).

❏ **Capture/Compare Block 1** — generates a positive PWM signal with the period defined by the period register. The pulse length is stored in the RAM word TA1PWM. A dedicated interrupt handler is used.

❏ **Capture/Compare Block 2** — the length, Δt2, of the high part of the input signal at the CCI2A input terminal is measured and stored in the RAM word PP2. The captured time of the leading edge is stored in the RAM word TIM2. The max. repetition rate used is 2 kHz.

❏ **Capture/Compare Block 3** — the event time of the leading edge of the signal at the CCI3A input terminal is captured. The last captured value is stored in the RAM word TIM3. The max. repetition rate used is 3 kHz.

❏ **Capture/Compare Block 4** — generates a negative PWM signal with the period defined by the period register. The pulse length is stored in the RAM

word TA4PWM. Update is made with the interrupt handler of capture/compare block 0.

For the example, 3.801 MHz is used. The resolution is 224 steps due to the repetition frequency of 16.969 kHz (3.801 MHz/16.969 kHz = 224).

Table 6–23.   Short Description of the Capture and PWM Mix

C/C BLOCK	TIME INTERVAL	TIMER I/Os	COMMENT
0	Doubled period	Outputs 0.5 × PWM Frequency	Period register CCR0. Output of a symmetrical 8.484 kHz signal
1	Period	Outputs PWM 1 .. PERIOD–1	Generation of PWM. Pulse length stored in TA1PWM. Dedicated interrupt handler for update.
2	External event	Input pin CCI2A is used	Measures high signal part Δt2. Length of positive signal part is stored in PP2. Maximum input frequency is 2 kHz.
3	External event	Input pin CCI3A is used	Captures event time t3 of the trailing edge of the input signal. Event time t3 stored in TIM3. Maximum input frequency is 3 kHz.
4	Period	Outputs PWM 0 –100%	Generation of PWM. Pulse length stored in TA4PWM. Update by capture/compare block 0.

Note:

The maximum input frequencies for capturing purposes shown above are used for the overhead calculation only. The limits of the Timer_A hardware allow it to capture much higher input frequencies.

Figure 6–43 illustrates the four tasks described above — they are not shown to scale.

| Period Counter TIMACNT
Cycle Counter TIMACYC | n
n × Period | n+1
(n+1) × Period | n+2
(n+2) × Period | n+3
(n+3) × Period |
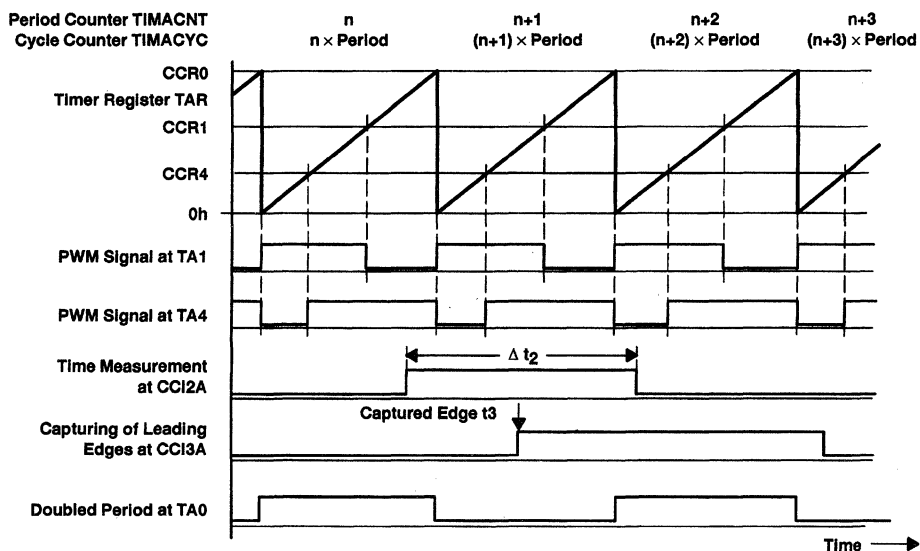
Figure 6–43. PWM Generation and Capturing With the Up Mode

Example 6–46. PWM Generation and Capturing With the Up Mode

```
; Timer_A used for PWM–generation and Capturing.
;
FLLMPY    .equ    116               ; fMCLK = 3.801MHz
fper      .equ    16969             ; 16.969kHz repetition rate
TCLK      .equ    FLLMPY*32768      ; TCLK: FLLMPY x fcrystal
PERIOD    .equ    ((2*TCLK/fper)+1)/2 ; frep = 19.969kHz
;
; RAM Definitions
;
TA1PWM    .equ    202h              ; PWM pulse length TA1
TIM2      .equ    204h              ; Time of leading edge at CCI2A
PP2       .equ    206h              ; Length of high part at CCI2A
TIM3      .equ    208h              ; Time of leading edge at CCI3A
TA4PWM    .equ    20Ah              ; PWM pulse length for TA4
TIMACYC0  .equ    20Ch              ; Cycle counter low
TIMACYC1  .equ    20Eh              ; Cycle counter high
```

```
TIMACNT  .equ    210h            ; Period counter
STACK    .equ    600h            ; Stack initialization address
         .text                   ; Software start address
INIT     MOV     #STACK,SP       ; Initialize Stack Pointer
         CALL    #INITSR         ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Up Mode, INTRPTs on
;
         MOV     #ISMCLK+CLR,&TACTL ; No TIMOV interrupt
         MOV     #PERIOD-1,&CCR0   ; Define period
         MOV     #OMT+CCIE,&CCTL0  ; Toggle TA0, INTRPT on
         MOV     #OMRS+CCIE,&CCTL1 ; Reset/Set Mode, INTRPT on
         MOV     #CMBE+ISCCIA+SCS+CAP+CCIE,&CCTL2 ; Both edges
         MOV     #CMPE+ISCCIA+SCS+CAP+CCIE,&CCTL3 ; Pos. edge
         MOV     #OMSR,&CCTL4      ; Set/Reset Mode, no INTRPT
         MOV.B   #TA4+TA3+TA2+TA1+TA0,&P3SEL ; Define I/Os
;
         CLR     TIMACYC0          ; Clear low cycle counter
         CLR     TIMACYC1          ; Clear high cycle counter
         CLR     TIMACNT           ; Clear period counter
         MOV     #1,TA1PWM         ; TA1 pulse length = 1
         MOV     #0,TA4PWM         ; TA4 pulse length = 0
         MOV.B   #CBACLK+CBE,&CBCTL ; Output ACLK at XBUF pin
         BIS     #MUP,&TACTL       ; Start Timer in Up Mode
         EINT                      ; Enable interrupts
MAINLOOP ...                       ; Continue in background
; Calculations for the new PWM values start.
; The new result in R6 is written to TA1PWM after completion.
; The PWM range is from 1 to PERIOD-1: no checks necessary
;
         ...                       ; Calculate TA1PWM value to R6
         MOV.B   R6,TA1PWM         ; Actualize pulse length
;
; The new result in R6 is written to TA4PWM after completion.
; The PWM range is from 0% to 100%: no checks necessary
;
```

```
        ...                        ; Calculate TA4PWM value to R6
        MOV.B    R6,TA4PWM         ; Actualize pulse length
        ...                        ; Continue in background
;
; Interrupt handler for the Period Register CCR0. 8.484kHz
; are output at TA0 for synchronization.
;
TIMMOD0 MOV      TA4PWM,&CCR4      ; Update CCR4
        ADD      #PERIOD,TIMACYC0  ; Actualize cycle counters
        ADC      TIMACYC1
        INC      TIMACNT           ; Incr. period counter
        RETI
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
TIM_HND ADD      &TAIV,PC          ; Add Jump table offset
        RETI                       ; Vector 0: No interrupt pending
        JMP      TIMMOD1           ; Vector 2: Block 1
        JMP      TIMMOD2           ; Vector 4: Block 2
        JMP      TIMMOD3           ; Vector 6: Block 3
        RETI                       ; Update by C/C Block 0
        RETI                       ; Vector 10: TIMOV not used
;
; Capture/Compare Block 1 generates a positive PWM signal at
; output TA1. Pulse width is defined in TA1PWM
;
TIMMOD1 MOV      TA1PWM,&CCR1      ; Define pulse width
        EINT                       ; Allow nested interrupts
        ...                        ; Task1 starts here
        RETI
;
; The high part of the CCI2A input signal is measured.
; The result is stored in PP2. The complete handler is time
; critical: nested interrupts cannot be used.
```

```
; First a check is made if the cycle counter TIMACYC0 contains
; the value corresponding to the captured value in CCR2, or if
; TIMACYC0 is yet updated due to interrupt latency time.
;
TIMMOD2  CMP    &CCR2,&TAR     ; Occurred overflow of TAR?
         JHS    TM20           ; No, Timer Reg. > capt. value
         BIT    #CCIFG,&CCTL0   ; Yes, TIMACYC0 yet updated?
         JNZ    TM20           ; No, value matches with CCR2
         SUB    #PERIOD,&CCR2   ; Yes, use CCR2 for correction
;
TM20     BIT    #CCI,&CCTL2     ; Input signal high?
         JZ     TM21           ; No, time for calculation
         MOV    TIMACYC0,TIM2   ; Yes, store cycle counter
         ADD    &CCR2,TIM2     ; Time for leading edge in TIM2
         RETI
;                              ; High part is calculated:
TM21     MOV    TIMACYC0,PP2    ; Event time of trailing edge
         ADD    &CCR2,PP2      ; Add captured time
         SUB    TIM2,PP2       ; Subtr. time of leading edge
         RETI                   ; Length of high part in PP2
;
; Capture/Compare Block 3 captures the time of trailing edges
; at CCI3A. TIM3 stores the time of the actual edge
;
TIMMOD3  CMP    &CCR3,&TAR     ; Occurred overflow of TAR?
         JHS    TM30           ; No, Timer Reg. > capt. value
         BIT    #CCIFG,CCTL0    ; Yes, TIMACYC0 yet updated?
         JNZ    TM30           ; No, value matches with CCR3
         SUB    #PERIOD,&CCR3   ; Yes, use CCR3 for correction
;
TM30     MOV    TIMACYC0,TIM3   ; Store sum of cycle counter
         ADD    &CCR3,TIM3     ; and captured event time
         RETI
;
         .sect  "TIMVEC",0FFF0h ; Timer_A Interrupt Vectors
         .word  TIM_HND               ; C/C Blocks 1..4 Vector
```

```
.word    TIMMOD0              ; Vector for C/C Block 0
.sect    "INITVEC",0FFFEh  ; Reset Vector
.word    INIT
```

The above example results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

CCR0 — repetition rate 16.969 kHz   19 cycles for the task, 11 cycles overhead   30 cycles
CCR1 — repetition rate 16.969 kHz   6 cycles for the update, 17 cycles overhead   23 cycles
CCR2 — repetition rate max. 2 kHz   60 cycles for the update, 32 cycles overhead   92 cycles
CCR3 — repetition rate max. 3 kHz   20 cycles for the update, 16 cycles overhead   36 cycles
CCR4 — repetition rate 16.969 kHz   6 cycles for the update, 0 cycles overhead   6 cycles

$$u_{CPU} = \frac{16.969 \times 10^6 \times (30+23+6)+2.0 \times 10^3 \times 92+3.0 \times 10^3 \times 36}{3.801 \times 10^6} = 0.34$$

The above result means a worst case CPU loading of approximate 34% due to the Timer_A activities (the tasks of the capture/compare blocks 2, 3 and 4 are not included).

6.3.9.10 Conclusion

This section demonstrated the possibilities of the Timer_A running in the up mode. Despite the dominance of the period register (CCR0) it is possible to capture signals, compare time intervals, and create timings in a real-time environment — all this in parallel with the pulse width modulation generated with the up mode.

6.3.10 Software Examples for the Up/Down Mode

This section shows several proven application examples for the Timer_A running in the up/down mode. Software definitions appear in the appendix. Whenever possible, the abbreviations defined in the *MSP430 Architecture Guide and Module Library* are used.

The software examples are independent of the MCLK frequency in use. Only the FLL multiplier constant, FLLMPY, and the repetition rate, f_{per}, need to be redefined if another combination is needed. The source lines for the definition of these important values are:

```
FLLMPY   .equ    122              ; 1. FLL multiplier

fper     .equ    19200            ; 2. PWM Repetition rate

TCLK     .equ    FLLMPY*32768/4   ; 3. FLLMPY x fcrystal/4

HLFPER   .equ    (TCLK/fper)/2    ; 4. Half Period of the PWM
```

Note:

The definitions assume an external crystal or an external frequency at the XIN input with a frequency of 32.768 kHz (2^{15} Hz).

1) **Definition of the CPU frequency f_{MCLK}.** The multiplier FLLMPY for the digitally controlled oscillator (DCO) is defined. The value for the actual frequency f_{MCLK} is (FLLMPY $\times 2^{15}$). The value 122 stands for $f_{MCLK} = 122 \times 2^{15} = 3.9977$ MHz.

2) **Definition of the desired repetition rate.** The value 19200 stands for a repetition rate of 19.2 kHz, which means 19200 complete up and down counts of the timer register TAR.

3) **Definition of the input frequency for the Timer Register (TAR).** The expression /4 indicates that the input divider is switched to the *Divide–by–Four* mode. The value shown stands for TCLK = 3.9977 MHz /4 = 999.424 kHz. Only the predivider used for the input divider (here /4) needs to be defined.

4) **Calculation of the TCLK cycles for the defined half period.** The full period consists of the half period counting up to the content of the period register CCR0 and the one counting down to 0 again. No change is necessary for this line.

6.3.10.1 Common Remarks

The up/down mode should be considered only for pulse width modulation (PWM) or DC generation. The advantage of this special PWM mode is the contributed switching of the output signals — unlike the up mode that switches on all output pulses at exactly the same time (when the timer register TAR is reset to 0), the up/down mode switches on and off the output pulses symmetrical to the 0 content of the timer register. See figure 6–44. If this feature is not needed, then the up mode with its simpler handling or the continuous mode with its five independent timings should be used.

Advantages of the Up/Down Mode:

❏ Distributed current switching (e.g. for digital motor control (DMC) applications)

❏ Free run without CPU loading for fixed PWM values (DAC, DMC)

❏ High PWM frequency possible due to pure hardware control

❏ Clever timings of the period register are usable for more than one real-time job

❏ For a given PWM repetition rate, an equally spaced second interrupt is available from the timer overflow interrupt, TIMOV. This doubles the available resolution for some applications

Disadvantages of the Up/Down Mode:

❏ Dominance of the period register — defines the time frame

❏ Direction change of the period register during the run needs special software handling. Interrupt-driven count direction indication is necessary for the software.

❏ Capturing has an inherent uncertainty for capturing values near the zero point (TAR = 0) and the middle of the period (TAR = CCR0).

❏ RAM extension for the timer register is necessary due to the normally short period.

❏ Change of the pulse width may cause an erroneous signal during one period.

6.3.10.1.1 Initialization

The initialization subroutine INITSR is used by all examples. This subroutine was explained and included in section *Software Examples of the Continuous Mode.* It includes the following tasks:

❏ Checks the reason for the initialization (switch on of the supply voltage, watchdog interrupt, or activation of the RESET input)

❏ Clears the RAM — or not — depending on the result of the check above

❏ Programs the system clock oscillator (multiplication factor N and optimum current switch FN_2, FN_3, or FN_4)

❏ Allows the digitally controlled oscillator to settle at the appropriate tap, providing the correct MCLK frequency

6.3.10.1.2 Timer Clock

For the timer clock, there is no difference between the up mode and the up/down mode. See section *Software Examples for the Up Mode* for details.

6.3.10.1.3 Timing Considerations

As with the up mode, the independence of the five timings provided by the continuous mode is not possible with the up/down mode. The period register (CCR0) dictates the timing frame for all other capture/compare blocks. With the up/down mode, things are a little bit more complex due to the count direction change of the timer register (TAR) when it reaches the content of the period register (CCR0).

Two additional RAM registers — as with the up mode — are used for the management of the compared or captured data:

❏ TIMACNT — Period counter. This register counts the number of half periods. Its bit 0 (LSB) functions as the count direction bit for the timer register TAR:

■ TIMACNT.0 = 0 — Timer register counts upward to nCCR0

■ TIMACNT.0 = 1 — Timer register counts downward to 0

❏ TIMACYCx — Cycle counter. Counts the TCLK cycles of the timing (one or more words)

See also figure 6–48. The contents of these two registers, including the count direction bit, are shown there for an example. Figure 6–52 gives an explanation of the update of these two registers.

Update of Extension Registers — Unlike with the continuous mode and the up mode, the update of these extension registers is made with the interrupt handlers of both the period register (CCR0) and the timer overflow interrupt (TIMOV). The reason is the count direction bit that needs to be updated each half period (up and down count direction). The main part is executed by the interrupt handler of the period register due to its higher interrupt priority and faster interrupt response. The method used for the update of the extension registers allows an automatic self synchronization:

```
BIS      #1,TIMACNT       ; CCR0: TIMACNT always odd

...

INC      TIMACNT          ; Timer Overflow: increment
```

Real Time Environment — See section *Software Examples for the Up Mode* for details. There is no difference between the up mode and the up/down mode.

Output Units — The shown PWM examples all use the toggle/reset mode (positive output pulses) or the toggle/set mode (negative output pulses) of the output units. The other output modes are not applicable for PWM generation in the up/down mode.

6.3.10.1.4 Interrupt Overhead

The calculations for the CPU loading that are appended to the software examples split the necessary cycles for an interrupt into two parts:

❑ **Overhead** — This part sums the cycles that are necessary for the CPU to execute the interrupt (saving of the program counter and the status register, decision as to which interrupt needs to be serviced, and restoring of the CPU registers).

❑ **Update or Task** — This actually does the work that needs to be done (incrementing of counters, changing of status bytes, reading of input information, etc.).

Like it is for the up mode, the number of overhead cycles is:

Interrupt of the period register CCR0	11 MCLK cycles
Interrupt of capture/compare registers x:	16 MCLK cycles
Interrupt of the timer register overflow:	14 MCLK cycles

6.3.10.2 Differences Between the Timer_A Versions

Two versions of the Timer_A hardware exist. They differ only in the perfor-
mance of the up/down mode:

❑ The version in the current MSP430C33x outputs a 50% PWM signal with
a doubled period if the capture/compare register contains 0. See Figure
6–44.

❑ The improved version running in the MSP430C11x, MSP430C33xA, and
all future family members outputs a fixed voltage (0% or 100% PWM) for
the capture/compare register content = 0. See Figure 6–45.
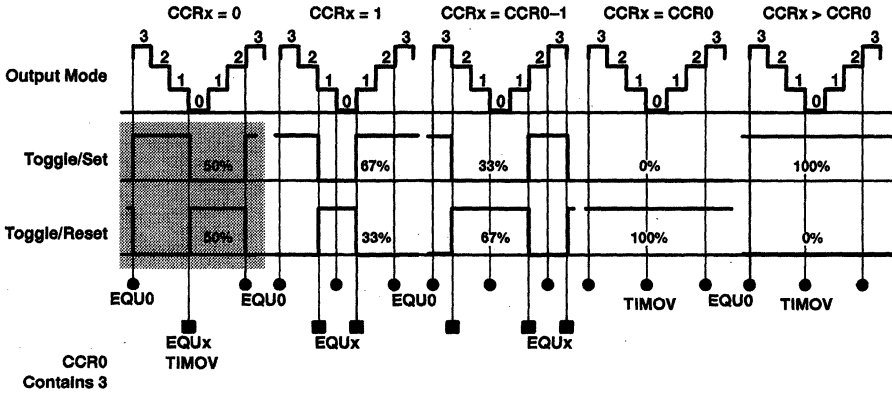
Figure 6–44. PWM Signals at Pin TAx for the Current MSP430C33x Version
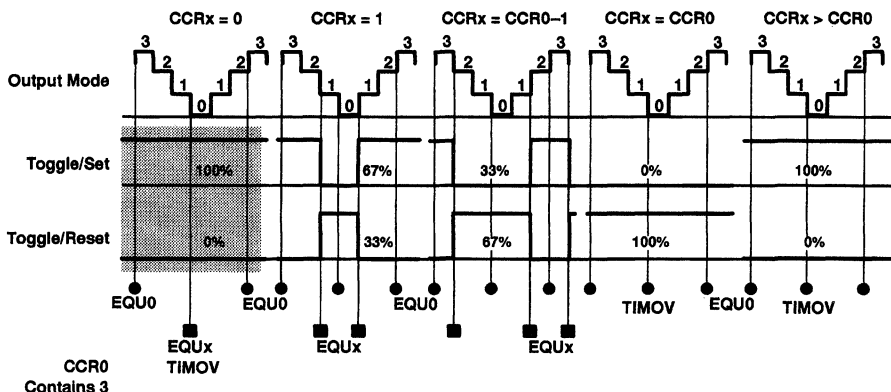
Figure 6–45. PWM Signals at Terminal TAx for the Improved MSP430C11x Version

The software examples are applicable to both versions — the distinction is made by a software flag named TAV0:

TAV0 = 0 — the Timer_A version of the current MSP430C33x is used

TAV0 = 1 — the improved Timer_A version for the MSP430C11x is used

Both versions output the correct 0 value for CCRx > CCR0. The longest half period that can be used is 0FFFEh, due to the value 0FFFFh necessary for 0.

6.3.10.2.1 .MACRO Definition for the PWM Range Check

Due to the behavior of the Timer_A running in the up/down mode, checks must be made to determine if the calculated PWM values are in the acceptable range or not.

Note:

These checks are not necessary if tables that contain valid data only are used, — 0FFFFh for the output value 0 and the content of the period register CCR0 as the maximum value (100%), for example.

To get a legible source, these checks are written as an assembler macro. This macro replaces the following two checks:

❑ If the calculated PWM value is greater than the half period contained in the period register CCR0

❑ If the calculated PWM value is 0

If one of these two possibilities is true, then a corrected value is used.

The macro is designed for two modes. They are distinguished by the software flag PERIOD_VAR:

❏ **Fixed period** — period register CCR0 always contains the same value. PERIOD_VAR = 0

❏ **Variable period** — CCR0 contains variable values. PERIOD_VAR = 1

The macro also distinguishes between the two Timer_A hardware versions (see Section 6.3.10.2 for details):

❏ The current MSP430C33x hardware: TAV0 = 0

❏ The improved MSP430C11x hardware: TAV0 = 1

Example 6–47. Macro Code

```
; The MACRO corrects input values addressed by arg1
; (0 to 0FFFEh) to valid input values.
; The four destination addressing modes are valid for arg1.
;
CHCK_PWM_RNG     .macro   arg1       ; arg1: address of PWM value
         .if     PERIOD_VAR=0        ; Fixed or variable period?
         CMP     #HLFPER+1,arg1      ; Fixed: result > HLFPER?
         JLO     L$1?                ; No, proceed
         MOV     #HLFPER,arg1        ; Yes, use HLFPER (100%)
         .else                       ; Variable period
         CMP     arg1,&CCR0          ; Result > Period Register?
         JHS     L$1?                ; No, proceed
         MOV     &CCR0,arg1          ; Yes, use HLFPER (100%)
         .endif
L$1?     .equ    $
         .if     TAV0=0              ; MSP430x33x or x1xx?
         TST     arg1                ; MSP430x33x:
         JNZ     L$2?                ; is arg1 = 0?
         MOV     #0FFFFh,arg1        ; Yes, use max. value
L$2?     .equ    $
         .endif
```

```
        .endm
The call of the above macro is
; Definitions for the .MACRO
;
PERIOD_VAR .equ   0               ; Fixed period
TAV0       .equ   0               ; MSP430x33x version
           . . .
        CHCK_PWM_RNG  R6          ; Check calc. PWM value in R6
        MOV      R6,TA1PWM        ; Corrected value to buffer
; or
        CHCK_PWM_RNG  HELP        ; Check PWM value in HELP
        MOV      HELP,TA1PWM      ; Update buffer
```

Note:

Software written for the MSP430C33x version of the Timer_A is upward compatible with the MSP430C11x version — it will also run well with the improved Timer_A hardware (only an unnecessary check for zero is made).

6.3.10.3 Update of the Capture/Compare Registers

As with the up mode, only a synchronous update will give undisturbed output pulses. The update with the accompanying interrupt handler is not possible for the up/down mode — the required toggling results in unpredictable output pulses for this kind of update. Four possibilities are shown here for the synchronous update by the Interrupt Handlers of capture/compare block 0 and the timer overflow:

1) Frequent common update of the capture/compare registers by the CCR0 handler

2) Frequent common update of the capture/compare registers by the TIMOV handler

3) Infrequent common update

4) Infrequent individual update

Unlike with the continuous mode and the up mode, only the interrupts of the period register (CCR0) and the timer overflow (TIMOV) are enabled for all of the four update modes.

The four possibilities are described in the following paragraphs. To find the appropriate solution for a given timing problem, the following decision path may be used:

❏ Is a very fast update of the capture/compare registers necessary? If yes, use solution 1 or 2, if no, continue.

❏ Are all of the new update values available at the same time? If yes, use solution 3, otherwise use solution 4.

6.3.10.3.1 Frequent Common Update by CCR0

The interrupt handler of capture/compare block 0 updates the capture/ compare registers CCRx with the repetition rate defined by the period register CCR0.

This update mode is used for the *Digital Motor Control with Symmetric Pulse Width Modulation.*

If the range for the PWM output values is limited from 1 cycle to (CCR0) cycles, then the following simple update sequence may be used:

```
; R6 contains new PWM info for CCR2. Range: 1 to (CCR0).
;
        MOV     R6,TA2PWM         ; Actualize PWM buffer
```

If the calculation results for the PWM output values can be 0% or >100%:

CCRx > CCR0 .or. CCRx = 0   for the current MSP430C330x

CCRx > CCR0   for the MSP430C110x,

then a special treatment is necessary due to the special behavior of the capture/compare logic under these circumstances. The capture/compare register x value then needs to be modified. To determine these special cases, the following update sequence may be used (the macro CHCK_PWM_RNG is explained in Section 6.3.10.2.1 *.MACRO Definition for the PWM Range Check*):

```
; R6 contains the calculated PWM info for CCR2.
; Range: 0 to HLFPER+x. Check if a modification is necessary
; Software is written for a constant Period Register CCR0
;
PERIOD_VAR .equ   0                  ; Constant period
TAV0    .equ   0                     ; MSP430x33x version
        ...
        CHCK_PWM_RNG  R6             ; Correct R6 if out of range
        MOV     R6,TA2PWM            ; Actualize TA2PWM buffer
        ...                          ; Continue
```

> If a variable period is used — the content of the period register CCR0 changes during the program flow — then the lines above change to:

```
; R6 contains the calculated PWM info for CCR2.
; Range: 0 to HLFPER+x. Check if a modification is necessary
; Software is written for a variable Period Register CCR0
;
PERIOD_VAR .equ   1                 ; Variable period
TAV0       .equ   0                 ; MSP430x33x version
           . . .
           CHCK_PWM_RNG  R6         ; Correct R6 if out of range
           MOV     R6,TA2PWM        ; Actualize TA2PWM buffer
           . . .                    ; Continue
```

> The part of the code that modifies the PWM values of the Timer_A looks like this:

```
; Handler of the Period Register CCR0
;
TIMMOD0 MOV    TA1PWM,&CCR1     ; Modify C/C Block 1 synchr.
        MOV    TA2PWM,&CCR2     ; Modify C/C Block 2 synchr.
        MOV    TA3PWM,&CCR3     ; Modify C/C Block 3 synchr.
        ADD    #2*HLFPER,TIMACYC0
        . . .                  ; Other tasks of the handler
        RETI
```

6.3.10.3.2 Frequent Common Update by the Timer Overflow TIMOV

> If the interrupt handler of the period register CCR0 has to perform many tasks, then it is advised to shift one half of these tasks to the interrupt handler of the timer overflow (TIMOV). This handler has the lowest interrupt priority, but with the up/down mode, this does not play a role because the interrupts of the capture/compare blocks 1 to 4 are normally disabled. The same background software is used as is shown with the update by the period register (CCR0) (the macro CHCK_PWM_RNG is explained in Section 6.3.10.2.1 *MACRO Definition for the PWM Range Check*).

```
;
PERIOD_VAR .equ   0                 ; Fixed period
TAV0       .equ   1                 ; MSP430x11x version
           . . .                    ; Calc. PWM value in R7
```

```
        CHCK_PWM_RNG  R7            ; Correct R7 if out of range
        MOV      R7,TA2PWM          ; Actualize TA2PWM buffer
        ...                         ; Continue
```

The part of the code that modifies the PWM values of the Timer_A looks like this:

```
TIM_HND  ADD      &TAIV,PC          ; Serve highest Timer_A request
         RETI                       ; No request
         RETI                       ; C/C Block 1: INTRPT disabled
         RETI                       ; C/C Block 2: INTRPT disabled
         RETI                       ; C/C Block 3: INTRPT disabled
         JMP      TIMMOD4           ; C/C Block 4: Capturing
;
; Handler of the Timer Overflow TIMOV
;
TIMOV    MOV      TA1PWM,&CCR1      ; Timer_A reached zero:
         MOV      TA2PWM,&CCR2      ; Modify C/C Blocks x
         MOV      TA3PWM,&CCR3
         INC      TIMACNT           ; Actualize half period counter
         RETI
```

6.3.10.3.3 Infrequent Common Update

The interrupt handlers of the capture/compare block 0 or the timer overflow update the capture/compare registers CCRx with a repetition rate given by the calculation speed of the background program. If new PWM values are calculated or read for all capture/compare blocks, then a common flag is set and the update is enabled in this way. This solution is used if the PWM values for the update are available at (nearly) the same time — by table processing, for example.

This update mode is used with the example *TRIAC Control.*

If the range for the calculated PWM output values is limited from 1 cycle to (CCR0) cycles, then the following simple update sequence may be used:

```
; R6 to R8 contain new PWM info for Output Units 1 to 3
; Range: 1 to (CCR0).
;
```

```
        MOV     R6,TA1PWM       ; Actualize CCR1 pulse length
        MOV     R7,TA2PWM       ; CCR2
        MOV     R8,TA3PWM       ; CCR3
        BIS.B   #1,FLAG         ; Set update flag
        ...                     ; Intrpt handler resets FLAG
```

If the output values 0% or >100% are actually used, then a special treatment is necessary. To correct these special cases, the following update sequence may be used (the macro CHCK_PWM_RNG is explained in Section 6.3.10.2.1 *.MACRO Definition for the PWM Range Check*):

```
; R6 to R8 contain new PWM info for Output Units 1 to 3.
; Range: 0 to (CCR0)+x.
; Check if a correction is necessary.
;
        CHK_PWM_RNG  R6        ; Check the PWM range
        MOV     R6,TA1PWM       ; Write corrected R6 to buffer
        CHK_PWM_RNG  R7        ; Check the PWM range
        MOV     R7,TA2PWM       ; Write corrected R7 to buffer
        CHK_PWM_RNG  R8        ; Check the PWM range
        MOV     R8,TA3PWM       ; Write corrected R8 to buffer
        BIS.B   #1,FLAG         ; Start common update
        ...                     ; Continue in background
```

The update part of the code in the interrupt handlers of the period register CCR0 or the timer overflow TIMOV looks like this:

```
        BIT.B   #1,FLAG         ; Is update flag set?
        JZ      L$1             ; No, continue
        MOV     TA1PWM,&CCR1    ; Actualize CCR1 pulse length
        MOV     TA2PWM,&CCR2    ; dito CCR2
        MOV     TA3PWM,&CCR3    ; dito CCR3
        BIC.B   #1,FLAG         ; Reset update flag
L$1     ...                     ; Continue INTRPT handler
```

If the other seven bits of the RAM byte FLAG are not used, then a faster version of the above update sequence may be used. The resetting of the bit is not necessary and saves 4 cycles.

```
        RRA.B   FLAG            ; Is update flag FLAG.0 set?
```

```
        JNC     L$1                 ; No, continue
        MOV     TA1PWM,&CCR1        ; Actualize CCR1 pulse length
        MOV     TA2PWM,&CCR2        ; dito CCR2
        MOV     TA3PWM,&CCR3        , dito CCR3
L$1     ...                         ; Continue INTRPT handler
```

6.3.10.3.4 Infrequent Individual Update

The interrupt handler of the period register or the Timer Overflow update the capture/compare register CCRx with a repetition rate given by the calculation speed of the background program. If a new PWM value is calculated for a capture/compare block, then an individual flag is set and the update for this capture/compare block is made. This method is used if the PWM values for the update are not available at the same time. This update mode is used with the example *Capturing with the Up/Down Mode*. It is the update mode with the lowest overhead. The macro CHCK_PWM_RNG is detailed in Section 6.3.10.2.1 *.MACRO Definition for the PWM Range Check*.

```
; R6 contains new PWM info for CCR1. Range: 0 to (CCR0)+x.
; Check if a modification is necessary:
; Software is written for a variable Period Register CCR0
;
TAV0        .equ    0               ; MSP430X33x version
PERIOD_VAR  .equ    1               ; Variable period
            ...
            CHK_PWM_RNG  R6          ; Check/correct result in R6
            MOV     R6,TA1PWM        ; Actualize TA1PWM buffer
            BIS     #2,FLAG          ; Start update of CCR1
            ...                      ; Start calculation for CCR2
;
; R6 contains new PWM info for CCR2. Range: 0 to (CCR0)+x
;
            CHK_PWM_RNG  R6
            MOV     R6,TA2PWM        ; Actualize TA2PWM buffer
            BIS     #4,FLAG          ; Start update of CCR2
            ...                      ; Continue
```

The interrupt handler of the period register CCR0 or the timer overflow (TI-MOV) decodes the necessary task as follows (4 to 20 MCLK cycles are needed):

```
        ...                         ; TIMOV or CCR0 handler
        ADD     FLAG,PC             ; Flag contains 0 to 6
        RETI                        ; 0: No update necessary
        JMP     P1                  ; 2: Update CCR1
        JMP     P2                  ; 4: Update CCR2
        MOV     TA1PWM,&CCR1        ; 6: Update CCR2 and CCR1
P2      MOV     TA2PWM,&CCR2        ; 4: Update only CCR2
        CLR     FLAG
        RETI
;
P1      MOV     TA1PWM,&CCR1        ; 2: Update only CCR1
        CLR     FLAG
        RETI
```

The above sequence may be changed easily for the update of three capture/compare registers (like is used for three phase DMC).

6.3.10.3.5 Overhead for the Update

These four update modes may be mixed if this is an advantage.

Table 6–24 shows the overhead calculation and the percentage of the update overhead for the four different update methods. The calculation results are based on:

f_{MCLK}	Frequency of the system clock generator (MCLK)	4 MHz
f_{update}	Update frequency for the capture/compare registers	1 kHz
f_{per}	Timer_A repetition rate defined by the period register CCR0	12 kHz
n	Number of C/C blocks used for the PWM generation	3

Table 6–24. Interrupt Overhead for the Four Different Update Methods

UPDATE METHOD	OVERHEAD FORMULA (CPU CYCLES)	OVERHEAD PERCENTAGE
Frequent Update with CCR0	$n \times f_{per} \times 6$	5.4%
Frequent Update with TIMOV	$n \times f_{per} \times 6$	5.4%
Infrequent Common Update	$(f_{per} \times 6) + (f_{update} \times (n \times 6 + 4))$	2.3%
Infrequent Individual Update	$(f_{per} - f_{update}) \times 3 + (f_{update} \times 15)$	1.2%

> **Note:**
>
> No interrupt is generated – and therefore no interrupt overhead – for capture/compare registers containing a value greater than the content of the period register CCR0 (output TAx = 0 for Toggle/Reset resp. TAx = 1 for Toggle/Set).

6.3.10.4 Digital Motor Control With Symmetric Pulse Width Modulation

The medium output voltage V_{PWM} at the TAx terminal with respect to the necessary register content (n_{CCRx}) for a given voltage V_{PWM} is:

$$V_{PWM} = Vcc \times \frac{n_{CCRx}}{n_{CCR0}} = Vcc \times \frac{tpw}{tper} \rightarrow n_{CCRx} = \frac{V_{PWM}}{Vcc} \times n_{CCR0}$$

Where:

V_{PWM}	Medium output voltage at the TAx terminal	[V]
V_{CC}	Supply voltage of the system	[V]
n_{CCR0}	Content of the period register CCR0	
n_{CCRx}	Content of the capture/compare register CCRx	
t_{pw}	Time generated by the capture/compare register	[s]
t_{per}	Period generated by the period register CCR0	[s]

Table 6–25 shows the necessary content of a capture/compare register CCRx to get some defined values for an unsigned output voltage V_{PWM}:

Table 6–25. Output Voltages for Unsigned PWM

OUTPUT VOLTAGE (VPWM)	CONTENT OF CCRx n_{CCRx}
0 V	0
$0.25 \times V_{CC}$	$n_{CCR0} \times 0.25$
$0.5 \times V_{CC}$	$n_{CCR0} \times 0.5$
$0.75 \times V_{CC}$	$n_{CCR0} \times 0.75$
V_{CC}	n_{CCR0}

If the output voltage is seen as a signed voltage — like for 3-phase digital motor control — then the voltage $0.5 \times V_{CC}$ is seen as the 0 point. The signed output voltage V_{PWM} gets:

$$V_{PWM} = Vcc \times \left(\frac{n_{CCRx}}{n_{CCR0}} - 0.5 \right) \rightarrow n_{CCRx} = \left(\frac{V_{PWM}}{Vcc} + 0.5 \right) \times n_{CCR0}$$

To calculate the value for nCCRx for the sine of a given angle, α, the formula is (full V_{CC} range):

$$n_{CCRx} = \frac{1 + sin\alpha}{2} \times n_{CCR0}$$

Table 6–26 shows the necessary contents of a capture/compare register CCRx to get some defined values for a signed output voltage V_{PWM}.

Table 6–26. Output Voltages for Signed PWM

OUTPUT VOLTAGE (VPWM)	CONTENT OF CCRx n_{CCRx}	COMMENT
$-0.5 \times V_{CC}$	0	Most negative output voltage
$-0.25 \times V_{CC}$	$n_{CCR0} \times 0.25$	Half negative output voltage
0 V	$n_{CCR0} \times 0.5$	0 voltage
$0.25 \times V_{CC}$	$n_{CCR0} \times 0.75$	Half positive output voltage
$0.5 \times V_{CC}$	n_{CCR0}	Most positive output voltage

Figure 6–46 shows some of the output voltages listed above for a three-phase system.
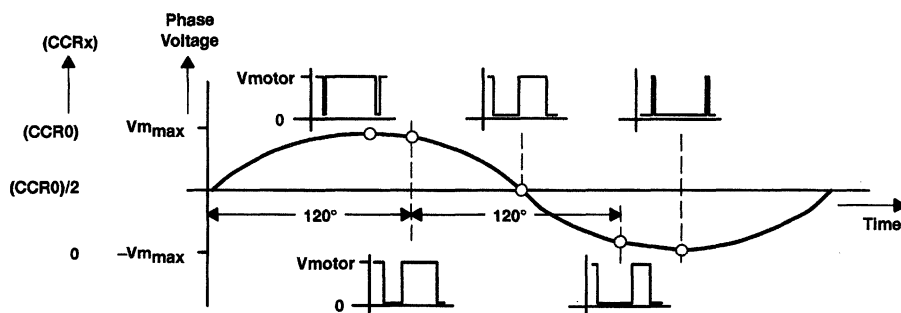
Figure 6–46. PWM Outputs for Different Phase Voltages

Note that 0 volt for a motor phase is generated by a pulse width of one half of the length of the period.

Example 6–48. PWM Outputs for Different Phase Voltages

The software example shows the generation of PWM output signals for a three-phase electric motor. The MSP430 delivers the PWM output signals and controls the speed of the motor by the input signal CCI4A coming from the tach/generator.

The capture/compare blocks 1 to 3 are used for the generation of the PWM signals for the three phases.

The capture/compare block 4 is used for the capturing of the speed signal coming from the shaft of the motor. Up to 6000 rpm (100 rev/sec) are used with this example, with four output pulses per revolution. The positive edge of the input signal is captured and requests interrupt.

All security functions are included in the external control chip IR2130 (over current, delays for the transistors, etc.).
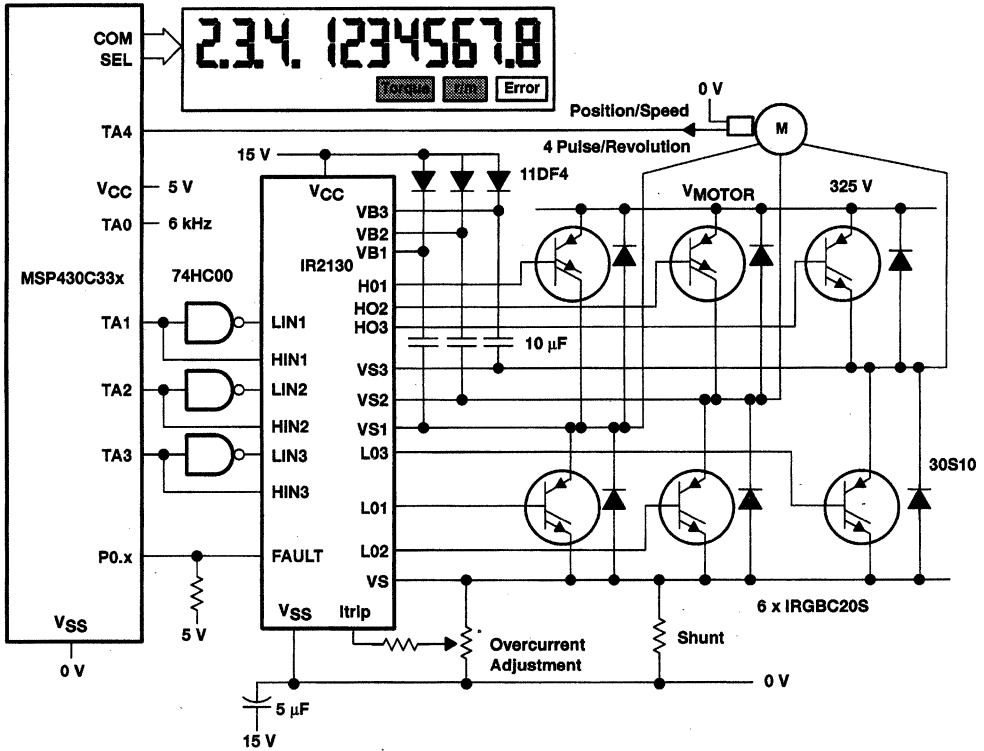
Figure 6–47. PWM Motors Control for High Motor Voltages

The system clock frequency is 4 MHz (exactly $f_{MCLK} = 122 \times 32768 = 3.9977$ MHz). The pulse repetition frequency is 12 kHz.

The output unit 0 outputs 6 kHz without any overhead. This signal may be used for peripherals or for synchronization. The signal is always present, even if the signals at the TAx outputs disappear due to an output signal with 0% or 100% pulse width.

The example uses the frequent common update of the compare/compare register. See Section 6.3.10.3.1 *Frequent Common Update by CCR0* for details.

Figure 6–48 shows the output signals at the times that they have phase shifts of 0°, +120° and –120°.
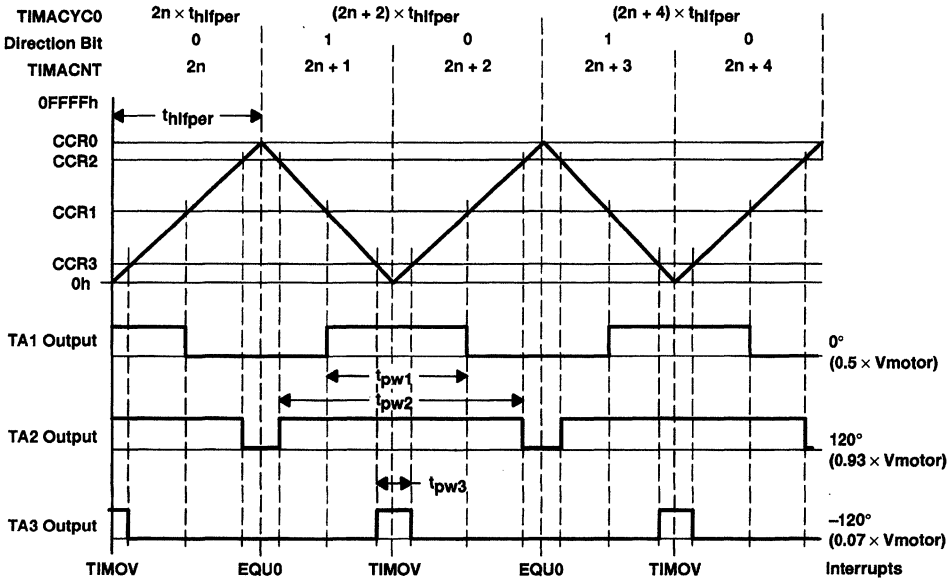
Figure 6–48. Symmetric PWM Timings Generated With the Up/Down Mode

Example 6–49. Symmetric PWM Timings Generated With the Up/Down Mode

```
; Software example:
; TA0: symmetric output signal  6.0kHz
; TA1: positive PWM signal     12.0kHz. Length in TA1PWM
; TA2: positive PWM signal     12.0kHz. Length in TA2PWM
; TA3: positive PWM signal     12.0kHz. Length in TA3PWM
;
; Hardware definitions
;
FLLMPY    .equ    122             ; FLL multiplier for 3.9977MHz
fper      .equ    12000           ; 12.0kHz repetition rate
TCLK      .equ    FLLMPY*32768    ; TCLK: FLLMPY x fcrystal
HLFPER    .equ    (TCLK/fper)/2   ; Period of output signals
```

```
TAV0        .equ     0                 ; MSP430C33x Timer_A
PERIOD_VAR .equ     0                 ; Invariable period in CCR0
STACK       .equ     600h              ; Stack initialization address
;
; RAM definitions
;
TA1PWM     .equ     202h              ; Pulse length Block 1 (0..167)
TA2PWM     .equ     204h              ; Pulse length Block 2 (0..167)
TA3PWM     .equ     206h              ; Pulse length Block 3 (0..167)
CPT4        .equ     208h              ; Captured motor shaft events
TIMACYC0 .equ      20Ah              ; Low cycle counter (15..0)
TIMACYC1 .equ      20Ch              ; High cycle counter (31..16)
TIMACNT   .equ     20Eh              ; Period Counter, Bit 0 = Dir
;
          .text                       ; Software start address
;
INIT      MOV       #STACK,SP         ; Initialize Stack Pointer
          CALL      #INITSR           ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Up/Down Mode, INTRPTs on for
; TIMOV, Period Register and C/C Block 4 (Capture Mode)
;
          MOV       #ISMCLK+CLR+TAIE,&TACTL ; Define Timer_A
          MOV       #HLFPER,&CCR0     ; Period Register
          MOV       #HLFPER/2,R5      ; Value for 0V to R5
          MOV       R5,&CCR1          ; TA1: pulse width = 0V
          MOV       R5,&CCR2          ; TA2: as before
          MOV       R5,&CCR3          ; TA3: as before
          MOV       #OMT+CCIE,&CCTL0  ; TA0: Toggle Mode
          MOV       #OMTR,&CCTL1      ; TA1: Toggle/Reset Mode
          MOV       #OMTR,&CCTL2      ; TA2: Toggle/Reset Mode
          MOV       #OMTR,&CCTL3      ; TA3: Toggle/Reset Mode
          MOV       #CMPE+ISCCIA+SCS+CAP+CCIE,&CCTL4 ; +edge shaft
          MOV.B     #TA4+TA3+TA2+TA1+TA0,&P3SEL ; Define I/Os
;
          MOV       R5,TA1PWM         ; Start value Block 1: 0V
```

```
        MOV     R5,TA2PWM         ; Start value Block 2: 0V
        MOV     R5,TA3PWM         ; Start value Block 3: 0V
        CLR     TIMACYC0          ; Clear low cycle counter
        CLR     TIMACYC1          ; Clear high cycle counter
        CLR     TIMACNT           ; Clear period counter
        MOV.B   #CBMCLK+CBE,&CBCTL ; Output MCLK at XBUF pin
        BIS     #MUPD,&TACTL      ; Start in Up/Down Mode
        EINT                      ; Enable interrupts
MAINLOOP ...                      ; Continue in background
;
; Calculations resulted in new PWM values. The new results
; are stored in R6 (C/C Block 1), R7 (C/C Block 2) and R8
; (C/C Block 3). Check if ranges are valid:
;
        CHCK_PWM_RNG  R6          ; Correct R6 range
        MOV     R6,TA1PWM
;
        CHCK_PWM_RNG  R7          ; Correct R7 range
        MOV     R7,TA2PWM
;
        CHCK_PWM_RNG  R8          ; Correct R8 range
        MOV     R8,TA3PWM
        ...                       ; Continue in background
;
; Read the last captured value of the tacho generator
;
        MOV     CPT4,R6           ; For calculations
        ...                       ; Control algorithm for speed
;
; Interrupt handler for CCR0: the Period Register. The cycle
; counters and the half period counter are updated.
; A symmetric 6.0kHz signal is output by the Output Unit 0
; TIMACYC0 points to next the 0-crossing of the TAR
;
TIMMOD0 MOV     TA1PWM,&CCR1      ; Update PWM registers
        MOV     TA2PWM,&CCR2
```

```
          MOV       TA3PWM,&CCR3
          ADD       #2*HLFPER,TIMACYC0 ; Add fixed period to
          ADC       TIMACYC1           ; cycle counters
          BIS       #1,TIMACNT         ; Half period counter +1 (Down)
          RETI
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4
; and Timer Overflow.
; Only the timer overflow interrupt and the C/C Block 4 are
; used. The other interrupts are disabled. The PWM generation
; is made by the timer hardware and updated by the CCR0 intrpt
;
TIM_HND   ADD       &TAIV,PC           ; Add Jump table offset
          RETI                         ; No interrupt pending
          RETI                         ; C/C Block 1: Intrpt disabled
          RETI                         ; C/C Block 2: Intrpt disabled
          RETI                         ; C/C Block 3: Intrpt disabled
          JMP       TIMMOD4            ; C/C Block 4: Capturing used
;
; Timer overflow: the half period counter is incremented
;
TIMOV     INC       TIMACNT            ; Make TIMACNT even (DIR = UP)
          RETI                         ; Back to main program
;
; C/C Block 4 captures the revolutions of the motor. Dependent
; on the count direction of TAR, CCR4 is added or subtracted.
; The positive edge of the input signal at TA4 is captured
; and requests interrupt. Time out cannot occur due to
; low input frequency.
;
TIMMOD4   MOV       TIMACYC0,CPT4      ; Cycle counter for calculation
          BIT       #1,TIMACNT         ; Direction UP?
          JNZ       T40                ; No, DOWN (1)
;                                        Direction is UP
          ADD       &CCR4,CPT4         ; Build time of captured event
          RETI                         ; Back to main program
```

```
;                                ; Direction is DOWN
T40       SUB       &CCR4,CPT4   ; Build time of captured event
          RETI
;
          .sect     "TIMVEC",0FFF0h  ; Timer_A Interrupt Vectors
          .word     TIM_HND          ; C/C Blocks 1 to 4
          .word     TIMMOD0          ; Capture/Compare Block 0
          .sect     "INITVEC",0FFFEh ; Reset Vector
          .word     INIT
```

The example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

$$u_{CPU} = \frac{1}{f_{MCLK}} \; \Sigma \left(n_{intrpt} \times f_{rep} \right)$$

Where:

f_{MCLK}	Frequency of the system clock (DCO)	[Hz]
n_{intrpt}	Number of cycles executed by the interrupt handler	
f_{rep}	Repetition rate of the interrupt handler	[Hz]

Note:

The formula and the definitions given above are also valid for all subsequent software examples. Therefore they are not repeated.

CCR0 — repetition rate 12 kHz	32 cycles for the task, 11 cycles overhead	43 cycles
CCR4 — repetition rate 0.4 kHz	18 cycles for the task, 16 cycles overhead	34 cycles
TIMOV — repetition rate 12 kHz	4 cycles for the update, 14 cycles overhead	18 cycles

$$u_{CPU} = \frac{12000 \times \left(43 + 18\right) + 400 \times 34}{3.9977 \times 10^6} = 0.186$$

The result means a CPU loading of 19% due to the Timer_A for the digital motor control task.

6.3.10.5 TRIAC Control

TRIAC control for electric motors (DMC) or other loads is possible using the up/down mode as shown with the up mode of the Timer_A. But due to the sec-

ond interrupt coming from the timer overflow (TIMOV), the doubled resolution is possible as with the up mode. The control software now counts the number of half periods and fires the TRIAC after the reaching of the calculated value.

The medium resolution p_{med} is:

$$pmed = \frac{1}{2 \times f_{MAINS} \times t_{hlfper}}$$

Where:

f_{MAINS}	AC line frequency	[Hz]
t_{hlfper}	Half period of the Timer_A, defined by CCR0	[s]

All considerations and formulas shown for the up mode are also valid for the up/down mode, except the doubled resolution for the same PWM period. Again, no capture/compare register is needed for the TRIAC control because only the period register with its interrupt and output unit 0 is used. This frees the remaining capture/compare blocks for other tasks.

Figure 6–49 shows the hardware for the TRIAC control of this example. The TRIAC hardware is exactly the same hardware as used with the up mode. In addition, a second three-phase motor is controlled by the same MSP430.
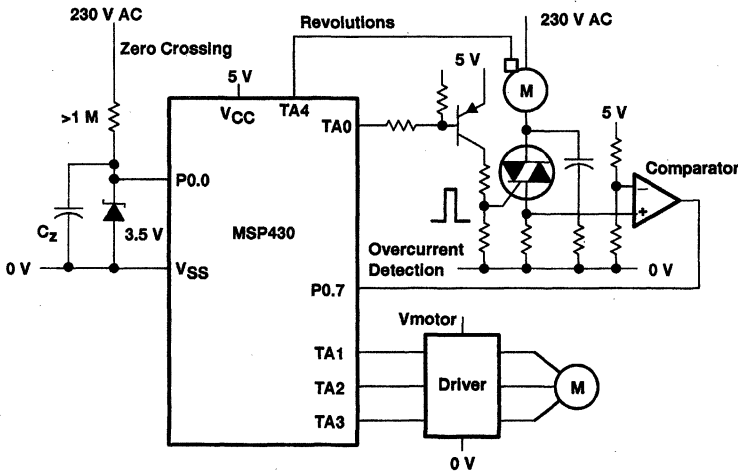
Figure 6–49. TRIAC Control and 3–Phase Control With the Timer_A

Figure 6–50 illustrates the software example given below. The timer register (TAR) is not shown to scale — 320 steps make one half wave of the 50-Hz line.
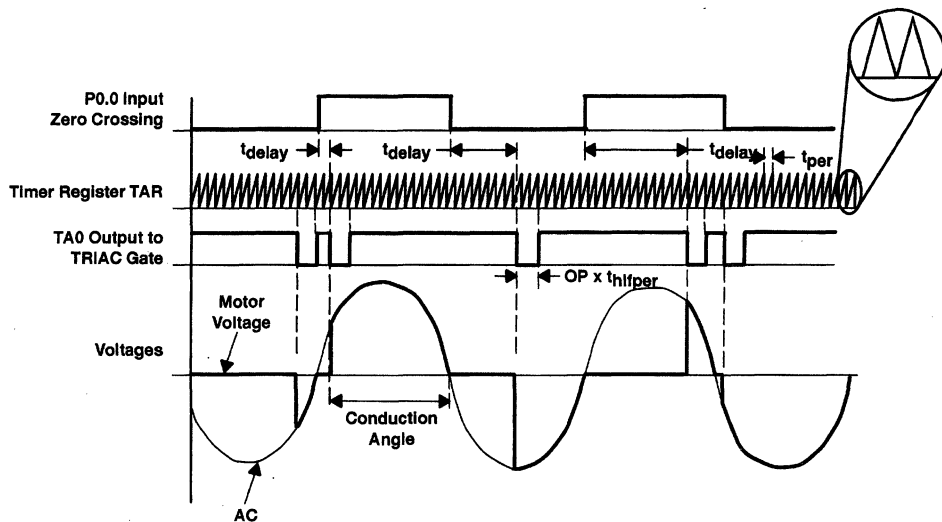
Figure 6–50. Signals for the TRIAC Gate Control With Up/Down Mode

Example 6–50. Static TRIAC Control Software

A static TRIAC control software example is shown. The calculated number of half periods until the TRIAC gate is fired after the zero crossing of the AC line voltage, is contained in the RAM word FIRANGL.

The medium resolution p_{med} is 320 steps per line half wave (2×16 kHz/100 Hz = 320). The minimum resolution, p_{min}, is 204 steps ($320 \times 2/\pi = 204$) which means approximately. 0.5% resolution. See the equations above.

At the TA1, TA2, and TA3 terminals negative PWM signals for digital motor control are output. The half period is defined by the period register (CCR0), the actual pulse length (TCLK cycles) is contained in the RAM words TA1PWM, TA2PWM, and TA3PWM. The common update is made with ≈1 kHz.

The speed of the TRIAC-controlled motor is measured with the input signal at input TA4 (CCI4A). The negative edges are captured and an interrupt is requested afterward.

The example uses the *infrequent common update* executed by the timer over-flow handler. See Section 6.3.10.3 *Update of the Capture/Compare* Registers-for details.

```
; Definitions for the TRIAC control software
;
FLLMPY      .equ     122                  ; FLL multiplier for 4.0MHz
fper        .equ     16000                ; 16.000kHz repetition rate
TCLK        .equ     FLLMPY*32768         ; TCLK (Timer Clock) [Hz]
HLFPER      .equ     (TCLK/fper)/2        ; Half period in Timer clocks
OP          .equ     4                    ; TRIAC gate pulse length
TAV0        .equ     0                    ; MSP430C33x Timer_A
PERIOD_VAR .equ      0                    ; Fixed half period in CCR0
;
; RAM definitions
;
TIMACYC0 .equ        202h                 ; Timer Register Extensions:
TIMACYC1 .equ        204h                 ; Cycle counters
TIMACNT  .equ        206h                 ; Counter for half periods
FIRANGL  .equ        208h                 ; Half wave - conduction angle
FIRTIM   .equ        20Ah                 ; Fire time rel. to TIMACNT
TA1PWM   .equ        20Ch                 ; PWM cycle count C/C Block 1
TA2PWM   .equ        20Eh                 ;                 C/C Block 2
TA3PWM   .equ        210h                 ;                 C/C Block 3
STTRIAC  .equ        212h                 ; Control byte (0 = off) Status
FLAG     .equ        213h                 ; 1: update for PWM request
CPT4     .equ        214h                 ; Captured shaft value
STACK    .equ        600h                 ; Stack initialization address
         .text                           ; Start of ROM code
;
; Initialize the Timer_A: MCLK, Up/Down Mode. Enable INTRPT
; for C/C Blocks 0 and 4 and Timer Overflow TIMOV.
; Prepare Timer_A Output Units
;
INIT     MOV         #STACK,SP            ; Initialize Stack Pointer SP
         CALL        #INITSR              ; Init. FLL and RAM
```

```
          MOV       #ISMCLK+CLR+TAIE,&TACTL ; Init. Timer
          MOV       #HLFPER,&CCR0      ; Half period to CCR0
          MOV       #OMOO+CCIE+OUT,&CCTL0 ; Set TA0 high, Output
          MOV       #OMTS,&CCTL1       ; TA1: neg. PWM pulses
          MOV       #OMTS,&CCTL2       ; TA2: neg. PWM pulses
          MOV       #OMTS,&CCTL3       ; TA3: neg. PWM pulses
          MOV       #CMNE+ISCCIA+SCS+CAP+CCIE,&CCTL4 ; -edge shaft
          BIS.B     #TA4+TA3+TA2+TA1+TA0,&P3SEL ; Define I/Os
          BIS.B     #P0IE0,&IE1        ; Enable P0.0 interrupt mains
          MOV.B     #CBMCLK+CBE,&CBCTL ; MCLK at XBUF pin
;

          CLR       TIMACYC0           ; Clear low cycle counter
          CLR       TIMACYC1           ; Clear high cycle counter
          CLR       TIMACNT            ; Clear half period counter
          CLR.B     STTRIAC            ; TRIAC off status (0)
          MOV       #HLFPER/2,TA1PWM   ; TA1: 0V
          MOV       #HLFPER/2,TA2PWM   ; TA2: 0V
          MOV       #HLFPER/2,TA3PWM   ; TA3: 0V
          MOV.B     #1,FLAG            ; Update PWM registers CCRx
          BIS       #MUPD,&TACTL       ; Start Timer_A (Up/Down)
          EINT                         ; Enable interrupts
MAINLOOP  ...                          ; Continue in mainloop
;
; Some TRIAC control examples:
; Start electric motor: checked result (half periods) in R5
; The result is the time difference from the zero crossing
; of the mains voltage (P0.0) to the first gate pulse
; (measured in Timer_A half periods)
;
          MOV       R5,FIRANGL         ; Delay (half per.) to FIRANGL
          MOV.B     #2,STTRIAC         ; Activate TRIAC control
          ...                          ; Continue in background
;
; The motor is running. A new calculation result is available
; in R5. It will be used with the next mains half wave
;
```

```
          MOV     R5,FIRANGL        ; Delay (half per.) to FIRANGL
          ...                       ; Continue in background
;
; Stop motor: switch off TRIAC control, TRIAC gate off
;
          CLR.B   STTRIAC           ; Disable TRIAC control
          BIS     #OUT,CCTL0        ; TA0 high, Output only Mode
          ...                       ; Continue with background
;
; Read the captured value of the tacho generator
;
          MOV     CPT4,R6
          ...                       ; Control algorithm for speed
;
; Preparation for the new PWM values start. A table with
; valid values only is used: no check is necessary
;
          MOV     ANGLE,R6          ; Current phase angle
          MOV     TABLE(R6),TA1PWM  ; Ph1: add   0 degrees
          MOV     TABLE+120(R6),TA2PWM ; Ph2: add 120 degrees
          MOV     TABLE+240(R6),TA3PWM ; Ph3: add 240 degrees
          BIS.B   #1,FLAG           ; Initiate common update
          ...                       ; Continue in background
TABLE     .word   HLFPER/2,100,... ; sin 0 to sin 600
;
; Interrupt handler for CCR0: the Period Register.
; - The cycle counters and the half period counter are updated
; - The TRIAC control task is executed
;
TIMMOD0   ADD     #2*HLFPER,TIMACYC0 ; Add (fixed) period to
          ADC     TIMACYC0          ; Cycle counters
          BIS     #1,TIMACNT        ; Half period counter +1 (Down)
;
; Interrupt handler for the TRIAC control. Entry point also
; from the Timer Overflow handler
;
```

```
TRIACC   EINT                       ; Allow nested interrupts
         PUSH     R5                 ; Save help register R5
         MOV.B    STTRIAC,R5         ; Status of TRIAC control
         MOV      STTAB(R5),PC       ; Branch to status handler
STTAB    .word    STATE0             ; Status 0: No TRIAC activity
         .word    STATE0             ; Status 2: activation possible
         .word    STATE4             ; Status 4: wait for gate pulse
         .word    STATE6             ; Status 6: wait for gate off
;
; TRIAC status 4: TRIAC gate is switched on for "OP" half
; periods after the value in FIRTIM is reached
;
STATE4   CMP      FIRTIM,TIMACNT     ; TRIAC gate time reached?
         JNE      STATE0             ; No
         BIC      #OUT,&CCTL0        ; Yes, TRIAC gate on
         ADD.B    #2,STTRIAC         ; Next TRIAC status (6)
;
; TRIAC status 0: No activity. TRIAC is off always
;
STATE0   POP      R5                 ; Restore help register
         RETI                        ; Return from interrupt
;
; TRIAC status 6: gate pulse is active. Check if it's time
; to switch off the TRIAC gate.
;
STATE6   MOV      FIRTIM,R5          ; Time TRIAC firing
         ADD      #OP,R5             ; Gate-on time (half periods)
         CMP      R5,TIMACNT         ; On-time terminated?
         JLO      STATE0             ; No
         BIS      #OUT,&CCTL0        ; Yes, TRIAC gate off
         MOV.B    #2,STTRIAC         ; TRIAC status 2:
         JMP      STATE0             ; Wait for next zero crossing
;
; Interrupt handler for C/C Blocks 1 to 4 and Timer Overflow
;
TIM_HND  ADD      &TAIV,PC           ; Serve highest priority requ.
```

```
        RETI                        ; No interrupt pending
        RETI                        ; C/C Block 1: INTRPT off
        RETI                        ; C/C Block 2: INTRPT off
        RETI                        ; C/C Block 3: INTRPT off
        JMP     TIMMOD4             ; C/C Block 4: Speed measurement
;
; The Timer Overflow interrupt handler:
; - Updates the PWM registers if necessary: FLAG.0 = 1
; - The TRIAC control task is executed
;
TIMOV   INC     TIMACNT             ; Incr. period counter (UP)
        BIT.B   #1,FLAG             ; Update necessary?
        JZ      TRIACC              ; No, to TRIAC control task
        MOV     TA1PWM,&CCR1        ; Yes update C/C Blocks
        MOV     TA2PWM,&CCR2
        MOV     TA3PWM,&CCR3
        BIC.B   #1,FLAG             ; Clear update flag
        JMP     TRIACC              ; To TRIAC control task
;
; C/C Block 4 captures the revolutions of the motor. Dependent
; on the count direction of TAR, the captured TAR value in
; CCR4 is added or subtracted. CPT4 contains the 16-bit value
; of the captured negative edge of the signal at TA4.
;
TIMMOD4 MOV     TIMACYC0,CPT4       ; Save cycle counter
        BIT     #1,TIMACNT          ; Direction UP?
        JNZ     T40                 ; No, DOWN (1)
;                                   ;  Direction is UP:
        ADD     &CCR4,CPT4          ; Build time of captured event
        RETI                        ; Back to main program
;                                   ; Direction is DOWN:
T40     SUB     &CCR4,CPT4          ; Build time of captured event
        RETI
;
; P0.0 Handler: the mains voltage causes interrupt with each
; zero crossing. The TRIAC gate is switched off first, to
```

```
; avoid the ignition for the actual half wave.
; Hardware debounce is necessary for the mains signal!
;
P00_HNDLR BIS    #OUT,&CCTL0    ; Switch off TRIAC gate
          EINT                  ; Allow nested interrupts
          XOR.B  #1,&P0IES      ; Change intrpt edge of P0.0
;
; If STTRIAC is not 0 ( 0 = inactivity) then the next TRIAC
; gate firing is prepared: STTRIAC is set to 4
;
          TST.B  STTRIAC        ; TRIAC control active?
          JZ     P00            ; STTRIAC = 0: no activity
          MOV.B  #4,STTRIAC     ; Yes, STTRIAC > 0
;
; The TRIAC firing time is calculated: TIMACNT + FIRANGL
; (current time + angle) in half periods
;
          MOV    TIMACNT,FIRTIM ; Half period counter
          ADD    FIRANGL,FIRTIM ; TIMACNT + delay -> FIRTIM
P00       RETI
;
          .sect  "TIMVEC",0FFF0h ; Timer_A Interrupt Vectors
          .word  TIM_HND        ; Vector for C/C Blocks 1..4
          .word  TIMMOD0        ; Vector for C/C Block 0
          .sect  "P00VEC",0FFFAh ; P0.0 Vector
          .word  P00_HNDLR
          .sect  "INITVEC",0FFFEh ; Reset Vector
          .word  INIT
```

The TRIAC control example results in a nominal CPU loading u_{CPU} (ranging from 0 to 1) for the active TRIAC control (STTRIAC = 4):

CCR0 — repetition rate 16 kHz	35cycles for the task, 11 cycles overhead	46 cycles
TIMOV — repetition rate 16 kHz	34 cycles for the update, 14 cycles overhead	48 cycles
CCR4 — repetition rate 0.4 kHz	18 cycles for the update, 16 cycles overhead	34 cycles
P0.0 — repetition rate 100 Hz	17 cycles for the task, 11 cycles overhead	28 cycles
Update — 1 kHz (TIMOV)	22 cycles for the task, 0 cycles overhead	22 cycles

$$u_{CPU} = \frac{16.0 \times 10^3 \times (46 + 48) + 0.4 \times 10^3 \times 34 + 100 \times 28 + 1.0 \times 10^3 \times 22}{4.0 \times 10^6} = 0.386$$

This results in a CPU loading of approximate 39% due to the static TRIAC control. The necessary tasks for the update of the half period counter and the cycle counters are included. The PWM activities alone load the CPU with less than 1% this way (f_{update} = 1 kHz).

6.3.10.6 RF Timing Generation

The repetition rate of the up/down mode in use must be a multiple of the data change frequency. The timing is now made by the interrupts of the period register and of the timer overflow. This allows the output of biphase code modulation and biphase space modulation with a 19.2 kHz repetition rate. The three different modulation methods and their conversion subroutines were discussed in detail in the section *Software Examples for the Continuous Mode*. The software shown in this section may be used with the following, simple modifications:

1) The repetition frequency is also chosen to 19.2 kHz, but the equally spaced TIMOV interrupt allows a 38.4 kHz time frame.

2) The output handler for the 128-bit buffer is executed by the interrupt handlers of the period register and the timer overflow (TIMOV) to get the doubled bit rate (as shown for the TRIAC control example in Section 6.3.10.5).

3) The output unit 0 uses the output only mode (exactly as shown for the TRIAC control example in Section 6.3.10.5). The interrupt handler of the period register CCR0 sets and resets the output TA0 by software.

Figure 6–51 shows the biphase code modulation for an input byte containing the value 96h. The other two modulation modes work the same way. The timing of the interrupts is shown below:
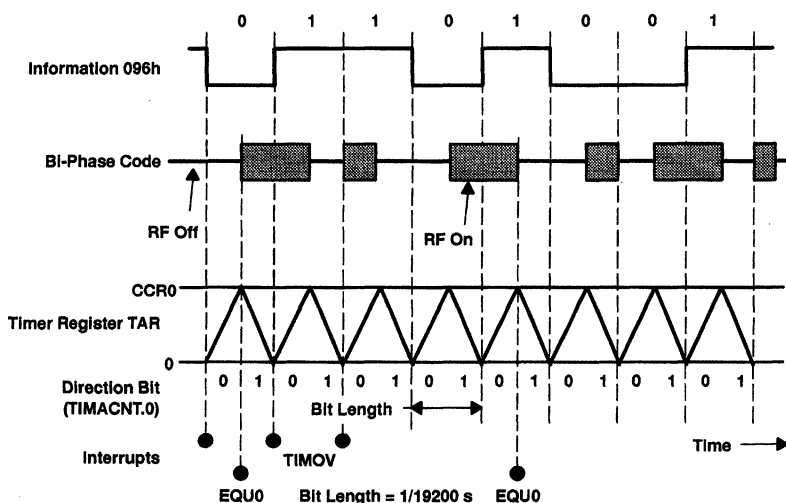
Figure 6–51. Biphase Code Modulation With the Up/Down Mode

6.3.10.7 Comparison With the Up/Down Mode

Comparison with the up/down mode is nearly impossible due to the uncertainty of the direction of the actual count. If comparison — which means precise interrupts or switching of the corresponding output unit — is important, then the up mode or the continuous mode should be used. With the up/down mode — and its normally high repetition rates — only interrupt-driven software switching is possible. The *TRIAC Control* example shows a method to use the interrupts of the period register (CCR0) and the timer overflow (TIMOV) for the control of outputs.

6.3.10.8 Capturing with the Up/Down Mode

Capturing of events is not as easy as with the continuous mode or the up mode. The reason is the changing count direction of the timer register (TAR) in the middle of the timer period. Due to the interrupt latency time, t_{IL}, an uncertainty zone exists at the two points where the timer register changes its direction. This uncertainty zone has the length $2 \times t_{IL}$. The interrupt latency time, t_{IL}, depends on the actual software — it ranges from 6 MCLK cycles to the longest program sequence with disabled interrupt. See also figure 6–52.

To get the time of an event with least calculation effort, the method shown in figure 6–52 is used:

❏ The interrupt handler of the period register CCR0 adds the length of a period to the cycle counters TIMACYCx. This is done in such a way, that these counters point forward to the next time point in which the timer register (TAR) reaches 0 again (TIMOV interrupt).

❏ The interrupt handler of the period register also sets the bit 0 (LSB) of the half period counter TIMACNT. This bit is used as the direction bit and indicates with this *1* the downward count direction.

❏ The interrupt handler of the timer overflow (TIMOV) increments the bit 0 (LSB) of the half period counter TIMACNT and sets it to *0* (upward count direction).

The setting (CCR0) and incrementing (TIMOV) of the direction bit (TIMACNT.0) results in an incrementing that is self synchronizing.

To calculate the time of an event at terminal TAx, it is only necessary to read the actual direction bit:

❏ If the direction bit TIMACNT.0 is 0 (upward count), then the captured time (0 to nCCR0) in the capture/compare register x is added to the cycle count in TIMACYC0. The captured event occurred after the time stored in TIMACYC0.

❏ If the direction bit TIMACNT.0 is 1 (downward count), then the captured time (0 to nCCR0) in the capture/compare register x is subtracted from the cycle count in TIMACYC0. The captured event occurred before the time stored in TIMACYC0.

The sections *Digital Motor Control* and *TRIAC Control* also contain examples for the use of capturing with the up/down mode.
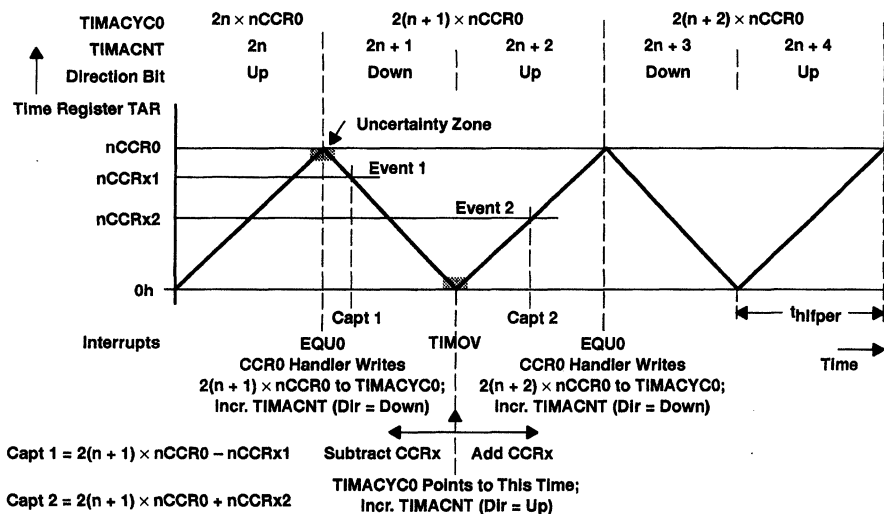
Figure 6–52. Capturing With the Up/Down Mode

Figure 6–53 illustrates the hardware and RAM registers used with the up/down mode for capturing. The RAM words TIM31 and TIM30 store the time of the last captured event. Figure 6–53 refers to the capture/compare block 3 of the following example.
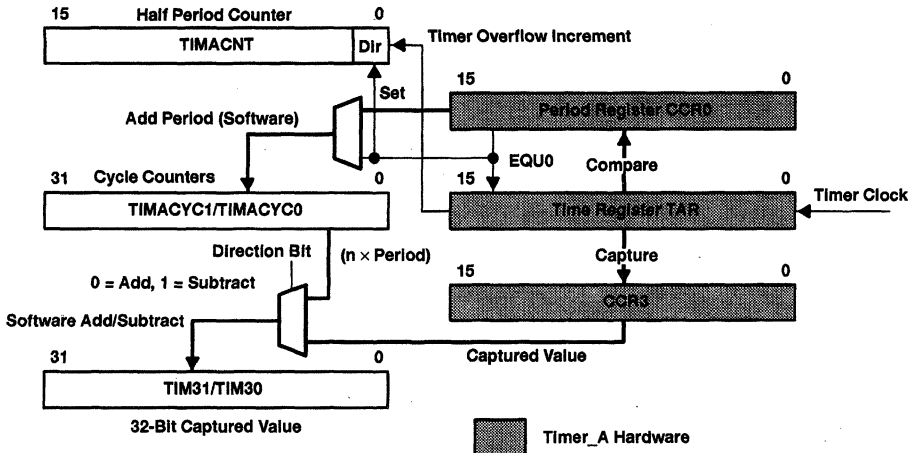
Figure 6–53. Capture Mode With the Up/Down Mode (Capture/Compare Block 3)

Figure 6–54 illustrates five tasks. They are exactly the same tasks that are used for the up mode in the section *Software Examples for the Up Mode* (only the capture/compare block 3 part — that captures the leading edge of an input signal — is extended to 32 bits). This way a comparison is possible between the up mode and the up/down mode. The tasks are defined as follows:

❏ **Capture/Compare Block 0** — outputs a symmetrical 8.484 kHz signal. The edges contain the information for the period generated by the period register CCR0. This signal is always available for external peripherals (the PWM signals of the capture/compare blocks disappear for pulse widths of 0% and 100%).

❏ **Capture/Compare Block 1** — generates a positive PWM signal with the half period defined by the period register CCR0. The pulse length is stored in the RAM word TA1PWM; it ranges from 1 to HLFPER.

❏ **Capture/Compare Block 2** — the length, $\Delta t2$, of the high part of the input signal at the CCI2A input terminal is measured and stored in the RAM word PP2. The captured time of the leading edge is stored in the RAM word TIM2. The maximum repetition rate used is 2 kHz.

❏ **Capture/Compare Block 3** — the event time of the leading edge of the signal at the CCI3A input terminal is captured. The last captured value (TCLK cycles, 32 bits length) is stored in the RAM words TIM30 and TIM31. The maximum repetition rate used is 3 kHz. See also figure 6–53.

❑ **Capture/Compare Block 4** — generates a negative PWM signal with the period defined by the period register. The pulse length is stored in the RAM word TA4PWM; it ranges from 0 to HLFPER.

For the example, 3.801 MHz is used. The resolution for the PWM is 224 steps due to the repetition frequency of 16.969 kHz (3.801 MHz/16.969 kHz = 224). The *Infrequent Individual Update Mode* is used. See Section 6.3.10.3 for details.

The maximum input frequencies for capturing purposes mentioned above are used for the overhead calculation only. The limits of the Timer_A hardware allow the capture much of higher input frequencies.

Figure 6–54 illustrates the four tasks described above — they are not shown to scale:
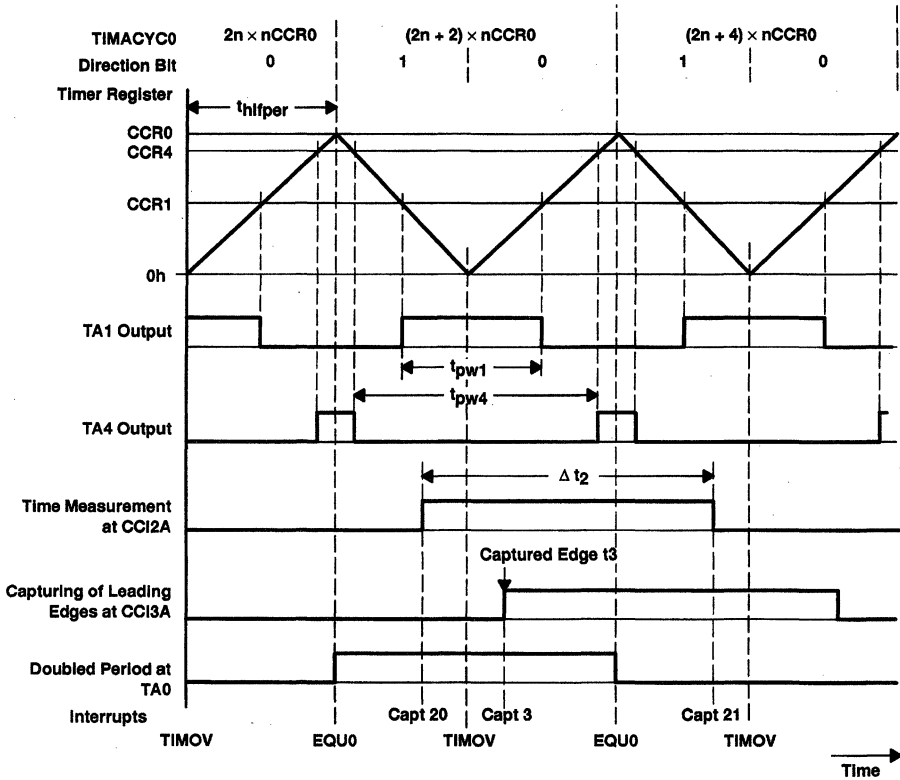
Figure 6–54. PWM Generation and Capturing With the Up/Down Mode

Example 6–51. Timer_A Used for PWM Generation and Capturing

```
; Timer_A used for PWM-generation and Capturing.
;
FLLMPY      .equ    116              ; fMCLK = 3.801MHz
fper        .equ    16969            ; 16.969kHz repetition rate
TCLK        .equ    FLLMPY*32768     ; TCLK: FLLMPY x fcrystal
HLFPER      .equ    (TCLK/fper)/2    ; fper = 16.969kHz
TAV0        .equ    0                ; MSP430C33x version
PERIOD_VAR  .equ    0                ; Fixed period
```

```
;
; RAM Definitions
;
TA1PWM    .equ    202h            ; PWM pulse length TA1
TIM2      .equ    204h            ; Time of leading edge at CCI2A
PP2       .equ    206h            ; Length of high part at CCI2A
TIM30     .equ    208h            ; Time of leading edge   LSBs
TIM31     .equ    20Ah            ; at CCI3A                MSBs
TA4PWM    .equ    20Ch            ; PWM pulse length for TA4
TIMACYC0  .equ    20Eh            ; Cycle counter low
TIMACYC1  .equ    210h            ; Cycle counter high
TIMACNT   .equ    212h            ; Half period counter. Bit0 = Dir
FLAG      .equ    214h            ; Update information
STACK     .equ    600h            ; Stack initialization address
          .text                   ; Software start address
INIT      MOV     #STACK,SP       ; Initialize Stack Pointer
          CALL    #INITSR         ; Init. FLL and RAM
;
; Initialize the Timer_A: MCLK, Up/Down Mode, INTRPTs on
; for TIMOV, C/C blocks 0, 2, and 3
;
          MOV     #ISMCLK+CLR+TAIE,&TACTL ; Define Timer_A
          MOV     #HLFPER,&CCR0      ; Define half period
          MOV     #OMT+CCIE,&CCTL0   ; Toggle TA0, INTRPT on
          MOV     #OMTR,&CCTL1       ; Toggle/Reset Mode
          MOV     #CMBE+ISCCIA+SCS+CAP+CCIE,&CCTL2 ; Both edges
          MOV     #CMPE+ISCCIA+SCS+CAP+CCIE,&CCTL3 ; Pos. edge
          MOV     #OMTS,&CCTL4       ; Toggle/Set Mode
          MOV.B   #TA4+TA3+TA2+TA1+TA0,&P3SEL ; Define I/Os
          MOV.B   #CBACLK+CBE,&CBCTL ; Output ACLK at XBUF pin
;
          CLR     TIMACYC0        ; Clear low cycle counter
          CLR     TIMACYC1        ; Clear high cycle counter
          CLR     TIMACNT         ; Clear half period counter
          MOV     #1,TA1PWM       ; TA1 pulse length = 1
          MOV     #0,TA4PWM       ; TA4 pulse length = 0
```

```
          MOV      #6,FLAG          ; Actualize PWMs immed.
          BIS      #MUPD,&TACTL     ; Start Timer in Up/Down Mode
          EINT                      ; Enable interrupts
MAINLOOP ...                        ; Continue in background
; Calculations for the new PWM values start.
; The new result in R6 is written to TA1PWM after completion.
; The PWM range is from 1 to HLFPER-1: no checks necessary
;
          ...                       ; Calculate TA1 value to R6
          MOV      R6,TA1PWM        ; Actualize pulse length
          BIS      #2,FLAG          ; Initiate update
          ...                       ; Continue in background
;
; The new result in R6 is written to TA4PWM after completion.
; The PWM range is from 0% to 100%: check necessary
;
          ...                       ; Calculate TA4 value to R6
          CHCK_PWM_RNG   R6         ; Check and correct result
          MOV      R6,TA4PWM        ; Actualize pulse length
          BIS      #4,FLAG          ; Initiate update
          ...                       ; Continue in background
;
; Use the measured high part in PP2 for calculations
;
          MOV      PP2,R7           ; Read measured pulse length
          ...                       ; Control algorithm
;
; Use the captured 32 bit value in TIM31/TIM30 for calculations
;
          MOV      TIM31,R7         ; Captured MSBs
          MOV      TIM30,R6         ; Captured LSBs
          ...                       ; Control algorithm
;
; Interrupt handler for the Period Register CCR0. 8.484kHz
; are output at TA0 for synchronization.
;
```

```
TIMMOD0  ADD      #2*HLFPER,TIMACYC0 ; Actualize cycle counters
         ADC      TIMACYC1
         BIS      #1,TIMACNT      ; Incr. half period counter
         RETI                     ; Dir = Down
;
; Interrupt handlers for Capture/Compare Blocks 1 to 4.
; The interrupt flags CCIFGx are reset by the reading
; of the Timer Vector Register TAIV
;
TIM_HND  ADD      &TAIV,PC        ; Add Jump table offset
         RETI                     ; Vector 0: No interrupt pending
         RETI                     ; C/C Block 1: INTRPT disabled
         JMP      TIMMOD2         ; C/C Block 2: Capt. both edges
         JMP      TIMMOD3         ; C/C Block 3: Capt. pos. edge
         RETI                     ; C/C Block 4: INTRPT disabled
;
; TIMOV Interrupt: dependent on FLAG the CCR1 and CCR4
; PWM registers are updated.
;
TIMOV    INC      TIMACNT         ; Incr. half period cnt (Down)
         ADD      FLAG,PC         ; FLAG with update info
         RETI                     ; 0: Nothing to do
         JMP      P1              ; 2: Update CCR1
         JMP      P4              ; 4: Update CCR4
         MOV      TA1PWM,&CCR1    ; 6: Update CCR1 and CCR4
P4       MOV      TA4PWM,&CCR4    ; 4:
         CLR      FLAG
         RETI
P1       MOV      TA1PWM,&CCR1    ; 2: Update CCR1
         CLR      FLAG
         RETI
;
; The high part of the CCI2A input signal is measured.
; The result is stored in PP2. The complete handler is time
; critical: nested interrupts cannot be used.
;
```

```
TIMMOD2  BIT     #CCI,&CCTL2      ; Input signal high?
         JZ      TM21             ; No, time for calculation
         MOV     TIMACYC0,TIM2    ; Build time of event
         BIT     #1,TIMACNT       ; Pos. edge: count direction Up?
         JNZ     T20              ; No, Down (1)
;                                   Direction is Up
         ADD     &CCR2,TIM2       ; Build time of pos. edge in TIM2
         RETI
;                                   Direction is Down
T20      SUB     &CCR2,TIM2       ; Build time of pos. edge in TIM2
         RETI
                                  ; Neg. edge: High part is calc.
TM21     MOV     TIMACYC0,PP2     ; Event time of trailing edge
         BIT     #1,TIMACNT       ; Direction Up?
         JNZ     T22              ; No, Down (1)
;                                   Direction is Up
         ADD     &CCR2,PP2        ; Time of trailing edge in PP2
         JMP     T23              ; To calculation of high part
;                                 ; Direction is Down
T22      SUB     &CCR2,PP2        ; Time of trailing edge in PP2
T23      SUB     TIM2,PP2         ; Subtr. time of leading edge
         RETI                     ; Length of high part in PP2
;
; Capture/Compare Block 3 captures the time of leading edges
; at CCI3A. TIM3x stores the 32 bit time of the actual edge
;
TIMMOD3  MOV     TIMACYC0,TIM30   ; Store cycle counters (32 bit)
         MOV     TIMACYC1,TIM31
         BIT     #1,TIMACNT       ; Count direction Up?
         JNZ     T30              ; No, Down (1)
;                                   Direction is Up
         ADD     &CCR3,TIM30      ; Time of pos. edge in TIM3x
         ADC     TIM31
         RETI
;                                 ; Direction is Down
T30      SUB     &CCR3,TIM30      ; Time of pos. edge in TIM3x
```

```
         SBC       TIM31
         RETI
;
         .sect     "TIMVEC",0FFF0h  ; Timer_A Interrupt Vectors
         .word     TIM_HND          ; C/C Blocks 1..4 Vector
         .word     TIMMOD0          ; Vector for C/C Block 0
         .sect     "INITVEC",0FFFEh ; Reset Vector
         .word     INIT
```

The above example results in a maximum (worst case) CPU loading u_{CPU} (ranging from 0 to 1) by the Timer_A activities:

CCR0 — repetition rate 16.969 kHz   13 cycles for the task, 11 cycles overhead   24 cycles

CCR1 — update rate 1 kHz   12 cycles for the update, 16 cycles overhead   28 cycles

CCR2 — rep. rate max. 2 kHz   64 cycles for the update, 32·cycles overhead   96 cycles

CCR3 — rep. rate max. 3.0kHz   30 cycles for the update, 16 cycles overhead   46 cycles

CCR4 — update rate 1.0kHz   12 cycles for the update, 0 cycles overhead   12 cycles

TIMOV — rep. rate 16.969kHz   7 cycles for the task, 14 cycles overhead   21 cycles

$$u_{CPU} = \frac{16.969 \times 10^3 \times 45 + 1.0 \times 10^3 \times 40 + 2.0 \times 10^3 \times 96 + 3.0 \times 10^3 \times 46}{3.801 \times 10^6} = 0.298$$

This results in a worst case CPU loading of approximate 29% due to the Timer_A activities.

6.3.10.9 Conclusion

This section demonstrated the possibilities of the Timer_A running in the up/down mode. Despite the dominance of the period register CCR0 and its changing direction during a period, it is possible to capture signals, compare time intervals, and create timings in a real-time environment — all this in parallel with the pulse width modulation generated with the up/down mode.

6.4 The Hardware Multiplier

The 16 × 16-bit hardware multiplier of the MSP430 family is detailed in the following sections. Function and modes are discussed, and proven application examples are given for this fast and versatile peripheral. Also shown is a comparison of the speed of solutions using this peripheral compared to pure software solutions. The hardware multiplier can also execute the *Signed Multiply and Accumulate* function. The register to be used for the Operand 1 has the address 136h. The function is the same as for the *Signed Multiply* function, except that the new product is added to the accumulated sum in the SumHi/Sum-Lo registers. The SumExt register indicates the sign of the accumulated sum. It is the user's responsibilty to ensure that no overflow can occur (by worst-case calculation of the factors used).

6.4.1 Function of the Hardware Multiplier

The hardware multiplier allows three different multiply operations (modes):

❏ The multiplication for unsigned 16-bit and 8-bit operands

❏ The multiplication for signed 16-bit and 8-bit operands

❏ The multiply-and-accumulate function (MAC) for unsigned 16-bit and 8-bit operands

Any mixture of operand lengths (16 bits and 8 bits) is possible. If assisting software is used, other operations are also possible — the signed *Multiply-and-Accumulate* function, for example.
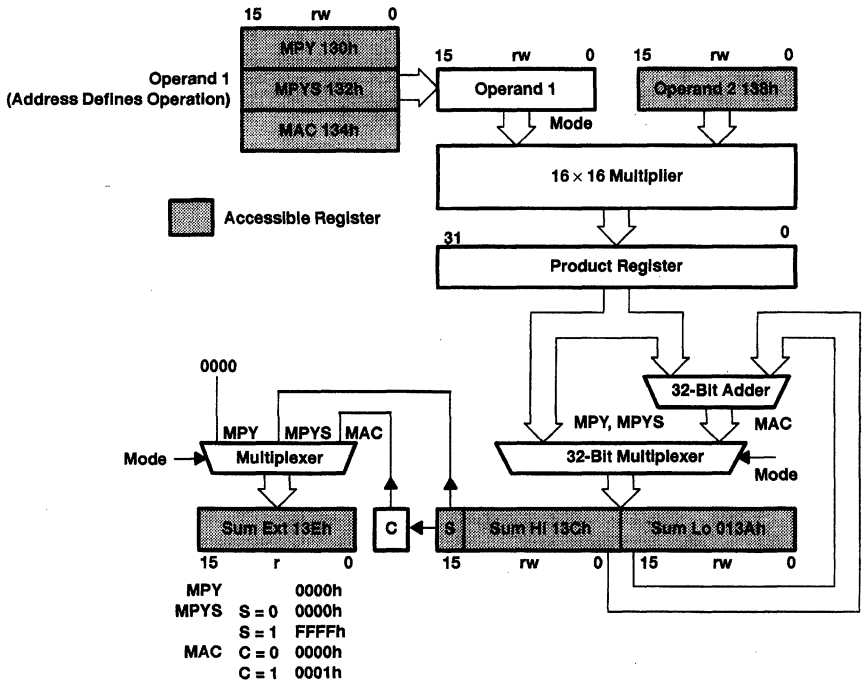
Figure 6–55. Block Diagram of the MSP430 16 × 16-Bit Hardware Multiplier

Figure 6–55 shows the hardware modules of the MSP430 multiplier. The accessible registers are explained in the following sections. The hardware of Figure 6–55 does not precisely depict the actual circuitry — it illustrates how the programmer sees the hardware multiplier.

6.4.1.1 Hardware and Register

The Hardware Multiplier is not part of the MSP430 CPU — it is a peripheral like the Timer_A or the Basic Timer. This means its activities do not interfere with the CPU activities. The multiplier registers are normal peripheral registers that are loaded and read with the CPU instructions. The registers that the programmer can access are explained in this section.

The hardware multiplier registers are not affected by POR or PUC.

With the exception of the SumExt register, all other registers can be read from and written to.

Definitions for the Hardware Multiplier appear in Section 6.4.3.
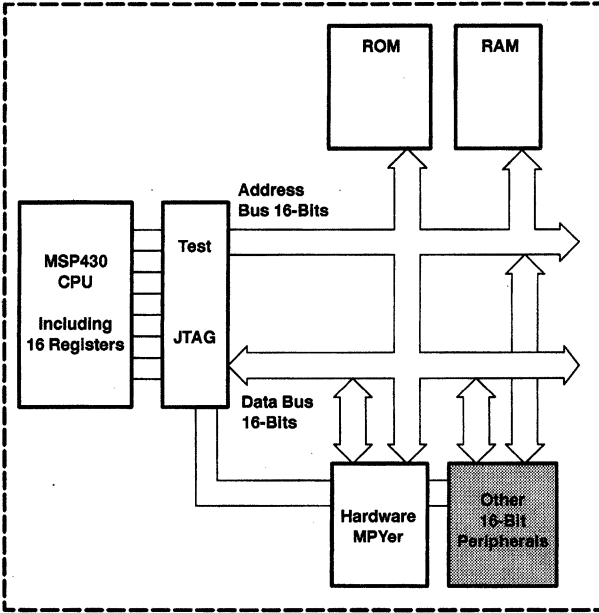
Figure 6–56. The Internal Connection of the MSP430 16 × 16-Bit Hardware Multiplier

6.4.1.2 The Operand1 Registers

The MSP430 hardware multiplier mode to be used is selected by the hardware address where the Operand 1 is written:

☐ **Address 130h** — the unsigned multiplication is executed

☐ **Address 132h** — the signed multiplication is executed

☐ **Address 134h** — the unsigned *Multiply–and–Accumulate* function is executed

Only the address used for the operand1 determines which operation the multiplier will execute (after the modification of the operand2). No operation is started with the modification of the operand register 1 alone.

Example 6–52. Multiply Unsigned

A MPY (multiply unsigned) operation is defined and started. The two operands reside in R14 and R15.

```
MOV     R15,&130h       ; Define MPY operation
MOV     R14,&138h       ; Start MPY with operand 2
...                     ; Product in SumHi|SumLo
```

6.4.1.3 The Operand2 Register

The operand register 2 (at address 138h) is common for all three multiplier modes. The modification of this register (normally with a MOV instruction) starts the selected multiplication of the two operands contained in the operand 1 and 2 registers. The result is written immediately into the three hardware registers: SumExt, SumHi, and SumLo. The result can be accessed with the next instruction unless the indirect addressing modes are used for the source addressing.

6.4.1.4 The SumLo Register

This 16-bit register contains the lower 16 bits of the calculated product or summed result. All instructions may be used to access or modify the SumLo register. The high byte cannot be accessed with byte instructions.

6.4.1.5 The SumHi Register

This 16-bit register contains — dependent on the previously executed operation — the following information:

❑ **MPY Unsigned Multiply** — the most significant word of the calculated product.

❑ **MPYS Signed Multiply** — the most significant word including the sign of the calculated product. Two's complement notation is used for the product.

❑ **MAC Unsigned Multiply-and-Accumulate** — the most significant word of the calculated sum.

All instructions may be used with the SumHi register. The high byte cannot be accessed with byte instructions.

6.4.1.6 The SumExt Register

The SumExt register (sum extension) eases the use of calculations with results exceeding the range of 32 bits. This *read only* register contains the information that is needed for the most significant parts of the result — the information for the bits 32 and higher. The content of the Sum Extension Register is different for the three multiplication modes:

❑ **MPY Unsigned Multiply** — SumExt always contains 0. No carry is possible and the maximum result possible is: 0FFFFh × 0FFFFh = 0FFFE0001h.

❑ **MPYS Signed Multiply** — SumExt contains the extended sign of the 32-bit result (bit 31). This means that if the result of the multiplication is negative (MSB = 1), then SumExt contains 0FFFFh. If the result is positive (MSB = 0), then SumExt contains 0000h.

❑ **MAC Unsigned Multiply-and-Accumulate** — SumExt contains the carry of the accumulate operation. SumExt contains 0001 if a carry occurred during the accumulation of the new product. SumExt contains 0 if no carry occurred.

The sum extension register improves multiple word operations. No time wasting and ROM–space wasting conditional jumps are necessary — ordinary adds are used instead.

The new product of a MPYS operation (multiplicands in R14 and R15) is added to a signed 64-bit result located in the RAM words RESULT to RESULT+6:

Example 6–53. 64-Bit Result

```
MOV     R15,&MPYS        ; First operand
MOV     R14,&OP2         ; Start MPYS with operand 2
ADD     SumLo,RESULT     ; Lower 16 bits of result
ADDC    SumHi,RESULT+2   ; Upper 16 bits
ADDC    SumExt,RESULT+4  ; Result bits 32 to 47
ADDC    SumExt,RESULT+6  ; Result bits 48 to 63
```

> **Note:**
>
> It is strongly recommended the MACROs defined in section *Assembler .MACROS* be used instead of the method shown above. The code above is much less descriptive than the MACROs, using known abbreviations like MPYU, MPYS and MACU.

With the software shown above, no checks and conditional jumps are necessary. The result always contains the signed, accumulated sum automatically.

6.4.1.7 Rules for the Hardware Multiplier

❏ The hardware multiplier is a word module. The hardware registers can be addressed in word mode or in byte mode, but the byte mode can address the lower bytes only (the upper byte cannot be addressed).

❏ The operand registers of the hardware multiplier (addresses 0130h, 0132h, 0134h and 0138h) behave like the CPU working registers R0 to R15 if modified in byte mode — the upper byte is cleared in this case. This allows 8-bit and 16-bit multiplications in any mixture. See the examples in Section 6.4.2.4.

❏ The foating point package (FPP) version 4 uses the hardware multiplier if the variable HW_MPY is defined as 1:

```
HW_MPY            .equ     1
```

See chapter 5.6 for details.

❏ If the result of a hardware multiplier operation is addressed with indirect mode or indirect-autoincrement mode, a NOP instruction is necessary after the multiplication to allow the completion of the multiplication. See the examples in Section 6.4.3.1.

6.4.2 Multiplication Modes

Three different multiplication modes are available. They are explained in the following sections.

6.4.2.1 Unsigned Multiply

The two operands written to the operand registers 1 and 2 are treated as unsigned numbers with:

▪ 00000h   as the smallest number

▪ 0FFFFh   as the largest number.

The maximum possible result is reached for the operands:

0FFFFh × 0FFFFh = 0FFFE0001h

No carry is possible, the SumExt register always contains 0. Table 6–27 shows the products for some special multiplicands.

Table 6–27. Results With the Unsigned Multiply Mode

OPERANDS	SumExt	SumHI	SumLo
0000 × 0000	0000	0000	0000
0001 × 0001	0000	0000	0001
7FFF × 7FFF	0000	3FFF	0001
FFFF × FFFF	0000	FFFE	0001
7FFF × FFFF	0000	7FFE	8001
8000 × 7FFF	0000	3FFF	8000
8000 × FFFF	0000	7FFF	8000
8000 × 8000	0000	4000	0000

6.4.2.2 Signed Multiply

The two operands written to the operand registers 1 and 2 are treated as signed two's complement numbers with:

- 08000h   as the most negative number (–32768)
- 07FFFh   as the most positive number (+32767)

The SumExt register contains the extended sign of the calculated result:

- SumExt = 00000h: the result is positive
- SumExt = 0FFFFh: the result is negative

Table 6–28. Results With the Signed Multiply Mode

OPERANDS	SumExt	SumHI	SumLo
0000 × 0000	0000	0000	0000
0001 × 0001	0000	0000	0001
7FFF × 7FFF	0000	3FFF	0001
FFFF × FFFF	0000	0000	0001
7FFF × FFFF	FFFF	FFFF	8001
8000 × 7FFF	FFFF	C000	8000
8000 × FFFF	0000	0000	8000
8000 × 8000	0000	4000	0000

6.4.2.3 Multiply-and-Accumulate (MAC)

The two operands written to the operand registers 1 and 2 are treated as unsigned numbers (0h to 0FFFFh). The maximum possible result is reached for the input operands:

$$0FFFFh \times 0FFFFh = 0FFFE0001h$$

This result is added to the previous content of the two sum registers (SumLo and SumHi). If a carry occurs during this operation, the SumExt register contains 1, otherwise it is cleared.

- SumExt = 00000h: no carry occurred during the accumulation
- SumExt = 00001h: a carry occurred during the accumulation

For the results of Table 6–29, it is assumed that SumHi and SumLo contain the accumulated content C000,0000 before the execution of each of the shown examples. See Table 6–27 for the results of an unsigned multiplication without accumulation.

Table 6–29. Results With the Unsigned Multiply-and-Accumulate Mode

OPERANDS	SumExt	SumHi	SumLo
0000 × 0000	0000	C000	0000
0001 × 0001	0000	C000	0001
7FFF × 7FFF	0000	FFFF	0001
FFFF × FFFF	0001	BFFE	0001
7FFF × FFFF	0001	3FFE	8001
8000 × 7FFF	0000	FFFF	8000
8000 × FFFF	0001	3FFF	8000
8000 × 8000	0001	0000	0000

6.4.2.4 Word Lengths for the Multiplication

The MSP430 hardware multiplier allows all combinations that are possible with 8-bit and 16-bit operands. The examples given in Section 6.4.3 for 8-bit and 16-bit operands may be adapted to mixed length operands.

It must be taken into account that the input operand registers operand1 and operand2 behave like CPU registers — the high register byte is cleared if the register is modified by a byte instruction.

This eases the use with 8-bit operands. Examples for the 8-bit operand are given for all three modes of the hardware multiplier.

```
; Use the 8-bit operand in R5 for an unsigned multiply.
;
        MOV.B   R5,&MPY          ; The high byte is cleared
; Use an 8-bit operand for a signed multiply.
;
        MOV.B   R5,&MPYS         ; The high byte is cleared
        SXT     &MPYS            ; Extend sign to high byte
; Use an 8-bit operand for a multiply-and-accumulate.
;
        MOV.B   R5,&MAC          ; The high byte is cleared
```

Operand2 is loaded as shown above for operand1. This allows all four possible combinations for the input operands:

$16 \times 16 \quad 8 \times 16 \quad 16 \times 8 \quad 8 \times 8$

The MACROS that can be modified are shown in the next section.

6.4.3 Programming the Hardware Multiplier

At the beginning, the registers of the hardware multiplier are defined in accordance with the *MSP430 Family Architecture Guide and Module Library*. This avoids confusion.

```
; MSP430 Hardware Multiplier Definitions
;
MPY      .equ    130h            ; Multiply unsigned
MPYS     .equ    132h            ; Multiply signed
MAC      .equ    134             ; Multiply-and-Accumulate
OP2      .equ    138h            ; Operand 2 Register
;
SumLo    .equ    013Ah           ; Result Register LSBs 15..0
SumHi    .equ    013Ch           ; Result Register MSBs 32..16
SumExt   .equ    013Eh           ; Sum Extension Register 47..33
```

6.4.3.1 Assembler .MACROS

Due to the MACRO construction of the multiply instructions for source and destination (normally two MOV instructions form the multiplication sequence), all seven addressing modes are possible. If the *register indirect* or *register indirect with autoincrement* addressing modes are used to address the result, then a NOP is necessary after the .MACRO call to allow the completion of the multiplication. The named addressing modes access the source operand so fast, that they do not allow the completion of the multiplication.

Examples are given with each .MACRO definition. The execution cycles depend on the addressing modes used for the multiplier and the multiplicand.

The given MACROs can easily be changed to subroutines. An example is given for the unsigned multiplication:

```
; Subroutine Definition for the unsigned multiplication
; 16 x 16 bits. The two operands are contained in R4 and R5
;
MPYU_16  MPYU16   R4,R5            ; Unsigned MPY 16 x 16
         RET                       ; Result in SumHi|SumLo
;
```

6.4.3.2 Unsigned Multiplication 16 x 16-bits

```
; Macro Definition for the unsigned multiplication 16 x 16 bits
;
MPYU16   .MACRO   arg1,arg2        ; Unsigned MPY 16x16
         MOV      arg1,&0130h
         MOV      arg2,&0138h
         .ENDM                     ; Result in SumHi|SumLo
;
; Multiply the contents of the two registers R4 and R5
;
         MPYU16   R4,R5            ; MPY R4 and R5 unsigned
         MOV      SumLo,R6         ; LSBs of result to R6
         MOV      SumHi,R7         ; MSBs of result to R7
         ...                       ; Continue
;
; Multiply the contents located in a table, R6 points to
; The result is addressed in indirect mode: a NOP is necessary
```

```
; to allow the completion of the multiplication
;
        MOV      #SumLo,R5          ; Pointer to LSBs of result
        MPYU16   @R6+,@R6           ; MPYU the table contents
        NOP                         ; Allow completion of MPYU16
        MOV      @R5+,R7            ; Fetch LSBs of result
        MOV      @R5,R8             ; Fetch MSBs of result
        ...                         ; Continue
;
; Macro Definition for the unsigned multiplication and
; accumulation 16 x 16 bits
;
MACU16  .MACRO   arg1,arg2          ; Unsigned MAC 16x16
        MOV      arg1,&0134h        ; Carry in SumExt
        MOV      arg2,&0138h
        .ENDM                       ; Result in SumExt|SumHi|SumLo
;
; Multiply-and-accumulate the contents of registers R5 and R6
; to the previous content (IROP1 x IROP2L) of the Sum registers
;
        MPYU16   IROP1,IROP2L       ; Initialize Sum registers
        MACU16   R5,R6              ; Add (R5 x R6) to result
        ADD      &SumExt,RAM        ; Add carry to RAM extension
        ...                         ; Continue
```

6.4.3.3 Signed Multiplication 16 × 16-bit

The following software examples perform signed 16 × 16-bit multiplications (MPYS16) or signed *Multiplication and Accumulation* (MACS16).

The SumExt register contains the extended sign of the result in SumHi and SumLo: 0000h (positive result) or 0FFFFh (negative result).

```
; Macro Definition for the signed multiplication 16 x 16 bits
;
MPYS16  .MACRO   arg1,arg2          ; Signed MPY 16x16 bits
        MOV      arg1,&0132h
        MOV      arg2,&0138h
```

```
        .ENDM                     ; Result in SumExt|SumHi|SumLo
;
; Multiply the contents of two registers R4 and R5
;
        MPYS16   R4,R5            ; MPY signed R4 and R5
        MOV      &SumLo,R6        ; LSBs of result to R6
        MOV      &SumHi,R7        ; MSBs of result to R7
        MOV      &SumExt,R8       ; Sign of result to R8
        ...                       ; Continue
;
; Multiply the contents located in a table, R6 points to
; The result is addressed in indirect mode: a NOP is necessary
; to allow the completion of the multiplication
;
        MOV      #SumLo,R5        ; Pointer to LSBs of result
        MPYS16   @R6+,@R6         ; MPY signed table contents
        NOP                       ; Allow completion of MPYS16
        MOV      @R5+,R7          ; LSBs of result to R7
        MOV      @R5+,R8          ; MSBs of result to R8
        MOV      @R5,R9           ; Sign of result to R9
        ...                       ; Continue
;
; Macro Definition for the signed multiplication-and-
; accumulation 16 x 16 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further extensions (RAM or registers) here shown for only
; one extension (48 bits).
;
MACS16  .MACRO   arg1,arg2        ; Signed MAC 16x16 bits
        MOV      arg1,&0132h      ; Signed MPY is used
        MOV      arg2,&0138h
        ADD      &SumLo,MAClo     ; Add LSBs to result
        ADDC     &SumHi,MACmid    ; Add MSBs to result
        ADDC     &SumExt,MAChi    ; Add SumExt to MSBs
        .ENDM
```

```
;
; Multiply and accumulate signed the contents of two tables
;
        MACS16    2(R6),@R5+        ; MACS for the table contents
        ....                        ; Accumulation is yet made
```

6.4.3.4 Unsigned Multiplication 8 × 8-bits

If byte instructions are used for the loading of the hardware multiplier registers, then the high byte of these registers is cleared like a CPU register. This behavior is used with the unsigned 8 × 8-bits multiplications.

```
; Macro Definition for the unsigned multiplication 8 x 8 bits
;
MPYU8    .MACRO    arg1,arg2        ; Unsigned MPY 8x8
         MOV.B     arg1,&0130h      ; 00xx to 0130h
         MOV.B     arg2,&0138h      ; 00yy to 0138h
         .ENDM                      ; Result in SumLo. SumHi = 0
;
; Multiply the contents of the low bytes of two registers
;
         MPYU8     R12,R15          ; MPY low bytes of R12 and R15
         MOV       &SumLo,R6        ; 16 bit result to R6
         ...                        ; SumExt = SumHi = 0
;
; Macro Definition for the unsigned multiplication-and-
; accumulation 8 x 8 bits
;
MACU8    .MACRO    arg1,arg2        ; Unsigned MAC 8x8
         MOV.B     arg1,&0134h      ; 00xx
         MOV.B     arg2,&0138h      ; 00yy
         .ENDM                      ; Result in SumExt|SumHi|SumLo
;
; Multiply-and-accumulate the low bytes of R14 and a table
;
         MACU8     R14,@R5+         ; CALL the MACU8 macro (R5+1)
```

6.4.3.5 Signed Multiplication 8 × 8-bits

If byte instructions are used for the loading of the hardware multiplier registers, then the high bytes of their registers are cleared like a CPU register. It therefore needs only to be sign-extended.

```
; Macro Definition for the signed multiplication 8 x 8 bits
;
MPYS8    .MACRO   arg1,arg2       ; Signed MPY 8x8
         MOV.B    arg1,&0132h     ; 00xx
         SXT      &0132h          ; Extend sign: 00xx or FFxx
         MOV.B    arg2,&0138h     ; 00yy
         SXT      &0138h          ; Extend sign: 00yy or FFyy
         .ENDM                    ; Result in SumExt|SumHi|SumLo
;
; Multiply signed the low bytes of R5 and location EDE
;
         MPYS8    R5,EDE          ; CALL the MPYS8 macro
         MOV      &SumLo,R6       ; Fetch result (16 bits)
         MOV      &SumHi,R7       ; Sign: 0000 or FFFF
;
; Macro Definition for the signed multiplication and
; accumulation 8 x 8 bits. The accumulation is made in the
; locations MACHi, MACmid and MAClo (registers or RAM)
; If more than 48 bits are used for the accumulation, the
; SumExt register is added to all further RAM extensions
;
MACS8    .MACRO   arg1,arg2       ; Signed MAC 8x8 bits
         MOV.B    arg1,&0132h     ; MPYS is used
         SXT      &0132h          ; Extend sign: 00xx or FFxx
         MOV.B    arg2,&0138h     ; 00yy
         SXT      &0138h          ; Extend sign
         ADD      &SumLo,MAClo    ; Accumulate LSBs 16 bits
         ADDC     &SumHi,MACmid   ; Accumulate MIDs
         ADDC     &SumExt,MAChi   ; Add SumExt to MSBs
         .ENDM                    ;
;
; Multiply-and-accumulate signed the contents of two byte
```

```
; tables
;
         MACS8    2(R6),@R5+         ; CALL the MACS8 macro (R5+1)
         ....                        ; Accumulation is yet made
```

6.4.3.6 Interrupt Usage

Operating in the foreground only (interrupt handlers), the hardware multiplier can be used freely. If the hardware multiplier is used in the foreground *and* the background, or in nested interrupt handlers, however, there are additional considerations.

The hardware multiplier may be used in interrupt handlers and in the background (which is not typical real-time programming practice), if three rules are observed:

❑   The loading of the two registers operand1 (MPY, MPYS and MAC) and operand2 may not be separated by an interrupt using the multiplier. The input information for operand1 cannot be restored due to the three input registers that are possible. See the example below.

❑   The registers operand1 and operand2 cannot be reread by the background software — they may be overwritten by the interrupt handler.

❑   The operand1 information cannot be used for more than one multiplication — only the operand2 register is changed for the next multiplication. The floating point package, FPP4, uses this method to speed up the calculation, so it must be changed. The place is indicated.

```
; Background: multiplication is used together with interrupt
; The interrupt latency time is increased by 9 cycles.
; The NOP is necessary: one additional instruction may
; be executed after the DINT instruction
;
         DINT                       ; Ensure non-interrupted -
         NOP                        ; load of the MPYer registers
         MPYU16   R4,R6             ; (R4) x (R6) -> Sum
         EINT                       ; Allow interrupts again
         ...                        ; Continue with result
; The interrupt handler must save and restore the Sum registers
;
INTRPT_H PUSH     &SumLo            ; Save the SumLo register
```

```
PUSH    &SumHi              ; Save the SumHi register

PUSH    &SumExt             ; Save the SumExt register

MPYU16  #X,C1               ; Call unsigned MPY: X x C1

....                        ; Continue with MPYer result

POP     &SumExt             ; Restore SumExt register

POP     &SumHi              ; SumHi register

POP     &SumLo              ; SumLo register

RETI                        ; Return to background
```

6.4.3.7 Speed Comparison with Software Multiplication

Table 6–30 shows the speed increase for the different 16 × 16-bit multiplication modes.

❏ The cycles given for the software loop include the subroutine call (CALL #MULxx), the subroutine itself, and the RET instruction. Only CPU registers are used for the multiplication.

❏ The cycles given for the hardware multiplier include the loading of the multiplier operand registers operand1 and operand2 from CPU registers, and — in the case of the signed MAC operation — the accumulation of the 48-bit result to three CPU registers (see Section 6.4.3.1.2).

Table 6–30. CPU Cycles Needed for the Different Multiplication Modes

OPERATION	SOFTWARE LOOP	HARDWARE MPYer	SPEED INCREASE
Unsigned Multiply MPY	139...171	8	17.4...21.4
Unsigned MAC	137...169	8	17.1...21.1
Signed Multiply MPY	145...179	8	18.1...22.4
Signed MAC	143...177	17	8.4...10.4

6.4.3.8 Software Hints

If the operand1 is used for more than one multiplication in sequence, then it is not necessary to move it again into the operand1 register. The first example shows two unsigned multiplications with the content of address TONI. Four bytes and six CPU cycles are saved compared to the normal procedure.

```
; Multiply TONI x R6 and TONI x R5. Results to diff. locations

;

MPYU16  TONI,R6             ; TONI x R6 -> SumHi|SumLo
```

```
MOV      &SumLo,R7          ; Result to R8|R7
MOV      &SumHi,R8
MOV      R5,&0138h          ; TONI still in &0130h
MOV      &SumLo,RESULT      ; TONI x R5 -> SumHi|SumLo
MOV      &SumHi,RESULT+2;   Result to RESULT+2|RESULT
```

The second example shows three multiply-and-accumulate operations with the same operand1. The three operands2 cannot be added simply and multiplied once — their sum may exceed the range of 16 bits. Eight ROM bytes and twelve CPU cycles are saved by using this method compared to the normal procedure.

```
; Multiply-and accumulate TONI x R6, TONI x R5 and TONI x EDE
; The accumulated result is moved to RESULT..RESULT+4
;
...                            ; Initialize SumXxx registers
MACU16   TONI,R6              ; TONI x R6 + SumHi|SumLo
ADD      &SumExt,RESULT+4 ; Add carry to extension
MOV      R5,&0138h            ; Add TONI x R5 to SumXxx
ADD      &SumExt,RESULT+4 ; Add carry to extension
MOV      EDE,&0138h           ; Add TONI x EDE to SumXxx
MOV      &SumLo,RESULT        ; TONI x (R5+R6+EDE) in SumXxx
MOV      &SumHi,RESULT+2      ; Result to RESULT..RESULT+4
ADD      &SumExt,RESULT+4
```

6.4.3.9   Speed Increase for the Floating Point Package

The hardware multiplier only increases the speed of floating point multiplication. For the speed evaluation shown, the variables X and Y are used. They are defined as follows:

```
   .if      DOUBLE=0            ; 32-bit format
X  .float   3.1416             ; 3.1416
Y  .float   3.1416*100         ; 314.16
   .else                       ; 48-bit format
X  .double  3.1416             ; 3.1416
Y  .double  3.1416*100         ; 314.16
   .endif
```

The execution cycles shown include the addressing of one operand and the subroutine CALL, itself:

```
MOV     #X,RPRES        ; Address 1st operand
MOV     #Y,RARG         ; Address 2nd operand
CALL    #FLT_MUL        ; Call the MPY subroutine
....                    ; Product X x Y on TOS
```

Table 6–31 shows the number of necessary cycles needed for the multiplication:

Table 6–31. CPU Cycles Needed for the FPP Multiplication (FLT_MUL)

OPERATION	.FLOAT	.DOUBLE	COMMENT
Multiplication X × Y	395	692	Software loop
Multiplication X × Y	153	213	Hardware MPYer used
Speed increase	2.58	3.25	SW cycles/HW cycles

Due to the speed advantage of the hardware multiplier only for multiplication, it is recommended that divisions be replaced by multiplications wherever possible. This is most simple for divisions by constants, like is shown in the next example.

Example 6–54. Division by Multiplication

The division of the last result — on top of the stack — by the constant 2.7182818 is replaced by a multiplication with the constant 1/2.7182818. This reduces the calculation time by a factor of 405/153 = 2.65. First, the original sequence:

```
DOUBLE  .equ    0               ; Use the .FLOAT format
HW_MPY  .equ    1               ; Use the HW-MPYer
        ...
        MOV     #FLTe,RARG      ; Address constant e
        CALL    #FLT_DIV        ; TOS/e: Division 405 cycl.
        ....                    ; Quotient on TOS
FLTe    .float  2.7182818       ; Constant e
```

The above division is replaced by a multiplication using the hardware multiplier:

```
HW_MPY  .equ    1               ; Use the HW-MPYer
        ...
        MOV     #FLTei,RARG     ; Address constant 1/e
```

```
        CALL     #FLT_MUL           ; TOS x 1/e. MPY 153 cycles

        ....                        ; Result on TOS
FLTei   .float   0.3678794          ; Constant 1/e
```

If the .DOUBLE version (48 bits) of FPP4 is used, then the division execution time is decreased by a factor of 756/213 = 3.55.

6.4.4 Software Applications

Typical proven software examples are given for the application of the hardware multiplier. The comments indicate for some examples the location of the (think hexa–) decimal point:
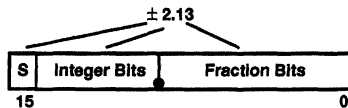
6.4.4.1 Multiplication Exceeding 16 Bits

The first software example shows the unsigned multiplication of two 40-bit numbers (the MSBytes contain 0) — 48 bits of the result are used subsequently. The lower 32 bits of the product are not used. The first operand is contained in the registers ARG1_xxx and the second operand in ARG2_xxx. The result is placed into RESULT_xxx (CPU registers or RAM). The multiply routine is abstracted from the FPP4 package.

The execution time for CPU registers is 94 cycles.
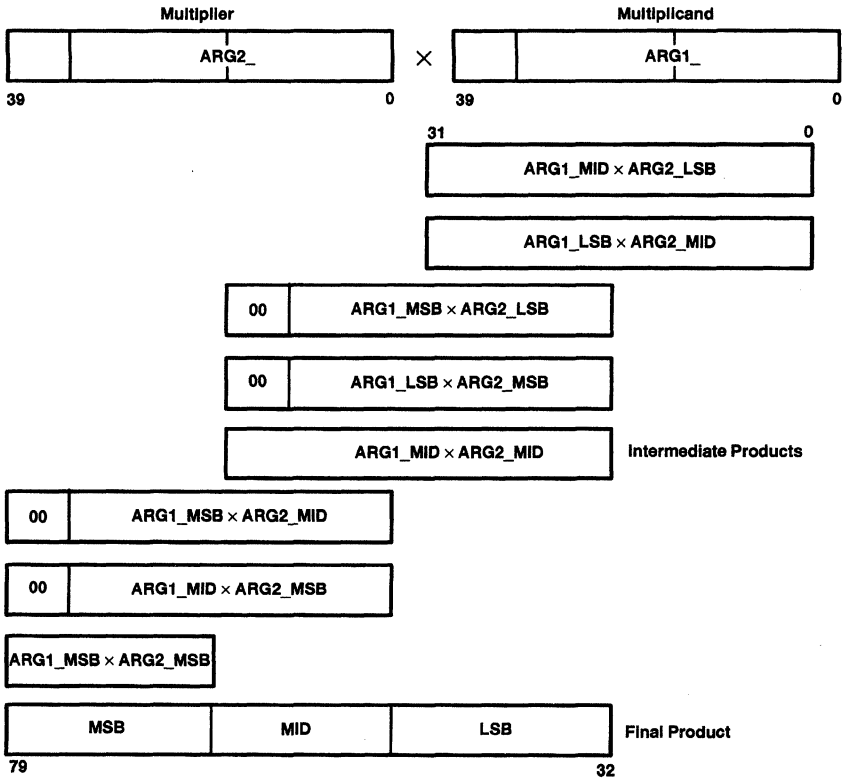
Figure 6–57. 40 × 40-Bit Unsigned Multiplication MPYU40

```
; Register Definitions for the 40 x 40 unsigned MPY and MAC
;
ARG1_MSB .equ      R5                 ; Argument 1 (Multiplicand)
ARG1_MID .equ      R6
ARG1_LSB .equ      R7
ARG2_MSB .equ      R8                 ; Argument 2 (Multiplier)
ARG2_MID .equ      R9
ARG2_LSB .equ      R10
RESULT_MSB .equ    R11                ; Result (Product)
```

```
RESULT_MID .equ   R12
RESULT_LSB .equ   R13
;
MPYU40    CLR     RESULT_MSB        ; Clear Result
          CLR     RESULT_MID
          CLR     RESULT_LSB
;
MACU40    MPYU16  ARG2_LSB,ARG1_MID ; Bits 16 to 47
          MACU16  ARG1_LSB,ARG2_MID
          ADD     &SumHi,RESULT_LSB
          ADDC    &SumExt,RESULT_MID
          MPYU16  ARG1_MSB,ARG2_LSB ; Bits 32 to 63
          MACU16  ARG1_LSB,ARG2_MSB
          MACU16  ARG1_MID,ARG2_MID
          ADD     &SumLo,RESULT_LSB
          ADDC    &SumHi,RESULT_MID
          ADDC    &SumExt,RESULT_MSB
          MPYU16  ARG1_MSB,ARG2_MID ; Bits 48 to 79
          MACU16  ARG2_MSB,ARG1_MID
          ADD     &SumLo,RESULT_MID
          ADDC    &SumHi,RESULT_MSB
          MPYU16  ARG1_MSB,ARG2_MSB ; Bits 64 to 79
          ADD     &SumLo,RESULT_MSB
          RET                       ; 48 MSBs in result
```

The second software example shows all four possible multiplication routines for two 32-bit numbers; the full 64-bit result may be used afterward. The signed 16×16-bit hardware multiplication MPYS cannot be used; it is designed for the special case of 16 × 16 bits. So the unsigned multiplication MPY is used with a correction of the final sum at the start of the subroutine.

Execution times (without CALL):

MACU32	58 cycles	unsigned MAC
MPYU32	64 cycles	unsigned MPY
MACS32	64 to 68 cycles	signed MAC
MPYS32	68 to 72 cycles	signed MPY
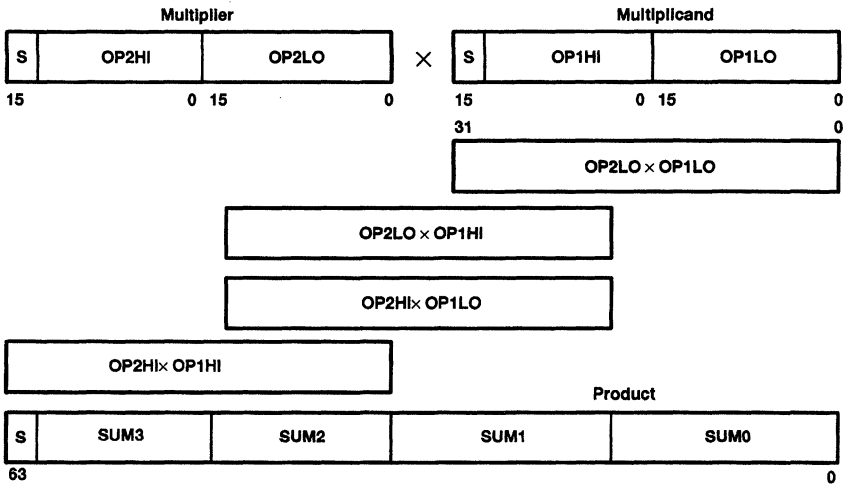
Figure 6–58. 32 × 32-Bit Signed Multiplication MPYS32

Example 6–55. 32 x 32-bit Multiplication and MAC Functions

All four possible 32 × 32-bit multiplication and MAC functions are shown below. The defined operands and result registers maybe working registers (as defined) or RAM locations.

```
SUM3      .equ    R15             ; Result: sign and MSBs
SUM2      .equ    R14             ; (registers or RAM locations)
SUM1      .equ    R13
SUM0      .equ    R12             ; LSBs
OP1HI     .equ    R11             ; 1st operand: sign and MSBs
OP1LO     .equ    R10             ; LSBs
OP2HI     .equ    R9              ; 2nd operand: sign and MSBs
OP2LO     .equ    R8          ,   ; LSBs
;
; The unsigned 32 x 32 bit multiplication
;
MPYU32    CLR     SUM3            ; Clear the result registers
          CLR     SUM2            ; 64 cycles
          CLR     SUM1
```

```
        CLR     SUM0
        JMP     MS321           ; Proceed at common part
;
; The signed 32 x 32 bit multiplication
;
MPYS32  CLR     SUM3            ; Clear the result registers
        CLR     SUM2            ; 68 to 72 cycles
        CLR     SUM1
        CLR     SUM0
;
; The signed 32-bit "Multiply-and-Accumulate" subroutine
; The final result is corrected. 64 to 68 cycles
;
MACS32  TST     OP1HI           ; Operand1 negative?
        JGE     MS320           ; No
        SUB     OP2LO,SUM2      ; Yes, correct final sum
        SUBC    OP2HI,SUM3
MS320   TST     OP2HI           ; Operand2 negative?
        JGE     MS321           ; No
        SUB     OP1LO,SUM2      ; Yes, correct final sum
        SUBC    OP1HI,SUM3
;
; The unsigned 32-bit "Multiply-and-Accumulate" subroutine
;
MACU32  .equ    $               ; 58 cycles
;
; Main part for all multiplication subroutines
;
MS321   MPYU16  OP1LO,OP2LO     ; LSBs x LSBs
        ADD     &SumLo,Sum0     ; Add product to result
        ADDC    &SumHi,Sum1
;
        ADC     Sum2            ; Necessary only for MACx32
        ADC     Sum3            ;
;
        MPYU16  OP1LO,OP2HI     ; LSBs x MSBs
```

```
        MACU16   OP2LO,OP1HI    ; LSBs x MSBs
        ADD      &SumLo,Sum1    ; Add accumulated products
        ADDC     &SumHi,Sum2    ; to result
;
        ADDC     &SumExt,Sum3   ; Necessary only for MACx32
;
        MPYU16   OP1HI,OP2HI    ; MSBs x MSBs
        ADD      &SumLo,Sum2    ; Add product to final result
        ADDC     &SumHi,Sum3
        RET
```

6.4.4.2 Sensor Characteristics

For many applications, the digital values delivered by analog-to-digital converters, I/O ports, or calculation results must be corrected or adapted. A common method is to use polynomials for this purpose. For example a cubic polynomial to calculate the corrected output value y from the input value x is:

$$y = a_3 \times x^3 + a_2 \times x^2 + a_1 \times x + a_0$$

With the hardware multiplier, a common solution may look like the following code. This subroutine is written for the highest possible speed — the coefficients a_3 to a_0 have decreasing numbers of bits after the (think hexa–) decimal point. If this cannot be tolerated, then shifts and stores between the multiplications are necessary. The input value x stays in operand1 (MPYS 0132h) and is used for all three multiplications.

Example 6–56. Value Correction

The output value of the ADC is corrected with a cubic polynomial. All values are scaled to values less than 1 to get the maximum resolution. The coefficients a_n used for correction are:

a3: +0.01   a2: –0.25   a1: –0.5   a0: +0.999

The HORNER scheme is used for the computation:

$$y = \left(\left(\left(a_3 \times x\right) + a_2\right) \times x + a_1\right) \times x + a_0$$

The numbers +–a.b in the code comments indicate the bits before and after the decimal point of the numbers used.

Execution time (without CALL): 45 cycles

```
;
; Polynomial Calculation for y = a3x^3 +a2x^2 +a1x^1 +a0x^0
; Result in SumHi register
;
POLYNOM   MPYS16    X,A3              ; +-0.15 x +-0.15 (+-1.14)
          ADD       A2,&SumHi         ; +-1.14 + +-1.14 -> +-1.14
          MOV       &SumHi,&OP2       ; +-1.14 x +-0.15 (+-2.13)
          ADD       A1,&SumHi         ; +-2.13 + +-2.13 -> +-2.13
          MOV       &SumHi,&OP2       ; +-2.13 x +-0.15 (+-3.12)
          ADD       A0,&SumHi         ; +-3.12 + +-3.12 -> +-3.12
          RET                         ; SumHi: +-3.12
;
; Table of coefficients
;
A3        .word     +100*08000h/10000 ; +0.01  (+-0.15)
A2        .word     -2500*04000h/10000 ; -0.25 (+-1.14)
A1        .word     -5000*02000h/10000 ; -0.5  (+-2.13)
A0        .word     +9999*01000h/10000 ; +0.9999 (+-3.12)
```

6.4.4.3 Table Calculation

The .MACRO instructions used for the different multiplication possibilities (8 bits versus 16 bits, signed and unsigned, multiply and multiply-and-accumulate) have the advantage to allow all seven addressing modes of the MSP430 architecture for source and destination. Therefore, the MPY instructions are ideal for table processing — both operands of a multiply instruction can also be addressed indirectly. An example for the table calculation is given in Section 6.4.4.5

6.4.4.4 Wave Digital Filters

The main advantage of wave digital filters is that for fixed coefficients, no multiplication is needed. Instead, an optimized shift-and-add sequence is used for the filter algorithm. But this optimization is not possible if *Adaptive Filter Algorithms* are used, which means changing coefficients. In this case, a hardware multiplier has significant advantages — the calculation time is independent of the coefficients used.

6.4.4.5 Finite Impulse Response (FIR) Digital Filter

The formula for a simple FIR filter is:

$$y_n = a0 \times x_n + a1 \times x_{n-1} + a2 \times x_{n-2} + \dots ak \times x_{n-k}$$

$$y_n = a_0 x_n + a_1 x_{n-1} \dots + a_k x_{n-k}$$
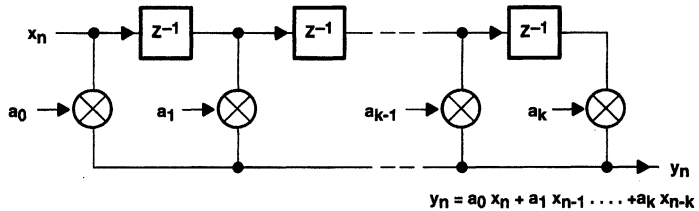
Figure 6–59.   Finite Impulse Response Filter

The example below shows an algorithm that uses the last ADC result for the input of a seventh-order FIR filter. The coefficients a_n are stored in ROM (fixed coefficients) or in RAM (adaptable coefficients). The filter maybe changed easily to a higher order:

❑ The value k must be changed to the desired order

❑ (k+1) words in RAM must be allocated for the input samples x_n starting at label X

❑ The table with the coefficients a_n must be enlarged to (k+1) coefficients

Execution time: 28 CPU cycles are necessary per filter tap.

The example does not show a real filter — for example, for a linear phase response the coefficients a_n must be:

$$a_n = a_{k-n}$$

which means: $a_0 = a_k$, $a_1 = a_{k-1}$ etc.

$$y_n = a_0 x_n + a_1 x_{n-1} \cdots + a_k x_{n-k}$$
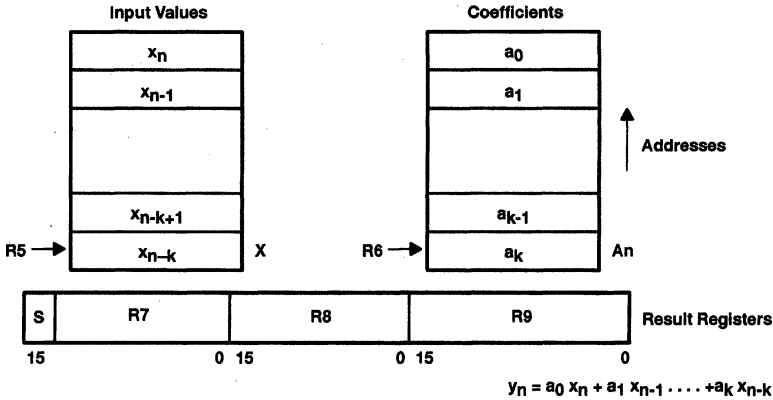
Figure 6–60. Storage for the Finite Impulse Response Filter

```
; The special "Multiply-and-Accumulate" .MACRO accumulates the
; products X x An in the registers R7|R8|R9.
; Execution time: 19 cycles for the example below with the
; indirect addressing mode used for both operands.
;
MACS16   .MACRO   arg1,arg2        ; Signed MAC 16x16
         MOV      arg1,&0132h      ; Signed MPY is used
         MOV      arg2,&0138h      ; Start MPYS
         ADD      &SumLo,R9        ; Add LSBs to result
         ADDC     &SumHi,R8        ; Add MSBs to result
         ADDC     &SumExt,R7       ; Add SumExt to result
         .ENDM                     ; Result in R7|R8|R9
; Definitions:
; - Value k defines the order of the FIR-filter
; - OFFSET is used to get signed values (E000h..1FFFh) out of
;   the unsigned 14-bit ADC results (0...3FFFh)
; - X defines the address for the oldest input sample x(n-k)
;   in a sample buffer with (k+1) words length
;
k        .equ     7               ; (k + 1)samples are used -
OFFSET   .equ     02000h          ; to get signed ADC values
```

```
X         .equ    0200h               ; x(n-k) sample address
;
; With the Timer_A interrupt the calculation is made
;
TIMA_INT PUSH     R5                  ; Save R5 and R6
         PUSH     R6
         MOV      #X,R5               ; Address xn buffer (oldest x)
         MOV      #An,R6              ; Address an constants (ak)
         MOV      &ADAT,2*k(R5)       ; New ADC sample to xn
         SUB      #OFFSET, 2*k(R5)    ; Create signed value for xn
         CLR      R7                  ; Clear result reg. (MSBs)
         CLR      R8
         CLR      R9
TA00     MACS16   @R5+,@R6+           ; ak * xn-k added to R7|R8|R9
         MOV      @R5,-2(R5)          ; xn-k+1 -> xn-k
         CMP      #X+2+(2*k),R5       ; Through? (R5 points outside)
         JNE      TA00                ; No, once more
         POP      R6                  ; Restore R5 and R6
         POP      R5
         BIS      #CS,&ACTL           ; Start next ADC conversion
         RETI                         ; Result: +-17.30 (3 words)
;
; The constants An are fixed in ROM. Format: +-0.15
; (1 bit sign, 15 bits fraction)
; Range: -0.99996 to +0.99996
;
An       .word    +9999*8000h/10000 ; ak     +0.9999
         .word    -9999*8000h/10000 ; ak-1   -0.9999
         ...                         ; ak-2 to a2
         .word    +5000*8000h/10000 ; a1     +0.5
         .word    -5000*8000h/10000 ; a0     -0.5
```

6.4.4.6 Fast Fourier Transform Algorithm

The buffer — located in the RAM — the pointer that pQR points to, is transformed and overwritten with the result of the fast Fourier transformation (FFT). The formula used for each block consists of real and imaginary numbers:

PRi'	=	(PRi + (QRi $\times$ WRi + QIi $\times$ WIi)/2	real part of Pi
PIi'	=	(PIi + (QIi $\times$ WRi − QRi $\times$ WIi)/2	imaginary part of Pi
QRi'	=	(PRi − (QRi $\times$ WRi + QIi $\times$ WIi)/2	real part of Qi
QIi'	=	(PIi − (QIi $\times$ WRi − QRi $\times$ WIi)/2	imaginary part of Qi

Where:

WRi	$\cos (i \times 2\pi /N) = \cos (\omega \times i)$
WIi	$\sin (i \times 2\pi /N) = \sin (\omega \times i)$
ω	$2\pi f$
i	Index number
PRi	Real part of PRi before FFT
PRi'	Real part of PRi after FFT

Figure 6–61 shows the allocation of the three tables in the RAM and ROM of the MSP430.

Execution time: the buffer shown, with eight complex numbers each for the P and Q part, needs 717 cycles (without CALL) for the transformation (185 µs @ 4 MHz).
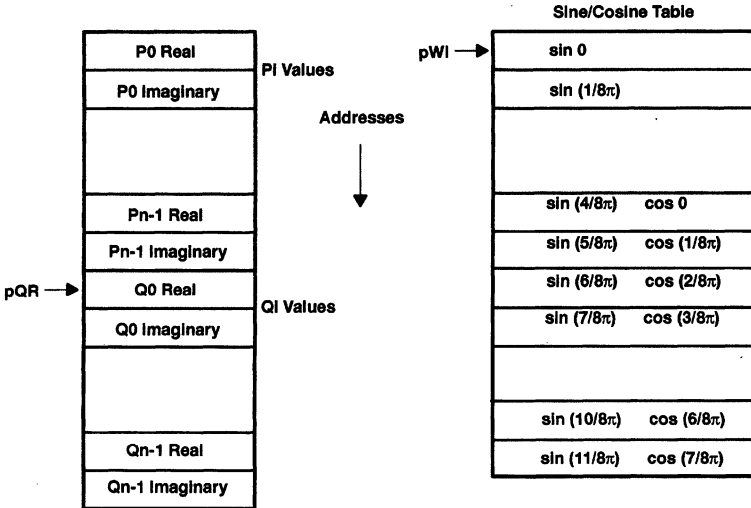
Figure 6–61. RAM and ROM Allocation for the Fast Fourier Transformation Algorithm

```
; Algorithm: 'FFT' optimized butterfly radix 2 for MSP430x33x
;
; Originally developed by M.Christ/TID for TMS320C80
;
; Input data: PR0,PI0,PR1,PI1,......,QRn-1,QIn-1 (16 bit words)
;
;    Algorithm:
;
;    PR' = (PR+(QR*WR+QI*WI))/2       WR=cos(wt)
;    PI' = (PI+(QI*WR-QR*WI))/2       WI=sin(wt)
;
;    QR' = (PR-(QR*WR+QI*WI))/2
;    QI' = (PI-(QI*WR-QR*WI))/2
;
;    Procedure:
;
;    real = (QR*WR+QI*WI)/2
;    imag = (QI*WR-QR*WI)/2
```

```
;
;    PR'  =  PR/2 + real
;    QR'  =  PR/2 - real
;
;    PI'  =  PI/2 + imag
;    QI'  =  PI/2 - imag


N         .equ    16              ; 16 point complex FFT
N2        .equ    N*2             ; Byte count (QR - PR)
pQR       .equ    R5              ; Pointer to QRi
pWI       .equ    R6              ; Pointer to sine table tabsin
real      .equ    R7              ; Storage  QR x WR + QI x WI
imag      .equ    R8              ; Storage  QI x WR + QR x WI
TEMP      .equ    R9              ; Temporary storage
TEMP1     .equ    R10             ;
;
; The subroutine FFT is called after the loading of the
; pointer to QR0.
;
; Call: MOV     #QR,pQR         ; Pointer to QR0 of block (RAM)
;       CALL    #FFT            ; Call the FFT subroutine
;       ...                     ; Input table contains results
;
; Definition of the input table located in the RAM
;
          .bss    PR,2,0200h      ; PR0    Preal
          .bss    PI,2            ; PI0    Pimaginary
          .bss    PRi,N2-4        ; PR1, PI1...PRn-1, PIn-1
          .bss    QR,2            ; QR0    Qreal
          .bss    QI,2            ; QI0    Qimaginary
          .bss    QRi,N2-4        ; QR1, QI1 ...QRn-1, QIn-1
;
; Start of the FFT subroutine. pQR contains address of QR0
;
FFT       MOV     #tabsin,pWI     ; Pointer to sin 0
;
```

```
; Execution of the 4 multiplications. The halfed result is
; calculated without additional shifts due to the format 2.14
; Calculation of the real part: real = (QR x WR + QI x WI)/2
;
FFTLOP    MPYS16    @pQR+,tabcos-tabsin(pWI);
          MOV       &SumHi,real       ; Store (QR x WR)/2  (2.14)
          MPYS16    @pQR,@pWI         ; (QI x WI)/2         (2.14)
          ADD       &SumHi,real       ; Store real part
;
; Calculation of the imaginary part:
; imag = (QI x WR - QR x WI)/2
;
          MPYS16    @pQR+,tabcos-tabsin(pWI);
          MOV       &SumHi,imag       ; Store (QI x WR)/2  (2.14)
          MPYS16    -4(pQR),@pWI+     ; (QR x WI)/2        (2.14)
          SUB       &SumHi,imag       ; Store imaginary part
;
; Calculation of PR', PI', QR', QI'. pQR points to QRi+1
; Calculation of PR': PR' = (PR + (QR x WR + QI x WI))/2
;
          MOV       -N2-4(pQR),TEMP ; PRi to TEMP
          RRA       TEMP              ; PRi/2
          MOV       TEMP,TEMP1        ; Copy PRi/2
          ADD       real,TEMP1        ; PRi/2 + (QRxWR + QIxWI)/2
          MOV       TEMP1,-N2-4(pQR) ; to PR'  (1.15)
;
; Calculation of QR': QR' = (PR - (QR x WR + QI x WI))/2
;
          SUB       real,TEMP         ; PR/2 - (QRxWR + QIxWI)/2
          MOV       TEMP,-4(pQR)      ; to QR' (1.15)
;
; Calculation of PI': PI' = (PI + (QI x WR - QR x WI))/2
;
          MOV       -N2-2(pQR),TEMP   ; PI
          RRA       TEMP              ; PI/2
          MOV       TEMP,TEMP1        ; Copy PI/2
```

```
        ADD     imag,TEMP1      ; PI/2 + (QIxWR - QRxWI)/2
        MOV     TEMP1,-N2-2(pQR) ; to PI' (1.15)
;
; Calculation of QI': QI' = (PI - (QI x WR - QR x WI))/2
;
        SUB     imag,TEMP       ; PI/2 - (QI*WR+QR*WI)/2
        MOV     TEMP,-2(pQR)    ; to QI' (1.15)
;
; To next input data. Check if FFT is finished
;
        CMP     #tabsin0,pWI    ; Through? (pWI = tabsin0)
        JLO     FFTLOP          ; No
        RET                     ; Yes, return
;
; Sine and cosine table. Format: s.fraction (1.15)
;
tabsin  .word   +0000*8000h/10000 ; sin 0.0 =        0.00000
        .word   +3827*8000h/10000 ; sin π/8 =        0.38268
        .word   +7071*8000h/10000 ; sin 2π/8 =       0.70711
        .word   +9239*8000h/10000 ; sin 3π/8 =       0.92388
tabcos  .word   10000*8000h/10000-1 ; sin 4π/8      cos 0.0
        .word   +9239*8000h/10000 ; sin 5π/8       cos π/8
        .word   +7071*8000h/10000 ; sin 6π/8       cos 2π/8
        .word   +3827*8000h/10000 ; sin 7π/8       cos 3π/8
tabsin0 .word   +0000*8000h/10000 ;       cos 4π/8
        .word   -3827*8000h/10000 ;       cos 5π/8
        .word   -7071*8000h/10000 ;       cos 6π/8
        .word   -9239*8000h/10000 ;       cos 7π/8
;
; An example is given for the FFT:
; The following table contains 32 values that are the data
; for the FFT
; 16 point complex FFT radix 2 DIT
;
DataSt  .word   014abh,02e90h,0f6d4h,005d3h ; PR0,PI0..PI1
        .word   004b2h,0fecdh,0f78ch,0fcb2h ; PR2,PI2..PI3
```

```
            .word    0093ch,004f0h,0ffb5h,0017ch  ; PR4,PI4..PI5
            .word    0fbebh,002a5h,0f3a3h,0fb38h  ; PR6,PI6..PI7
            .word    01854h,02a29h,0ffb9h,0f9beh  ; QR0,QI0..QI1
            .word    0fa49h,00907h,00a10h,0f99bh  ; QR2,QI2..QI3
            .word    0030ch,0fdadh,0fa2ah,002e3h  ; QR4,QI4..QI5
            .word    0fddbh,0029bh,0fdf9h,00225h  ; QR6,QI6. QI7
;
; The following 32 values are output by the FFT
;
Result      .word    0167fh,02c5ch,0fa16h,00013h  ; PR'0..PI'1
            .word    00384h,0049ch,0fabeh,0f879h  ; PR'2..PI'3
            .word    00374h,000f2h,0024dh,002e2h  ; PR'4..PI'5
            .word    0ffa4h,00128h,0fb2ah,0fd01h  ; PR'6..PI'7
            .word    0fe2bh,00233h,0fcbdh,005bfh  ; QR'0..QI'1
            .word    0012dh,0fa30h,0fccdh,00438h  ; QR'2..QI'3
            .word    005c7h,003fdh,0fd67h,0fe99h  ; QR'4..QI'5
            .word    0fc47h,0017ch,0f879h,0fe36h  ; QR'6..QI'7
```

6.4.4.7 Conclusion

As shown by the examples, the hardware multiplier has its biggest advantages when used for signed and unsigned 16-bit operands. But also for other applications — like for 8-bit operands, 32-bit operands or floating point numbers — the speed increase is valuable compared to the pure software solution.

6.5　The System Clock Generator

The system clock generator of the MSP430 family provides many features not available with other microcomputers. To allow the full use of all the possibilities, some basics concerning the function of the oscillator are needed. A detailed description of the hardware is given in the *MSP430 Family Architecture User's Guide and Module Library*, chapter *Oscillator and System Clock Generator*.

The output frequency, MCLK, of the system clock generator is generated in a digitally controlled oscillator (DCO), having 32 *taps*. Each one of these taps represents a typical output frequency ranging from 500 kHz to 4 MHz. These tap frequencies depend on temperature and supply voltage, and referencing to a crystal is therefore necessary.

```
; Software definitions for the programming examples

;

SCG1      .equ    080h             ; System Clock Generator Control Bit 1

SCG0      .equ    040h             ; System Clock Generator Control Bit 0

OSCoff    .equ    020h             ; If 1: Oscillator off

CPUoff    .equ    010h             ; If 1: CPU off

GIE       .equ    008h             ; General Interrupt Enable Bit

SCFI0     .equ    050h             ; System Clock Frequency Integrator Reg.

FN_2      .equ    004h             ; DCO current switch for 2 x fnom

SCFI1     .equ    051h             ; DCO tap register 2^9 to 2^2

TAP       .equ    008h             ; 2^5 bit in SCFI1

SCFQCTL   .equ    052h             ; System Clock Frequency Control Register

M         .equ    080h             ; Modulation Bit in SCFQCTL. M = 1: off
```

6.5.1　Initialization

After the application of the supply voltage, V_{CC}, the system clock frequency f_{system} is initialized to 1.024 MHz, if a 32.768 kHz crystal is used. This is automatically made by setting of the multiplication factor, N, to 32 and clearing of the FN_x bits in the control bytes SCFI0 and SCFI1. If the CPU is always on afterward and 1.024 MHz is the desired frequency, then there is nothing else to do.

6.5.1.1　First Setting of the DCO Taps during Initialization

The digitally controlled oscillator of the MSP430 starts at tap 0, which means at the lowest possible frequency ($\approx$ 500 kHz). To get from one tap to the next

one, 2^{10} (1024) cycles are needed. Thirty-two taps are implemented, so 32 × 1024 cycles are needed, worst case, to get to the correct DCO tap. The initialization routine should therefore have a length of 32000 cycles. If this is not the case, a delay routine should be added to guarantee this length. An example is given below:

```
INIT    ...                     ; Loop Control is on (SCG1 = SCG0 = 0)
        MOV     #11000,R5       ; Init delay to allow DCO setting
L$1     DEC     R5              ; 11000 x 3 cycles = 33000 cycles
        JNZ     L$1             ;
        BR      #MAINLOOP       ; Start program
```

6.5.2 Entering of Low Power Mode 3

The low power mode 3 (LPM3 —crystal on, DCO and loop control off) is the normal mode for battery-powered systems. Enabled interrupts (e.g. the basic timer) wake up the CPU. LPM3 is entered with the following source code:

```
        BIS     #CPUoff+GIE+SCG1+SCG0,SR    ; Enter LPM3
```

6.5.3 Wake-Up From Interrupts In Low Power Mode 3

Wake-up from LPM3 clears only bit SCG1 (LPM1). Due to the set bit SCG0, the loop control of the DCO is off. Normal interrupt routines are too short to allow the loop control to adjust the DCO tap — 1024 cycles are necessary to get from one tap to the other one. It is not necessary, therefore, to switch on the loop control. The CPU uses the DCO tap set during the last adaptation. A normal, short interrupt routine looks like this:

```
BT_HAND INC     COUNTER         ; LPM1: Loop Control stays off:
        RETI                    ; DCO is on for 17 cycles only
```

If woken–up from LPM3, the interrupt latency time (6 cycles) is increased by typ. 2 μs @ 1 MHz versus 1 μs @ 2 MHz (if FN_2 = 1). This means 8 cycles are typically needed from the interrupt event to the start of the interrupt handler. The time the DCO needs to settle to the nominal frequency is typically 4 cycles. This means interrupt handlers are processed with the correct frequency.

6.5.4 Adaptation of the DCO Tap During Calculations

The DCO tap of the system clock generator should be updated during longer on times of the CPU (e.g. during calculations). This is necessary especially if

accurate timing of the instructions is needed. During all calculations that exceed 100 cycles in length, the loop control of the DCO should be switched on. The way to do this is to reset the SCG0 bit in the status register after the wake-up:

```
; Calculations are necessary. Allow adaptation of the DCO tap
;
        BIC     #SCG0,SR        ; Switch on DCO loop control
        ...                     ; Calculate energy (>100 cycles)
        RETI                    ; Return to LPM3 with adapted DCO tap
```

The RETI instruction restores the CPU mode from the stack as it was when the interrupt occurred.

6.5.5 Wake–Up From Interrupts in Low Power Mode 4

The low power mode 4 normally lasts much longer than the low power mode 3 — it may last for months until a stored module is woken–up for calibration. This means that the environment temperature may have changed seriously. If the LPM4 was entered at a high temperature, the used DCO tap will be a relatively high one due to the negative temperature coefficient of the DCO. If then the device is woken–up at a low temperature and the crystal turns on fast, this high DCO tap may lead to a very high DCO frequency, a frequency the system cannot operate with. Therefore, it is a good programming practice, to program a low DCO tap before entering LPM4:

```
; Enter Low Power Mode 4: Set DCO tap to 2
;
        MOV.B   #TAP*2,&SCFI1     ; Set DCO tap to 2
        BIS     #CPUoff+OSCoff+GIE+SCG1+SCG0,SR ; Enter LPM4
```

If woken–up from LPM4, it may last up to seconds until the crystal has reached its nominal frequency. The frequency integrator counts down continuously as long as the crystal oscillator has not started its operation. This lasts until the lowest DCO tap (with the lowest system frequency) is reached. After the start of the crystal oscillator, the loop control will set the system frequency to its correct value by stepping up the taps.

6.5.6 Change of the System Frequency

The system clock frequency f_{system} depends on two values:

$$f_{system} = N \times f_{crystal}$$

Where:

N   Multiplication factor of the DCO loop (SCFQCTL contains N−1)

$f_{crystal}$ Frequency of the crystal (normally 32.768 kHz)

The normal way to change the system clock frequency is to change the multiplication factor N. The system clock frequency control register SCFQCTL is loaded with (N−1) to get the new frequency. To allow the DCO to work always in one of the center taps (13 to 18), three switches FN_2 to FN_4 are implemented in the register SCFI0. It gives a safety not to be at the frequency limits of the DCO. These switches increase the internal current of the DCO and allow higher output frequencies if set. The switch nearest to the programmed DCO output frequency should be used.

The switches FN_x typically settle within ±1 tap if the change is from the nominal frequency of one switch to the nominal frequency of the other one. For example, if in the example below, the initial system frequency is 1 MHz, then the new tap is one of the neighboring taps. This means, to settle at 2 MHz, needs a maximum of 1024 cycles (0.5 ms) only. If FN_2 is not used, it would take up to 16 x 1024 cycles (8 ms) because the misalignment could be up to 16 taps.

Note:

The switches FN_2, FN_3, and FN_4 need to be set correctly in dependence of the system clock frequency, f_{system}. Otherwise, erroneous behavior of the system will result. Only one switch may be in use at the same time — the one that is nearest to the actual frequency should be used. FN_2 controls frequencies near 2 MHz, FN_3 controls frequencies around 3 MHz, and FN_4 controls frequencies around 4 MHz.

```
; Change system frequency to 2.048MHz (fcrystal = 32.768kHz)

; N = 64  : Multiply 32kHz by 64 to get 2.048MHz

; FN_2 = 1: Adjust DCO current to 2MHz output frequency

; M = 0   : Switch on modulation

;

        MOV.B    #64-1,&SCFQCTL    ; 64 x 32kHz = 2.048MHz, M = 0

        MOV.B    #FN_2,&SCFI0      ; Adjust DCO current to 2MHz
```

6.5.7 The Modulation Bit M

The modulation bit M (SCFQCTL.7) switches off and on the influence of the 5 LSBs (N_{DCOmod}) of the system clock frequency integrator:

❑ **M = 0** — the modulation is on. This means all 10 bits of the integrator influence the DCO. The used tap of the DCO may be changed with every clock cycle to get the correct system clock frequency. This is the case if the programmed frequency lies exactly between two tap frequencies.

❑ **M = 1** — the modulation is off. This means only the 5 MSBs (N_{DCO}) of the integrator influence the DCO. The used tap of the DCO is changed only after 1024 clock cycles (for f_{system} = 1 MHz) to get the correct system clock frequency. If the programmed frequency lies exactly between two tap frequencies, then 1024 cycles are output with the lower tap frequency and 1024 cycles are output with the upper tap frequency.

In any case, independent of the modulation status, the integral error of the DCO will be zero.

The modulation may be switched off if a series of MCLK cycles is needed with exactly the same length (for measurements with the universal timer/port module, for example). To get this, the loop control also should be switched off.

```
; Ensure stable, non regulated output pulses with equal length:

;

        BIS.B    #SCG0,SR        ; Switch off loop control

        BIS.B    #M,&SCFQCTL     ; Switch off modulation

        ...                      ; Use non-regulated MCLK

 ;

; Return to a regulated MCLK with closed loop and modulation

;

        BIC.B    #SCG0,SR        ; Switch on loop control

        BIC.B    #M,&SCFQCTL     ; Switch on modulation
```

6.5.8 Use Without Crystal

If for an application, no precise timing is necessary, then the crystal may be omitted. If no ACLK is present (due to the missing crystal), then the DCO will run with its lowest frequency, which is approximately 500 kHz. No special instructions are necessary to get this behavior.

If this lowest DCO frequency is too low, then a higher DCO tap (eg. 10) may be used. This tap normally results in an MCLK frequency near 1MHz. It is important to switch off the FLL loop, otherwise the FLL control will step down to tap 0 slowly. The software for this use of the DCO follows:

```
; Initialization of the DCO for non-crystal mode:
; Loop control off, tap number = 10: MCLK ≈ 1MHz
;
        BIS.B    #SCG0,SR        ; Switch off loop control
        CLR.B    &SCFI0          ; Reset FN_x bits
        MOV.B    #050h,&SCFI1    ; Set bits for tap number 10
```

If an external reference like the ac line is available, then the actual MCLK frequency can be controlled simply by the counting of the MCLK output with one of the timers (e.g. for one ac line period). An example is given in section *The Timer_A* where the control of an LCD is also shown without a crystal and missing LCD control frequency due to the missing ACLK.

6.5.9 High System Frequencies Together With the 14-bit ADC

The maximum MCLK without input division is 1.5 MHz (132 cycles are needed for a conversion). To allow the full range of the system clock MCLK, together with the active ADC, a clock divider is included in the ADC module. It allows the division of the system frequency MCLK by factors of 1, 2, 3, and 4. See section *Analog-to-Digital Converters* for examples.

6.5.10 Dependencies of the System Clock Generator

If the DCO runs with an open loop, its frequency depends on the temperature and the supply voltage, V_{CC}. Nominal values for these dependencies are:

❑ Temperature dependence: -5.6 kHz/($°C \times$ MHz)

❑ Voltage dependence: $+60$ kHz/(V $\times$ MHz)

These two dependencies are brought to zero if the DCO loop is closed (SR–bits: SCG0 = SCG1 = OscOff = 0). See the next section for short term deviations of the system clock generator (MCLK).

6.5.11 Short Time Accuracy of the System Clock Generator

The error of the system clock generator is zero for long time periods (compared to the system frequency f_{system}). Normally, no tap of the DCO can deliver the correct system frequency, f_{system}, which is defined for the settled state to

$$f_{system} = N \times f_{crystal}$$

Therefore, the system clock generator switches continuously between two adjacent DCO taps — the one with a lower frequency f_N and the tap with a higher frequency f_{N+1}. This switching between the two DCO taps N_{DCO} and $N_{DCO}+1$ is interlaced in such a way that it results in a small error at any time within the ACLK period. The resulting error for a complete ACLK period is nearly zero and the integral error for a longer period is zero.

Figure 6–62 shows the use of the 10 bits of the registers SCFI0 and SCFI1. The five MSBs N_{DCO} control the DCO–taps, the five LSBs N_{DCOmod} control the modulation scheme of the DCO.

Bits located in the System Clock Frequency Integrator Registers SCFI0 and SCFI1
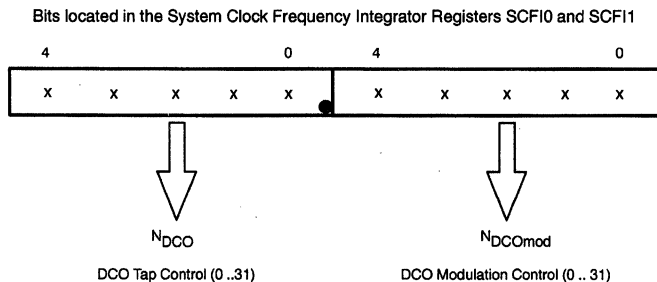
Figure 6–62. Control of the DCO by the System Clock Frequency Integrator

Figure 6–63 illustrates the DCO switching between the lower and the higher DCO tap for selected values of N_{DCOmod}.
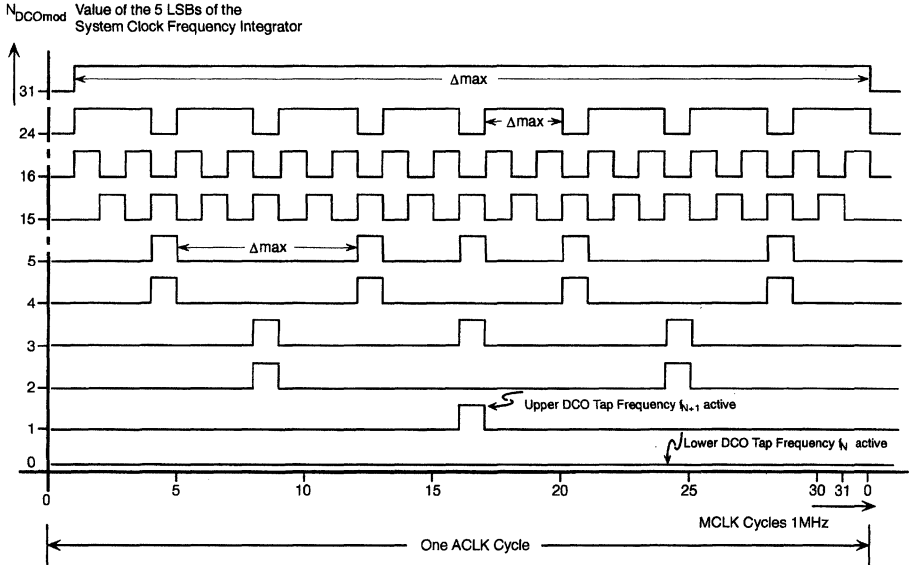
Figure 6–63. Switching of The DCO Taps Dependent on N_{DCOmod}

Table 6–32 lists the errors of the system clock generator. The assumptions for Table 6–32 are:

☐ The frequency step from the lower tap frequency f_N to the higher tap frequency f_{N+1} is 10% (multiplication factor 1.1).

☐ The two frequencies, f_N and f_{N+1}, allow an error free system frequency during one ACLK period — the two frequencies result in zero error if used with the shown N_{DCOmod}.

☐ The crystal error is not included — the crystal is considered to be errorfree.

☐ The system frequency is normalized to 1 MHz. For all other frequencies, the resulting errors can be calculated simply by a relation to 1 MHz.

☐ The FLL has settled, which means, it has had enough time (e.g. during calculation sequences) to switch to the appropriate DCO taps.

Three errors of the system clock generator are calculated in Table 6–32:

❏ The maximum time deviation, terr, during an ACLK period for a system with ideal tap frequencies (assumption 2 above). This inherent time deviation is mainly due to the length of Δmax.

❏ The worst case time deviation, $terr_{max}$, during an ACLK period for a system with ideal tap frequencies. To get this time deviation, the calculation is made with the DCO frequencies for a value of N_{DCOmod} that is one step above the correct frequencies. This results in the maximum possible time deviation for the FLL, independent of the tap frequencies.

❏ The worst case value of the integrated time deviation $terr_{per}$. This is the largest deviation seen at the end of a complete ACLK period.

Note:

The three errors named above do not accumulate — on the contrary, they get smaller with each ACLK period and tend to reach zero. This is a very important property of the system clock generator.

Values in brackets are used for calculations only, shading indicates the frequency used (f_N resp. f_{N+1}) for the error calculation.

Table 6–32. System Clock Generator Error

N_{DCO}mod	f_N (MHz)	f_{N+1}(MHz)	Δmax	terr (ns)	$terr_{max}$ (ns)	$terr_{per}$ (ns)
0	1.000	1.1000	32	0	89	91
1	0.9972	1.0969	31	87	177	91
2	0.9943	1.0938	15	86	129	91
3	0.9915	1.0906	15	128	172	93
4	0.9886	1.0875	7	80	101	92
5	0.9858	1.0844	7	101	121	92
(6)	0.9830	1.0813	7	121	–	–
15	0.9574	1.0531	3	134	143	96
16	0.9545	1.0500	1	48	51	0
(17)	0.9517	1.0469	1	51	–	–
24	0.9318	1.0250	3	–73	–64	98
(25)	0.9290	1.0219	4	–86	–	–
31	0.9119	1.0031	31	–97	–100	100
(0)	0.9091	1.0000	32	0	–	–

Note:

The values shown in Table 6–32 get smaller with increasing frequency. If f_N is 2 MHz, for example, the values are only one-half of the table values.

Where:

N_{DCOmod}	Value of the five LSBs of the system clock frequency integrator
f_N	DCO output frequency of the DCO tap N_{DCO} (lower frequency) [MHz]
f_{N+1}	DCO output frequency of the DCO tap $N_{DCO}+1$ (higher frequency) [MHz]
Δmax	Longest sequence with the same tap within the switching scheme for a given value of N_{DCOmod} measured in MCLK cycles. See figure 6–63
terr	Maximum time deviation (time error) within an ACLK period due to Δmax. terr is the inherent error for a given value of N_{DCOmod}. [ns]
$terr_{max}$	Worst case time deviation (time error) within an ACLK period due to Δmax. The higher error results from the correction with the tap frequencies for (N_{DCOmod} +1). For a full ACLK period the error reduces to $terr_{per}$ [ns]
$terr_{per}$	Worst case time error for a complete ACLK period (30.5 µs). The error results from the correction with the tap frequencies for ($N_{DCOmod}+1$). [ns]
f_{system}	Nominal, errorless value of the system frequency. Here 1MHz. [MHz]

The formulas used for the error calculations are:

$$terr \; = \; \Delta max \times \left(\frac{1}{f_N} - \frac{1}{f_{system}} \right) \qquad \qquad terr \; = \; \Delta max \times \left(\frac{1}{f_{N+1}} - \frac{1}{f_{system}} \right)$$

or for shaded cells

The formulas for $terr_{max}$ are the same as for terr, but tap frequencies f_N and f_{N+1} of ($N_{DCOmod}+1$) are used.

The formula for $terr_{per}$ also uses the tap frequencies f_N and f_{N+1} of ($N_{DCOmod}+1$).

$$terr_{per} \; = \; (32 - N_{DCOmod}) \times \left(\frac{1}{f_N} - \frac{1}{f_{system}} \right) + N_{DCOmod} \times \left(\frac{1}{f_{N+1}} - \frac{1}{f_{system}} \right)$$

6.5.12 The Oscillator Fault Interrupt Flag

If the value N_{DCO} contained in the DCO tap control byte SCFI1 moves out of its valid range:

$$0 < N_{DCO} < 28$$

the oscillator fault interrupt flag OFIFG (located in IFG1.1) is set. If the oscillator interrupt enable bit OFIE (located in IE1.1) is also set, an interrupt is requested.

Note:

The interrupt vector of the oscillator fault interrupt flag is shared with the non maskable interrupt (NMI). It is necessary, therefore, to test the oscillator fault interrupt flag first to determine the cause of the interrupt.

The oscillator fault interrupt flag is set after the supply voltage is applied to the MSP43, due to the start of the DCO at the lowest frequency ($N_{DCO} = 0$). When the interrupt is granted, the oscillator interrupt enable bit OFIE is reset automatically to disable further interrupt requests. The oscillator fault interrupt flag OFIFG must be reset by software.

6.5.13 Conclusion

The time deviations listed in Table 6–32 demonstrate the small error introduced by the modulation of the DCO. The largest time deviation inside of one ACLK period is 177 ns. This is relatively small compared to the inherent digital uncertainty, which is one MCLK cycle (1 μs @ 1 MHz). The time deviations of the system clock generator do not accumulate, but get smaller with the next ACLK period. Therefore, the overall error tends to move toward zero for a longer time period. The system clock generator, with its output frequency f_{system} (MCLK), is therefore usable for precise time measurements like a normal crystal oscillator.

6.6 The RESET Function

The RESET functions of the MSP430 family are described in detail below. Many problems can be avoided if the RESET functions are completely understood. Normally, the internal RESET hardware, together with the watchdog timer, avoids these problems. Under certain circumstances, however, additional external hardware is necessary. Several methods are described.

6.6.1 Description of the MSP430 RESET Function

The MSP430 generates two different internal RESET signals:

❑ The power-on reset signal (POR).

❑ The power-up clear signal (PUC).

These two signals are not available externally — they are used only internally (on-chip). Figure 6–64 gives a simplified overview of the RESET function. The numbers at the gate inputs refer to the signals described in sections 6.6.1.1 and 6.6.1.2.
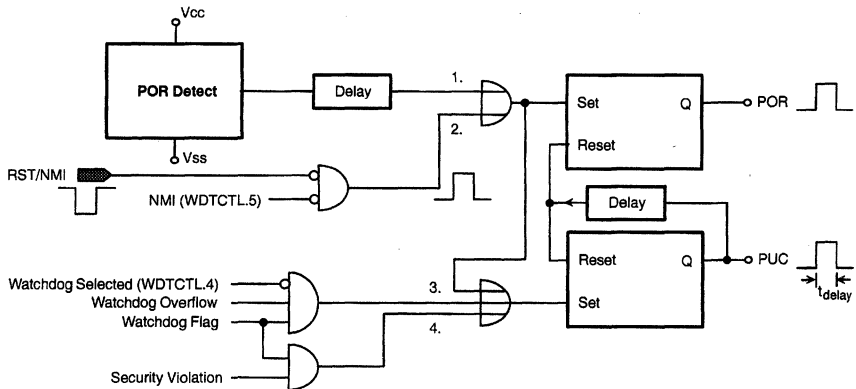
Figure 6–64. Simplified MSP430 RESET Circuitry

Note:

The power on detection circuit is not a supply voltage supervisor. Control of the supply voltage is normally performed by linear circuits. Stable supply voltage to the MSP430 must be maintained at all times, including when it is in low power mode 3 and 4.

6.6.1.1 The Power–On Reset Signal

The power-on reset signal, POR, is caused by two completely different external events:

❏ The power-on detection circuitry detects the rise of the supply voltage, V_{CC} (power-on signal).

❏ The RST/NMI terminal is reset to V_{SS} and set to V_{CC} afterward. This is the case only if the reset terminal is switched to the RESET function (default after power-on) and not to the NMI function. The RESET function is used if WDTCTL.5 = 0.

(NMI stands for non maskable interrupt, an external interrupt input that cannot be disabled by the interrupt enable bit (GIE) in the status register (SR). Each interrupt event will request an interrupt. The NMI function is used if WDTCTL.5 = 1).

6.6.1.2 The Power-Up Clear Signal

The power-up clear signal, PUC, can be caused by several events:

❏ The power-on detection circuitry detects the rise of the supply voltage (V_{CC}). This event also causes the POR signal.

❏ The RST/NMI terminal is reset to V_{SS} and then set to V_{CC} afterward (see above). This event also causes the POR signal.

❏ The expiring of the Watchdog Tmer if programmed to the watchdog mode (WDTCTL.4 = 0). The watchdog function is always active after PUC and POR.

❏ The use of an invalid password for the writing to the watchdog control word WDTCTL (security violation). This reset generation is independent of the watchdog function — it also occurs if the watchdog is used in timer mode (WDTCTL.4 = 1)

6.6.1.3 Common Operations for Power–Up and Power–On Reset

If one of the events described above occurs, the following operations are started:

❏ The digital I/O ports (Port0 to Port4) are set to the input direction.

❏ The I/O flags are set to 0 as discussed in the description of the peripherals.

❏ The address contained in the vector at address 0FFFEh is written into the Program Counter PC (software start address).

❑ The Status Register SR of the CPU is reset to 0. This means:

■ The CPU is set to the active mode.

■ The maskable interrupts are disabled by the reset GIE–bit (SR.4).

■ The loop control for the system clock generator is switched on (the FLL is active).

■ The system clock generator is set to an MCLK frequency of 1 MHz @ fACLK = 32.768 kHz.

❑ The digitally controlled oscillator (DCO) in the system clock generator (FLL for MCLK) is set to its lowest output frequency (DCO tap 0). The reason for this is to also include the possible malfunction of the system clock generator. Otherwise the erroneous FLL frequency is also active during the restoring phase of the system functionality with a fatal effect: the system cannot come up correctly.

❑ The RST/NMI terminal is configured to the RESET function.

❑ The Watchdog Timer is configured as a watchdog driven by the system clock MCLK.

❑ The CPU starts operation at the address written into the Program Counter (from address 0FFFEh) after the RST/NMI terminal is set to V_{CC} voltage.

6.6.1.4 Differences Between Power-Up Reset and Power-On Reset

The few differences between the two reset signals are detailed below.

6.6.1.4.1 Peculiar to The Power-On Reset Signal

❑ The power-on reset signal (POR) also generates the PUC signal.

❑ The power-on reset signal sets (1) or resets (0) the peripheral bits enclosed in round brackets (see *Architecture User's Guide*). These bits (all of the peripheral bits of the 16-Bit Timer_A, for example) are not influenced by the PUC signal. For example, rw–(0) means a readable and writable peripheral bit that is set to 0 by the POR signal only, but not by the PUC signal.

The reason is that some functions may not be modified by watchdog events. These functions are mostly controlled by software.

6.6.1.4.2 Peculiar to The Power-Up Clear Signal

❑ The power-up clear signal (PUC) does not also generate the POR signal.

❏ The power-up clear signal (PUC) sets or resets the peripheral bits not en-
closed in round brackets (see *Architecture User's Guide*). For example,
rw–0 means a readable and writable bit that is set to 0 by the POR and
PUC signals.

6.6.1.5 The RST/NMI Terminal Hardware

Some important facts that need to be considered when using the MSP430
RST/NMI terminal:

❏ The RST/NMI terminal does not have a pulldown or pullup resistor. The
user must ensure a stable DV_{CC} or V_{CC} voltage level at the RST/NMI ter-
minal during normal operation. Otherwise, hum or noise will generate arbi-
trary system resets if the terminal is left unconnected (floating).

❏ The RST/NMI terminal is an input pin only. No RESET signal is output if
an internal RESET occurs (by a watchdog overflow, for example). If exter-
nal devices must also be reset, then two possibilities exist (see Figure
6–65, right side):

■ An O-output is used. This output is set and reset by a short software
routine during the initialization phase following the RESET signal. The
state of an O-output is not defined after the power up.

■ An I/O terminal (one of Port0 to Port4) or a TP terminal is used. This
terminal is connected to V_{SS} (DV_{SS}) with a resistor ($\approx 10\,k\Omega$). While the
RESET signal is active, the pin is switched to the input direction
(compared to the HI–Z state) and the resistor pulls down the I/O termi-
nal. This low signal resets all external devices immediately. The sub-
sequent initialization software switches the terminal to an active high
signal. The external RESET signal is terminated. (A positive RESET
signal can be generated in the same way. The resistor is connected to
the supply voltage V_{CC}, the initialization software outputs an active
low signal.)

❏ The power-on detection circuitry is only able to detect a new, slow rise of
the supply voltage (V_{CC}) after a power fail, if V_{CC} falls below a defined volt-
age, V_{CCmin}. If this cannot be guaranteed, external hardware is recom-
mended. See the next section for the details of this hardware.

❏ To guarantee a successful RESET, the low signal at the RST/NMI terminal
needs a minimum length of time (t_{reset}). This minimum time is 2 µs.
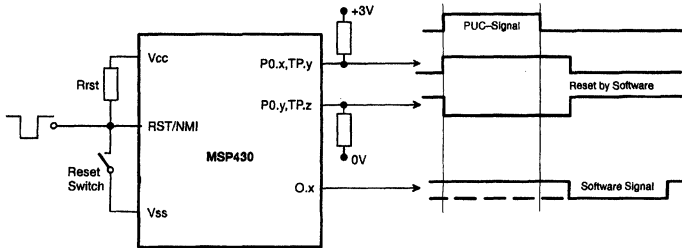
Figure 6–65. Generation of RESET–Signals for External Peripherals

6.6.2 RESET With the Internal Hardware, Including the Watchdog

The internal power-on detection hardware of the MSP430 allows very reliable initializations for the complete system. If, due to special circumstances, this normal process fails, the watchdog (which is completely different from that in most other microcomputer systems in that it is active after power on) resets the MSP430 once more.

6.6.2.1 Restrictions

The oscillator fault flag OFIFG is set as long as the system clock frequency MCLK is outside of the frequency limits. The flag information is set by the FLL hardware — if the DCO reaches its frequency limits then the flag OFIFG is set (IFG1.1 = 1). This flag must be reset by software.

Note:

The oscillator fault interrupt uses the same interrupt vector (0FFFCh) as the NMI interrupt. This means that the interrupt handler first has to check the cause of the interrupt if the NMI function is also used. This is possible by testing of the OFIFG flag (IFG1.1) or by testing the NMI flag. (IFG1.4).

The frequency limits of the digitally controlled oscillator are reached if the system clock frequency integrator register SCFI1 contains 0 or ≥ 0E0h (corresponds to DCO taps 0 or ≥ 28). See the *Architecture User's Guide*.

6.6.2.2 Start-Up of the Crystal

The ultra low power design used for the crystal oscillator of the MSP430 results in a relatively long time before it reaches oscillating stability. This may take up to four seconds. Until the crystal oscillator has reached a stable ACLK frequency (32 kHz), the digitally controlled oscillator (DCO) of the system clock generator remains at its lowest frequency (see appropriate data sheet). This is not a problem for most of the MSP430 applications. The crystal oscillator is switched on after power on, and runs continuously with no off periods.

The tap used is defined by the five most significant bits of the FLL register SCFI1 at address 051h. The actual value of register SCFI1 can be stored to provide quick return to the former system clock frequency (MCLK) in case of a RESET. The stored value needs to be checked, otherwise the value that caused an oscillator fault is restored again and again, which results in a system hangup.

6.6.3 Reliable RESET With Slowly Rising Power Supplies

To get reliable RESET conditions with power supplies that exhibit very slowly increasing voltages ($\Delta V/\Delta t$), or with voltage dropouts that do not reach the lower threshold voltage (V_{min}) of the POR detection circuitry (approximately 0.4 V, see data sheet), some external hardware is recommended. Some possibilities are shown in this chapter.

Note:

No call for emergency tasks is possible with all the solutions shown in this section. The RESET signal goes low without any warning to the software. If it is necessary to save important RAM contents in an external EEPROM and to execute defined emergency tasks before the RESET signal goes active, then the solutions shown in the section *Battery Check and Power Fail Detection* should be considered. Voltage supervision is performed at the regulator input and signals the loss of the supply voltage via the RST/NMI terminal switched to the NMI function. This early warning allows the execution of emergency tasks before power fails completely.

6.6.3.1 RESET With a Switch

This is the most simple RESET hardware for the MSP430. It is normally used if the battery is soldered into the system and the calibration constants reside in the RAM. But this solution may also be used for all other supply systems. If calibration constants are stored in the RAM, then a check at the start of the initialization routine is necessary if a *Warmstart* (system is calibrated) or a *Coldstart* (RAM is nonvalid) occurred. This distinction is normally made with

special constants written to the RAM during the calibration process. These constants use bit patterns that are relatively improbable (e.g. 05AF0h) due to their mix of 0s and 1s. See section *Software Applications* for more details.

To reset the MSP430, the switch (Figure 6–66) is pressed for a moment and then released. The V_{SS} potential at the RST/NMI terminal initiates the POR and the PUC signals.
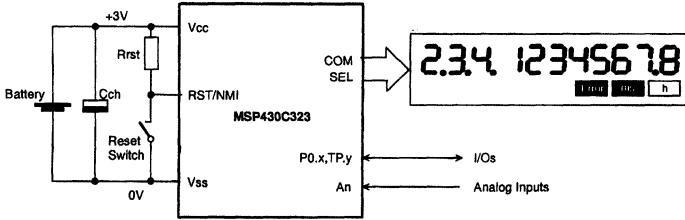
Figure 6–66. Battery-Powered System With RESET Switch

6.6.3.2 PNP Transistor With Zener Diode

This simple hardware (Figure 6–67) may be used if the supply voltage of the MSP430 is delivered from a higher system voltage, Vsys, of 6 V to 15 V. The PNP transistor, together with the 3.3-V or 4.5-V Zener diode, delivers the supply voltage and the RESET signal for the MSP430 system. The fast rise of the supply voltage V_{CC} provides a reliable power up.
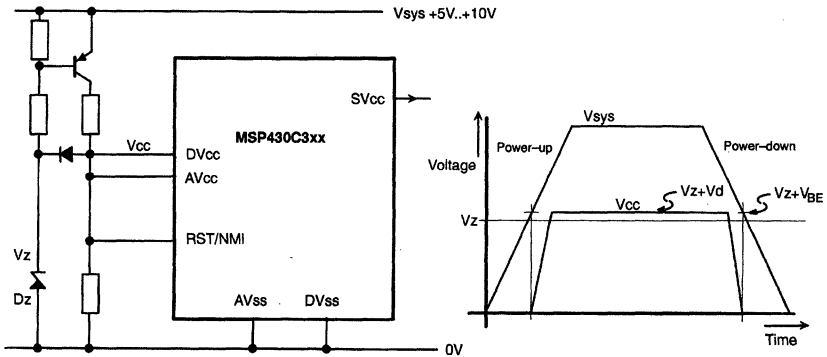
Figure 6–67. Simple RESET Circuit With a PNP Transistor

6.6.3.3 Operational Amplifier With Reference Diode

With an operational amplifier used as a comparator (an unused operational amplifier in a dual or quad package, for example) a simple and reliable RESET circuit can be built. Figure 6–68 illustrates this solution. During the start-up phase of the supply voltage, the voltage of the nonconducting reference diode is higher than the divided voltage at the noninverting input of the comparator. This causes the output voltage of the comparator to be low and the MSP430 is held in the reset state. When the supply voltage reaches the minimum voltage, V_{CCmin}, (defined by Vz, R2, and R3) then the comparator outputs a high signal and the MSP430 starts with its program. The value of V_{CCmin} is:

$$Vcc_{min} = Vref \times \left(\frac{R2}{R3} + 1\right)$$
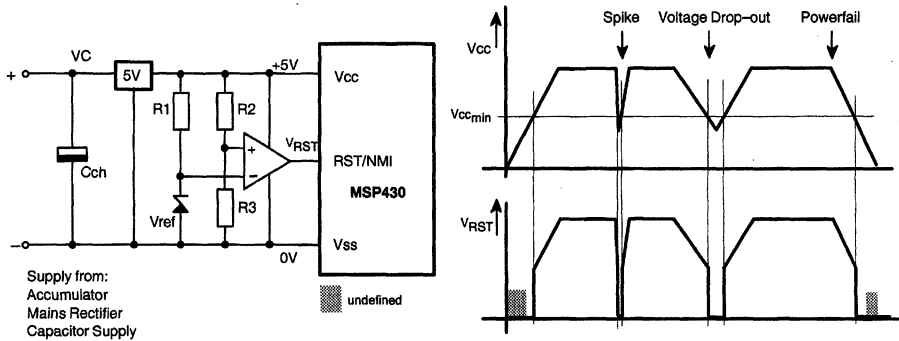
Figure 6–68. RESET Circuit With a Comparator and a Reference Diode

For reliable results, the operational amplifier used should be able to operate with relatively small supply voltages (approximately 1 V).

For the calculation of the resistor values for R2 and R3, see the formulas below for figure 6–69. Resistor R4 is simply set to infinite (∞) to get the values for this solution.

Circuitry similar to above is used in Figure 6–69. The only difference is the Schmitt trigger characteristic of the RESET circuitry — it rejects false RESET signals caused by hum, noise, and spikes. Two independent voltage threshold voltages, VTH+ and VTH–, can be calculated with the reference voltage Vref and the three resistors R2, R3, and R4. The formulas follow below.
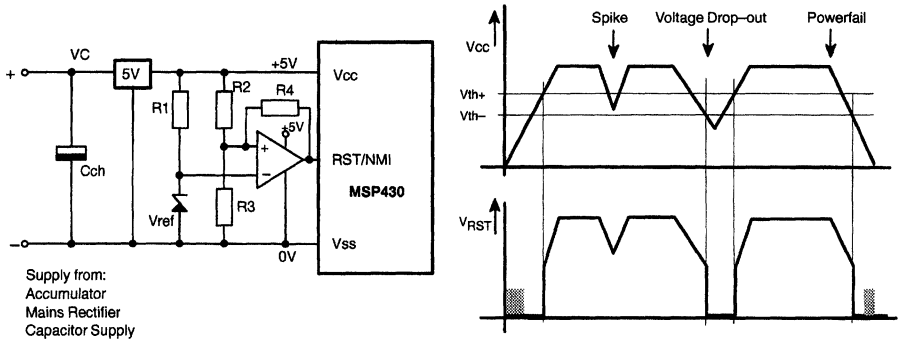
Figure 6–69. RESET Circuit With a Schmitt Trigger and a Reference Diode

All of the resistors (R2, R3, and R4) are relative to a calculated resistor Ri, which is the resulting resistance of the paralleled resistors R2 and R3. The value of Ri depends on the input offset current (Ioff) of the operational amplifier and the maximum tolerable error voltage (Ue) caused by Ioff. The maximum value of Ri is calculated first:

$$Ri \ < \ \frac{Ue}{Ioff}$$

With the calculated value of Ri, the three resistors (R2, R3, and R4) are calculated next:

$$R4 \ = \ Ri \times \left(\frac{V_{TH+} \times V_{TH-}}{Vref \times (V_{TH+} - V_{TH-})} - 1 \right)$$

$$R3 \ = \ Ri \times \left(\frac{1}{1 - \dfrac{Vref}{V_{TH+}} \times \left(\dfrac{Ri}{R4} + 1 \right)} \right)$$

$$R2 = Ri \times \left(\cfrac{1}{\cfrac{Vref}{V_{TH+}} \times \left(\cfrac{Ri}{R4} + 1\right)} \right)$$

Example 6–57. RESET Circuit

A RESET circuit similar to that shown in Figure 6–69 is built with the following behavior:

RST/NMI is V_{SS} for $V_{CC} < 2.5$ V

RST/NMI is V_{CC} for $V_{CC} > 3$ V

Vref = 1.25 V

$Ioff_{max} = \pm 200$ nA (maximum offset current of the inverting input)

Maximum voltage error due to Ioff: ± 150 mV

$$Ri < \frac{150mV}{200nA} \quad \rightarrow \quad Ri < 0.75M\Omega$$

$$R4 = 0.75M\Omega \times \left(\frac{3V \times 2.5V}{1.25V \times (3V - 2.5V)} - 1 \right) = 9M\Omega$$

$$R3 = 0.75M\Omega \times \left(\cfrac{1}{1 - \cfrac{1.25V}{3V} \times \left(\cfrac{0.75M\Omega}{9M\Omega} + 1\right)} \right) = 1.37M\Omega$$

$$R2 = 0.75M\Omega \times \left(\cfrac{1}{\cfrac{1.25V}{3V} \times \left(\cfrac{0.75M\Omega}{9M\Omega} + 1\right)} \right) = 1.67M\Omega$$

Figure 6–70 illustrates the connection of an operational amplifier with an output voltage higher than V_{CC} (here the unregulated voltage V_C) to the MSP430 RST/NMI terminal. The diode protects the RST/NMI terminal and resistor Rrst provides the positive voltage.
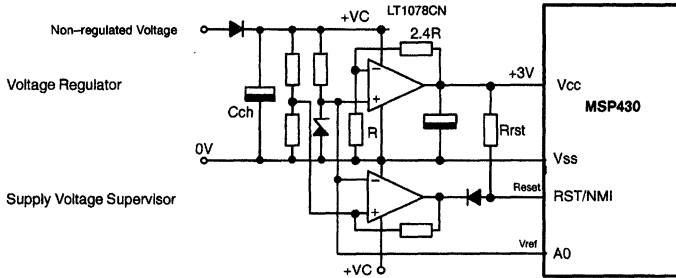
Figure 6–70. RESET Generation With a Comparator

6.6.3.4 Supply Voltage Supervisors

The use of a supply voltage supervisor is one of the best methods of getting a reliable RESET signal. All necessary parts such as reference, programmable delay, output stage, and so on are integrated in a single IC. Only a few external components are necessary. Two different solutions are explained:

❏ The TL770x supply voltage supervisor.

❏ The TP3750 supply voltage supervisor and regulator. This IC integrates two functions: supply voltage regulation, and supply voltage supervision.

6.6.3.4.1 TL7701 Supply Voltage Supervisor

The schematic for a supervised MSP430 is shown in Figure 6–71. The TLC7701 is programmed with the resistors R1 and R2 to reset the MSP430 when the output voltage of the 5-V regulator falls below Vcc_{min} (2.5 V).
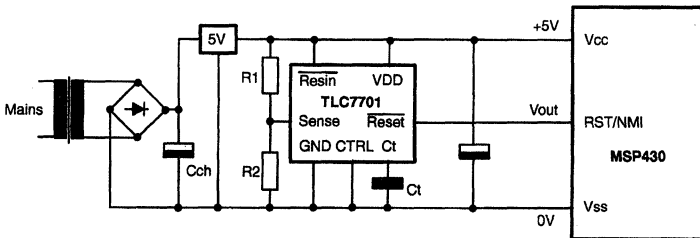
Figure 6–71. Power Fail Detection With a Supply Voltage Supervisor

Figure 6–72 shows the different system states of the voltage supervisor solution. The voltage V_{CC} is drawn in a simplified manner for a better understanding of the system function. The different *System States* (shown below in Figure 6–72) are:

❏ Up to a certain voltage, the output of the TLC7701 is undefined due to the too-low supply voltage. After this voltage is reached, the TLC7701 output is low until the voltage (Vcc_{min} — defined by R1 and R2) is reached. The RST/NMI input of the MSP430 is a reset input after the power up, so the MSP430 CPU is inactive.

❏ After the reaching of the voltage Vcc_{min} (and the expiration of the delay time, trc), the MSP430 begins operation.

❏ If the supply voltage (V_{CC}) drops below Vcc_{min} due to a voltage dropout, the RST/NMI input is pulled low, stopping the CPU.

❏ After the return of V_{CC} and the expiration of the delay time trc, the RST/NMI input is pulled high again and the CPU starts at the address contained in the reset vector (0FFFEh).

❏ If a real power fail occurs, the RST/NMI input is held low until the voltage region with undefined output is reached. This voltage is so low that CPU operation is not possible.
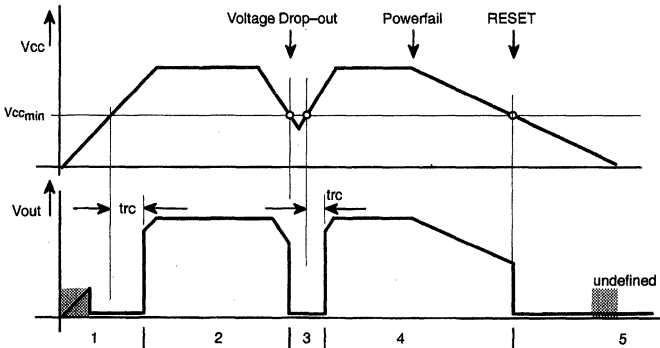
Figure 6–72. System Voltages With a Power Supply Supervisor

The threshold voltage Vcc_{min} of the TLC7701 is:

$$Vcc_{min} = Vref \times \left(\frac{R1}{R2} + 1\right)$$

Where:

V_{ref}	Voltage of the internal voltage reference of the TLC7701:	+1.1 V
R2	Resistor from SENSE input to GND. Nominal value:	100 kΩ to 200 kΩ

The delay trc after the return of V_C is defined by the capacitor Ct shown in Figure 6–71. If this delay is not desired, capacitor Ct is omitted. The formula for the delay time trc is:

$$trc = 21k\Omega \times Ct$$

6.6.3.4.2 TP3750 Supply Voltage Supervisor and Regulator

Figure 6–73 illustrates the use of a TPS7350 (regulator plus voltage supervisor), ensuring a highly reliable system initialization. The TPS7350 also allows the use of the RST/NMI terminal of the MSP430 as described in section *Battery Check and Power Fail Detection*. The RST/NMI terminal is used during normal program operation as an NMI (non maskable interrupt) input. This gives the possibility to save important data in an external EEPROM in case of power fail. This is possible because the PG terminal outputs a negative signal starting at V_{CC} = 4.75 V, which allows a large number of activities until Vcc$_{min}$ of the MSP430 (2.5 V) is reached.

Diode D, together with series resistor Rv and capacitor Cb allow the MSP430 system to bridge short voltage dropouts or disturbances of the supply voltage V_{sys}. The diode (D) prevents the rapid discharge of Cb by the other peripherals connected to V_{sys} and increases the possible active time for the MSP430 after loss of V_{sys}.
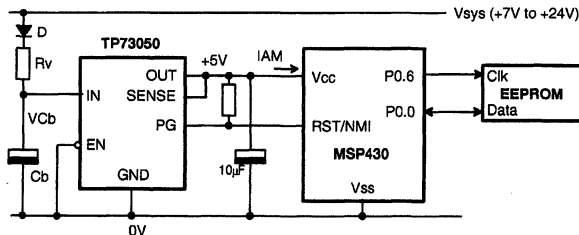
Figure 6–73. Power Supply From Other DC Voltages With a Voltage Regulator/Supervisor

6.6.4 Conclusion

It is an old truth that many difficulties are caused by the implementation of the RESET sections of projects. The hardware of the MSP430 family is designed to minimize these difficulties as much as possible without special considerations or external components. But when special circumstances exceed the builtin capabilities of the MSP430, the solutions shown in this section may help to significantly simplify the development phase of a project.

6.7 The Universal Timer/Port Module

The Universal Timer/Port module consists of two independent parts that work together for the measurement of resistors or voltages.

❏ **Counter With Controller** — two 8-bit counters, which may be connected in series to get a 16-bit counter. In addition, there is a controller, a comparator input CMPI, and a normal input CIN.

❏ **Input/Output Port** — five outputs (TP.0...TP.4) that can be switched to Hi–Z and an I/O port, TP.5

Three different inputs are available with the module:

❏ The CIN input have a Schmitt trigger characteristic. It is normally used for resistor measurements. The threshold voltages are the same as for the other inputs (P0.x).

❏ The comparator input CMPI — which is used for the voltage measurement — has a threshold voltage $V_{ref(com)}$ that is nominally $0.25 \times V_{cc}$ with small tolerances. The threshold voltage ($V_{ref(com)}$) itself, is temperature independent. The input CMPI shares a terminal with an LCD select line and must be switched by software to the input function. This input function is valid until the next PUC.

❏ The I/O terminal TP.5, which may be used as a clock input or an enable input.

Figure 6–74 shows the block diagram of the Universal Timer/Port module

2 x 8–Bit Counter or 1 x 16–BitCounter with Clock Frequency and Enable Control

I/O Port   Counter with Controller   Control Registers
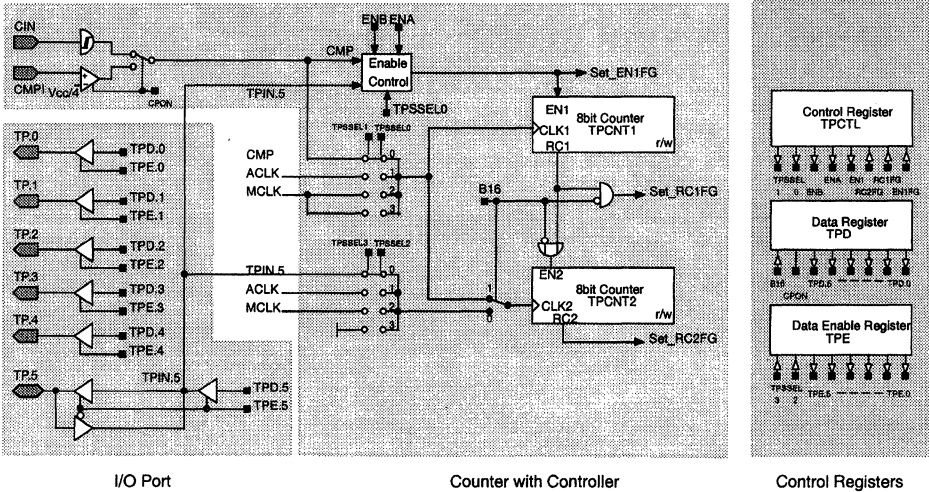
Figure 6–74. Block Diagram of the Universal Timer/Port Module

6.7.1 Universal Timer/Port Used as an Analog-to-Digital Converter

Applications of the Universal Timer/Port Module as an ADC are described in section *Analog-to-Digital Converters*. This section shows other applications, such as simple timers and similar functions.

6.7.2 Universal Timer/Port Used as a Timer

MSP430 family members that do not contain the Timer_A are equipped with at least the Universal Timer/Port module, a combination of two 8-bit timers with a common control unit. The Universal Timer/Port module is primarily regarded as an ADC, but it is also able to handle timing tasks that are not excessively complex. To get an interrupt request after a certain number of MCLK or ACLK cycles, it is only necessary to load the negated number of clocks into the count registers TPCNT1 and TPCNT2. When the 16-bit counter (used with the MCLK) or one of the 8-bit counters (used with the ACLK) overflows to 0, the RC2FG flag (or RC1FG flag) is set and an interrupt is requested. This method allows precise timings for TRIAC control or PWM control in the range of 128 Hz to 4000 Hz (repetition rate).

The Universal Timer/Port module can be used for:

☐ Low frequency pulse width modulation: up to two independent PWM–outputs.

☐ Measurement of the MCLK frequency e.g. if used without crystal (see section *Use Without Crystal*)

☐ Triggering: time measurement starting with the zero crossing of the mains voltage

☐ Other time measurements
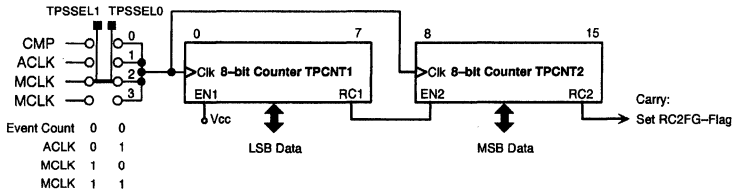
Figure 6–75. Block Diagram of the Universal Timer/Port Module (16-Bit Timer Mode)

6.7.2.1 Continuous Mode

The Universal Timer/Port module can be used like the Timer_A in continuous mode, allowing it to measure time differences. The 16-bit value is read out and corrected if an overflow to 0 occurred between the readings of the low and high bytes. The input frequency may be the ACLK or the MCLK.

```
; Read-out of the UTP/M running as a 16-bit timer
;
        MOV.B   &TPCNT1,R5      ; Read LSBs  00xx
        MOV.B   &TPCNT2,R6      ; Read MSBs  00yy
        CMP.B   R5,&TPCNT1      ; TPCNT1 still >= R5?
        JHS     L$1             ; Yes, no overflow
;
; Transition from 0FFh to 0 occured, read actual MSB;
; it now has the correct (value + 1).
;
        MOV.B   &TPCNT2,R6      ; Read actual MSBs  00yy
        DEC.B   R6              ; MSB - 1 is correct
L$1     SWPB    R6              ; MSBs to high byte  yy00
        ADD     R5,R6           ; Build 16-bit value in R6 yyxx
```

6.7.2.2 Pulse Width Modulation Mode

Figure 6–76 shows the generation of low frequency PWM with the Universal Timer/Port module. If the ACLK is used for the timing, then two PWM outputs with up to 256 Hz are possible. The software is described in the paragraph *PWM Digital-to-Analog Converter with the Universal Timer/Port Module* of section *Digital-to-Analog Converters*.
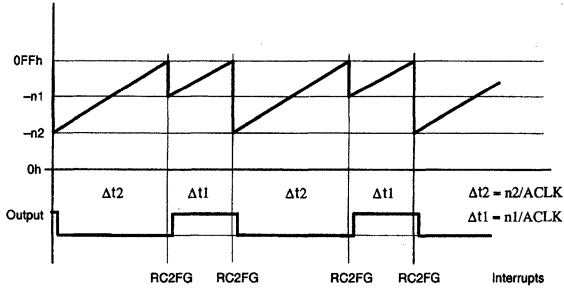
Figure 6–76. Low Frequency PWM Timing Generated With the Universal Timer/Port Module

6.8 The Crystal Buffer Output

This is a relatively simple module, but it can be very helpful. It allows the use of frequencies generated by the MSP430 for external modules without any influence to the accuracy of the crystal. Figure 6–77 shows an application with two MSP430s. The right side MSP430 uses the buffered crystal output — @ 32 kHz — of the left side MSP430. This allows both to be run with the accuracy of a crystal controlled oscillator, but only one crystal is necessary. The right side MSP430 uses the XBUF terminal for the output of the MCLK frequency to drive an external ASIC.
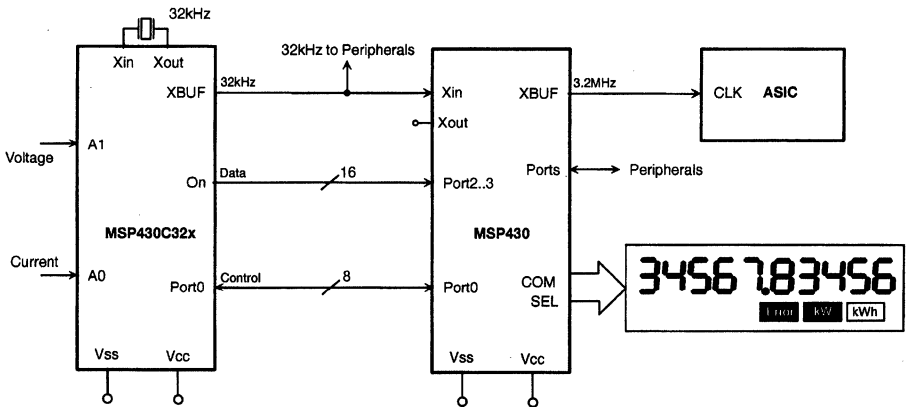
Figure 6–77. Two MSP430s Running From the Same Crystal

Only a single instruction is needed to implement the output of an internal frequency at the XBUF terminal:

```
; Hardware definitions for the Crystal Buffer
;
CBCTL     .equ    053h              ; Crystal Buffer Control Byte
CBE       .equ    001h              ; Enable XBUF output
CBACLK    .equ    000h              ; ACLK is output at XBUF
CBACLK2   .equ    002h              ; ACLK/2 is output at XBUF
CBACLK4   .equ    004h              ; ACLK/4 is output at XBUF
CBMCLK    .equ    006h              ; MCLK is output at XBUF
; Output the crystal frequency ACLK at pin XBUF
```

```
;
        MOV.B    #CBACLK+CBE,&CBCTL          ; ACLK to XBUF pin
;
; Output the half crystal frequency ACLK/2 at pin XBUF
;
        MOV.B    #CBACLK2+CBE,&CBCTL         ; ACLK/2 to XBUF pin
;
; Output the crystal frequency ACLK divided by four at pin XBUF
;
        MOV.B    #CBACLK4+CBE,&CBCTL         ; ACLK/4 to XBUF pin
;
; Output the MCLK frequency at pin XBUF
;
        MOV.B    #CBMCLK+CBE,&CBCTL          ; MCLK to XBUF pin
```

As shown with the previous definitions, four different frequencies can be output at terminal XBUF. With the CBE bit, these four frequencies can be enabled or disabled. The four possible frequencies are:

- ❑ MCLK The frequency of the system clock generator (DCO): 500 kHz to 4MHz

- ❑ ACLK The frequency of the crystal (normally 32768 Hz)

- ❑ ACLK/2 The halved crystal frequency (normally 16384 Hz)

- ❑ ACLK/4 The crystal frequency divided by 4 (normally 8192 Hz)

The crystal-buffer control byte CBCTL is a write-only byte. This means the full information must always be written to it. The following code sequence provides an output of ACLK and not — as it is intended — to output the MCLK frequency:

```
; Switch off and on the MCLK at pin XBUF
;
        MOV.B    #CBMCLK+CBE,&CBCTL ; MCLK to XBUF pin
        BIC.B    #CBE,&CBCTL        ; MCLK off
        BIS.B    #CBE,&CBCTL        ; WRONG: ACLK is output NOT MCLK
;
        MOV.B    #CBMCLK+CBE,&CBCTL ; CORRECT: MCLK on again
```

Figure 6–78 shows an application with a DC/DC converter that is controlled by the output frequency of the XBUF terminal. The converter is driven with the frequency that fits best the actual status (8.192 kHz, 16.384 kHz or 32.768 kHz). The sequence starts with the high output frequency and steps down after the buildup of the voltage to the 8 kHz frequency. No software overhead is necessary for the generation of this frequency.
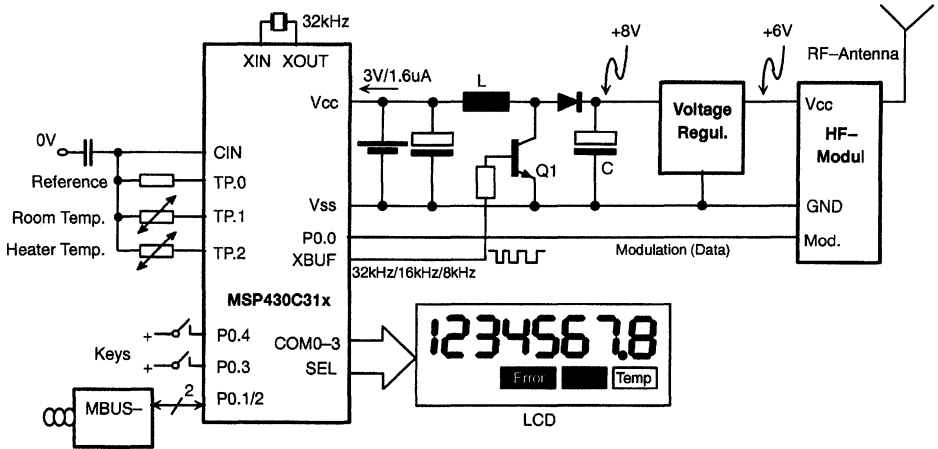
Figure 6–78. The Crystal Buffer Output Used for a DC/DC Converter

6.9 The USART Module

The universal synchronous asynchronous receive/transmit communication interface (USART) — whose block diagram is shown in figure 6–79 — can work in two different modes: the asynchronous mode and the synchronous mode. This section describes the software routines usable for the asynchronous mode (SCI, RS232).

Note:

It is recommended to also consult the data book *MSP430 Family Architecture Guide and Module Library.* The hardware-related information given there is very valuable and complements the information given in this section.

The examples and the hardware definitions shown use the addresses of the MSP430x33x. Future MSP430 family members may have different hardware addresses — especially for the I/O ports used.

The hardware features of the USART module substantially exceed the examples shown in this section. To get the USART running quickly, the UART mode is recommended, with or without the interrupt capability. The most often used features are included in the examples given.

Figure 6–79 shows the block diagram of the MSP430 USART module.
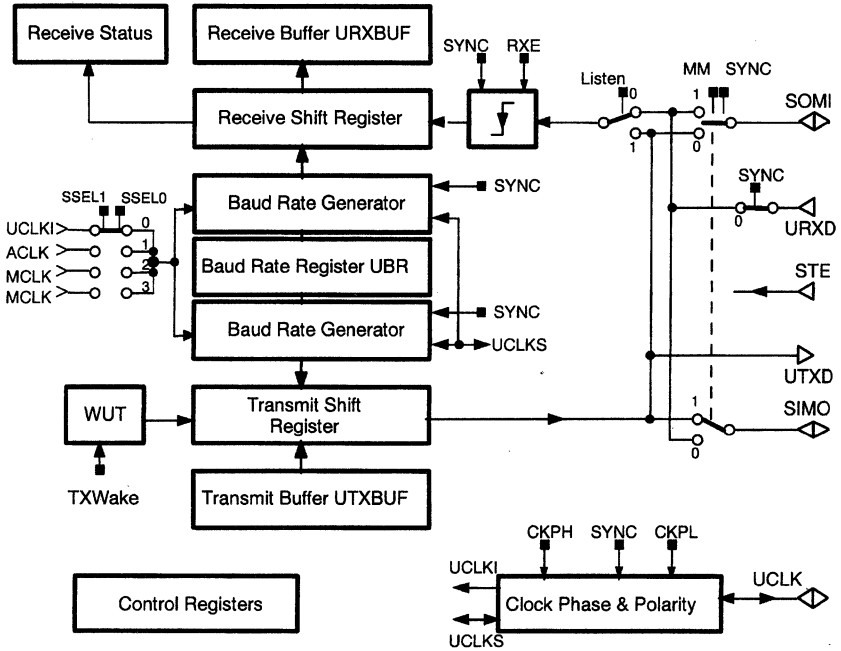
Figure 6–79. The MSP430 Family USART Hardware

6.9.1 Introduction

This chapter gives a short overview to the use of the MSP430 universal synchronous, asynchronous receive/transmit communication interface (USART) as an RS232 interface (also called serial controller interface, SCI). Tested software examples with and without the use of the interrupt capability are given for the transmission and the reception of UART signals (universal asynchronous receive/transmit). Full duplex mode is used for all examples running in the active mode and the low power mode 3 (LPM3).

If the USART is switched to the UART mode — made by setting the SYNC bit UCTL.2 to 0 — then the hardware of figure 6–79 reduces to the parts used by the UART as shown in figure 6–80.
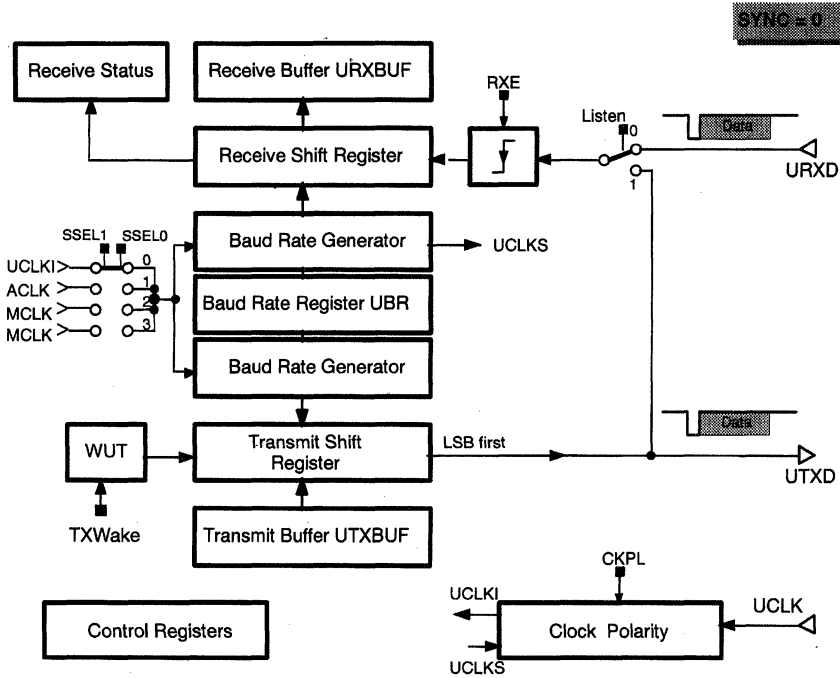
Figure 6–80. The USART Switched to the UART Mode

6.9.1.1 Definitions Used With the Application Examples

The abbreviations used for the hardware definitions are consistent with the *MSP430 Architecture User's Guide*, except for the stop bit definition (SP), which is a predefined symbol of the MSP430 assembler for the system stack pointer SP.

```
; HARDWARE DEFINITIONS
;
UCTL    .equ    070h            ; USART Control Register
SWRST   .equ    001h            ; 1: Software Reset 0: Run
SYNC    .equ    004h            ; 1: UART Mode     0: SCI Mode
CHAR    .equ    010h            ; 1: 8 Data Bits   0: 7 Data Bits
SP_     .equ    020h            ; 1: 2 Stop Bits   0: 1 Stop Bit
```

```
PEV      .equ    040h       ; 1: Even Parity    0: Odd Parity
PENA     .equ    080h       ; 1: Parity enabled 0: Parity dis.
;
UTCTL    .equ    071h       ; Transmit Control Register
TXEPT    .equ    001h       ; 1: Transmitter empty
URXSE    .equ    008h       ;
SSEL0    .equ    010h       ; Clock Selection 0: Ext. Clock
SSEL1    .equ    020h       ; 1: ACLK          2,3: MCLK
;
URCTL    .equ    072h       ; Receive Control Register
RXERR    .equ    001h       ; 1: Receive Error 0: No Error
URXEIE   .equ    008h       ; 1: all Char.    0: only w/o Error
BRK      .equ    010h       ; 1: Break detected 0: ok
OE       .equ    020h       ; 1: Overrun Error  0: ok
PE       .equ    040h       ; 1: Parity Error   0: ok
FE       .equ    080h       ; 1: Frame Error    0: ok
;
UMCTL    .equ    073h       ; Modulation Control Reg. m7..m0
UBR0     .equ    074h       ; Baud Rate Register 0
UBR1     .equ    075h       ; Baud Rate Register 1
URXBUF   .equ    076h       ; Receive Buffer
UTXBUF   .equ    077h       ; Transmit Buffer
;
IFG2     .equ    003h       ; SFRs: Flags
URXIFG   .equ    001h       ; Receive Flag IFG2.0
UTXIFG   .equ    002h       ; Transmit Flag IFG2.1
;
IE2      .equ    001h       ; SFRs: Interrupt Enable Bits
URXIE    .equ    001h       ; Receive Intrpt Enable Bit IE2.0
UTXIE    .equ    002h       ; Transmit Intrpt Enable Bit IE2.1
;
ME2      .equ    005h       ; SFRs: Mode Enable Bits
URXE     .equ    001h       ; Receiver Module Enable Bit ME2.0
UTXE     .equ    002h       ; Transmitter Mod. Enable Bit ME2.1
;
P4SEL    .equ    01Fh       ; Port4 Sel. Reg. (I/O <-> USART)
```

```
URXD      .equ      080h          ; Receive Input P4.7
UTXD      .equ      040h          ; Transmit Output P4.6
;
SCG1      .equ      080h          ; Low Power Mode bit 1
SCG0      .equ      040h          ; Low Power Mode bit 0
CPUoff    .equ      010h          ; Switches CPU off
GIE       .equ      008h          ; General Interrupt Enable Bit
;
SCFQCTL   .equ      052h          ; FLL multiplier and M bit
SCFI0     .equ      050h          ; FLL current switches
FN_2      .equ      004h          ; Switch for 2 MHz
FN_3      .equ      008h          ; Switch for 3 MHz
FN_4      .equ      010h          ; Switch for 4 MHz
```

6.9.1.2 MSP430 UART Attributes

A short overview to the USART running in the UART mode is given below:

❑ 7-bit and 8-bit data length is selectable

❑ Error detection for the receive path:

■ Frame error. The stop bits have space potential.

■ Parity error. Parity is enabled and the parity bit has the wrong value.

■ Overrun error. The next character is read in before the last one is read out by the software.

■ Break detect. The URXD terminal has low potential for more than 10 bits

❑ Baud rate generation possible also from 32 kHz crystal due to the modulation register

❑ Interrupt-driven transmit and receive functions

❑ Two independent interrupt vectors, one for transmit and one for receive

❑ Full functionality also during LPM3

❑ End-of-frame flag usable with interrupt or polling

6.9.1.3 Data Format

The RS232 data format is used. Figure 6–81 shows how this format is seen at the MSP430 ports (URXD and UTXD) and Figure 6–82 how it is defined on the transmission line. The format shown in Figures 6–81 and 6–82 is:

❑ Seven data bits. The least significant bit follows the start bit

❑ Parity enabled. The parity bit follows the most significant bit of the data

❑ No address bit. This is the normal case

❑ Two stop bits
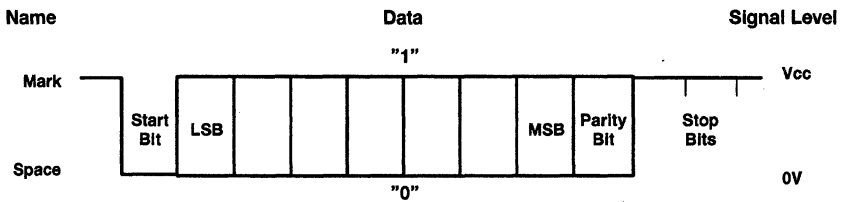
Figure 6–81. The RS232 Format (Levels at the MSP430)

The signal on the transmission line has the inverted state as seen at the MSP430 ports and different voltage potentials. Figure 6–82 shows this.
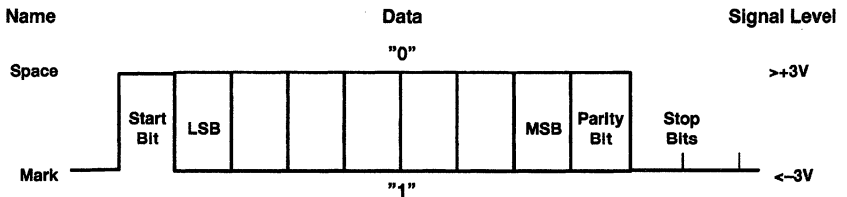
Figure 6–82. The RS232 Format (Levels on the Transmission Line)

6.9.1.4 UART Hardware Registers

The USART is controlled by seven control registers and one read-only register. All are 8-bit registers and therefore should be accessed only with byte instruc-

tions. Figure 6–83 gives an overview to these eight registers including the names, assembler mnemonics, hardware addresses and the initial states. The register and bit definitions are contained in Section 6.9.1.

Register Name	Mnemonic	Register Access	Address	Initial State
USART Control Register	UCTL	Read/write	070h	See below
Transmit Control Register	UTCTL	Read/write	071h	See below
Receive Control Register	URCTL	Read/write	072h	See below
Modulation Control Reg.	UMCTL	Read/write	073h	unchanged
Baud Rate Register 0	UBR0	Read/write	074h	unchanged
Baud Rate Register 1	UBR1	Read/write	075h	unchanged
Receive Buffer	URXBUF	Read Only	076h	unchanged
Transmit Buffer	UTXBUF	Read/Write	077h	unchanged

Figure 6–83. USART Control Registers Used for the UART Mode

6.9.2 Baud Rate Generation

To generate the desired baud rate from a relatively high frequency (1 MHz to 5 MHz) is a simple task. The resulting baud rate error is small due to the large integer part of the quotient compared to the truncated fractional part. This changes completely if the timebase is a crystal of only 32 kHz. Then the error due to the truncated fractional part of the quotient gets large and leads to the loss of synchrony at the trailing bits of the frame. The MSP430 USART therefore uses a correction to keep the baud rate error small. The modulation register, UMCTL, contains 8-bit data to correct the baud rate of the received or transmitted UART signal. The bits define how the predivider information contained in the two baud rate registers UBR0 and UBR1 is used:

❑ A *0* bit in the UMCTL register means that the information contained in UBR1/UBR0 is used as is.

❑ A *1* bit means that the 16-bit content of UBR1/UBR0 is incremented by one and used with this value. The content of UBR1/UBR0 is not changed.
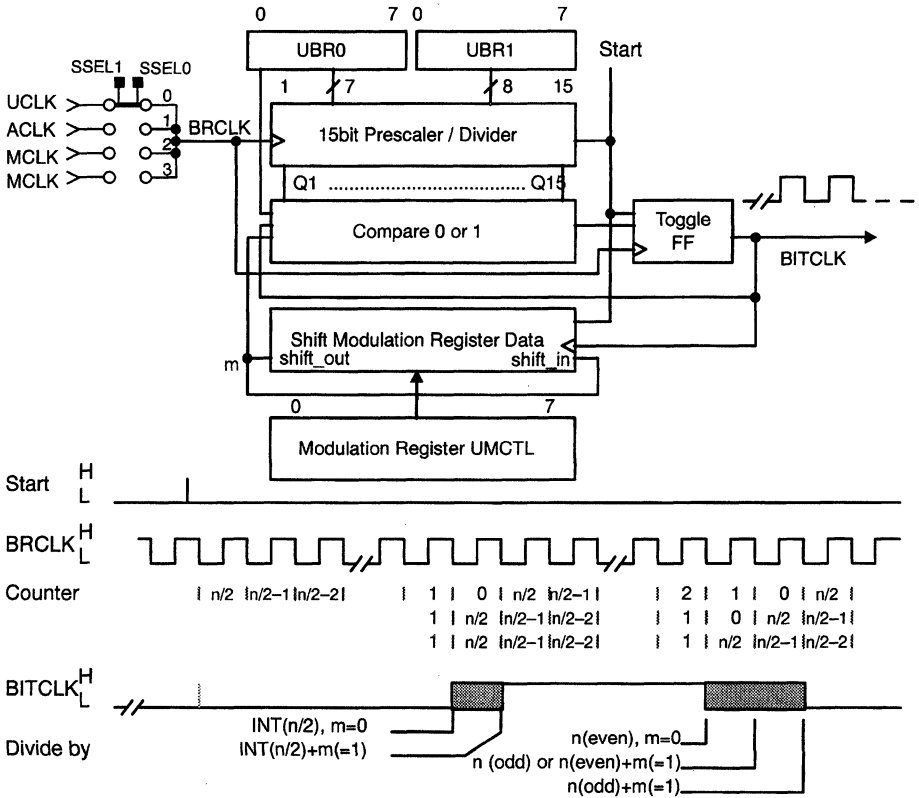
Figure 6–84. The Baud Rate Generator

The LSB (m0) of register UMCTL is used for the start bit, the next bit (m1) is used for the LSB of the data, and so on. After the use of bit m7, the bit sequence m0 to m7 is used again. See figure 6–85 for an explanation.

Example 6–58. 4800 Baud from 32 kHz Crystal

The baud rate of 4800 is needed from a crystal frequency of 32,768Hz. This is necessary because the UART also needs to run during the low power mode 3. With only the ACLK available, the theoretical division factor — the truncated value is the content of the baud rate register UBR (UBR1/UBR0) — is:

$$UBR = \frac{32768}{4800} = 6.82667$$

This means the baud rate register, UBR1, (MSBs) is loaded with 0 and the UBR0 register contains 6. To get a rough value for the 8-bit modulation register, UMCTL, the fractional part (0.826667) is multiplied by 8 (the number of bits in the register UMCTL):

$$UMCTL = 0.82667 \times 8 = 6.613$$

The rounded result, 7, is the number of 1s to be placed into the modulation register, UMCTL. The resulting, corrected baud rate with the UMCTL register containing seven 1s is:

$$BaudRate = \frac{32768}{\left(\frac{7 \times 7 + 1 \times 6}{8}\right)} = 4766.2545$$

This results in an average baud rate error of:

$$Baud\ Rate\ Error = \frac{4766.2545 - 4800}{4800} \times 100 = -0.703\%$$

To get the bit sequence for the modulation register, UMCTL, that fits best, the following algorithm can be used. The fractional part of the theoretical division factor is summed eight times and if a carry to the integer part occurs, the actual m-bit is set. Otherwise it is cleared. An example with the above fraction 0.82667 follows:

Fraction Addition	Carry to next Integer	UMCTL	Bits
0.82667 + 0.82667 = 1.65333	Yes	m0	1
1.65333 + 0.82667 = 2.48000	Yes	m1	1
2.48000 + 0.82667 = 3.30667	Yes	m2	1
3.30667 + 0.82667 = 4.13333	Yes	m3	1
4.13333 + 0.82667 = 4.96000	No	m4	0
4.96000 + 0.82667 = 5.78667	Yes	m5	1
5.78667 + 0.82667 = 6.61333	Yes	m6	1
6.61333 + 0.82667 = 7.44000	Yes	m7	1

The result of the calculated bits m7...m0 (11101111b) is EFh with seven 1s. In Section 6.9.3.3.2, a software macro (CALC_UMCTL) is contained that uses the algorithm shown above. It calculates for every combination of the USART clock and the desired baud rate, the optimum value for the modulation register, UMCTL. For the above example, the algorithm also finds EFh with its seven 1s.

A second software macro (CALC_UBR) calculates the values for the two UBR registers.

Example 6–59. 2400 Baud From 32 kHz ACLK

Figure 6–85 gives an example for a baud rate of 2400 generated with the ACLK frequency (32,768 Hz). The data format for figure 6–85 is:

Eight data bits, parity enabled, no address bit, two stop bits. Figure 6–85 shows three different frames:

❑ The upper frame is the correct one with a bit length of 13.65333 ACLK cycles (32,768/2400 = 13.65333)

❑ The middle frame uses a rough estimation with 14 ACLK cycles for the bit length

❑ The lower frame shows a corrected frame using the best fit (6Dh) for the modulation register.

It can be seen that the approximation with 14 ACLK cycles accumulates an error of more than 0.3 bit length after the second stop bit. The error of the corrected frame is only 0.011 bit length. The error of the crystal clock is not yet included, but it adds to the above errors.
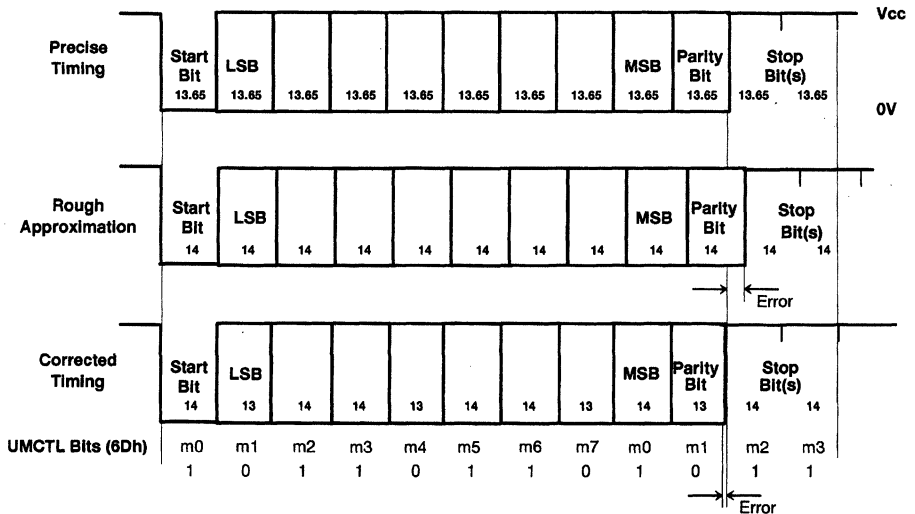
Figure 6–85. Baud Rate Correction Function

Tables 6–33 and 6–34 contain the average errors (full frame) for the normally used baud rates when produced with the described baud rate generation.

The software examples contain software MACROs that automatically insert the correct values for the UBR registers and the modulation register, UMCTL. The software MACROs — that do not need ROM or RAM — may be hidden in the listing by a .mnolist assembler directive. See Section 6.9.3.3.2.

6.9.2.1 Baud Rate Generation With the MCLK

Table 6–33 shows the optimum values for the UBR and UMCTL registers. The UART clock is the MCLK (1,048 MHz). The values for the UMCTL and UBR1/UBR0 registers are calculated by the software MACROs in Section 6.9.3.3.2. The crystal error is not included.

Table 6–33 contains the following columns:

Baud Rate — The baud rate for the data exchange (transmit and receive use the same baud rate)

Division Factor — The quotient UARTCLK/baud rate

UBR1/UBR0 Content — The truncated 16-bit hexadecimal result of the division factor (UARTCLK/baud rate). The value is calculated by the software macro CALC_UBR. The high byte is the UBR1 value, the low byte is the UBR0 value

Calculated UMCTL Content — The 8-bit result that fits best for the modulation register. It is calculated by the software macro CALC_UMCTL.

Used Fraction — The number of 1s in the Modulation Register divided by eight. It is an approximation to the truncated fractional part of the division factor.

Mean Error — The resulting error of a complete character caused by the approximation to the division factor

Table 6–33.   Baud Rate Register UBR Content (MCLK = 1,048 MHz)

BAUD RATE	DIVISION FACTOR	UBR1/UBR0 CONTENT	CALCULATED UMCTL CONTENT	FRACTION USED	MEAN ERROR (%)
110	9532.51	253Ch	55h	0.5	+0.000
300	3495.25	0DA7h	44h	0.25	0.000
600	1747.63	06D3h	6Dh	0.625	+0.000
1200	873.81	0369h	EFh	0.875	–0.007
2400	436.91	01B4h	FFh	1.000	–0.002
4800	218.45	00DAh	AAh	0.50	–0.023
9600	109.23	006Dh	88h	0.25	–0.018
19200	54.61	0036h	ADh	0.625	–0.027
38400	27.31	001Bh	24h	0.25	+0.220

6.9.2.2 Baud Rate Generation With the ACLK

With the relatively low ACLK frequency (32,768 Hz), the modulation register UMCTL becomes much more important compared to the normally high MCLK frequency used for the UART timing. Table 6–34 shows the optimum values for the UBR and UMCTL registers for commonly used baud rates generated with the ACLK (32,768 Hz). The table values are calculated by the MACROs described in Section 6.9.3.3.2. The crystal is considered to be without frequency error. The table columns are described in Section 6.9.2.1.

Table 6–34.   Baud Rate Registers UBR Content (ACLK = 32,768 Hz)

BAUD RATE	DIVISION FACTOR	UBR1/UBR0 CONTENT	CALCULATED UMCTL CONTENT	FRACTION USED	MEAN ERROR (%)
110	297.8909	0129h	FFh	1.00	–0.04
300	109.2267	006Dh	88h	0.25	–0.02
600	54.6133	0036h	ADh	0.625	–0.02
1200	27.3067	001Bh	24h	0.25	+0.21
2400	13.6533	000Dh	6Dh	0.625	+0.21
4800	6.8267	0006h	EFh	0.875	–0.71
9600	3.4133	0003h	4Ah	0.375	+1.12
19200	1.7067	–			
38400	0.8533	–			

6.9.3 Software Routines

The following sections show proven software routines, subroutines, and soft-
ware MACROs for the UART mode of the USART.

Note:

The program sequence for the initialization of the UART is important. As long
as the SWRST bit (UCTL.0) is set, the receive and transmit control registers
URCTL and UTCTL cannot be initialized. The program sequences given in
the software examples comply with this fact and are therefore recommen-
ded.

As long as the SWRST bit is set, the following control bits are held in the 0
state: TXWAKE, RXERROR, RXWAKE, BRK, OE, FE, PE, URXIFG, URXIE,
UTXIE.

The following control bits are held in the 1 state: UTXIFG, TXEPT

6.9.3.1 NonInterrupt Processing

The simplest way to use the USART is in the UART mode. The interrupt is not
enabled, the software checks if it is possible to output the next byte (UTXIFG
= 1) and it checks if a new character is received (URXIFG = 1).

Example 6–60. Full Duplex Modem

A full duplex UART software running without the use of the UART interrupt is
shown. It is designed for:

❑ Baud rate: 1200 baud

❑ The MCLK (1.048 MHz) is used for the UART clock

❑ Eight data bits

❑ Two stop bits

❑ Parity enabled with odd parity

❑ Receive of errorfree characters only

```
STACK     .equ     0600h            ; Stack start address
;
; Definitions for the UART part: user defined
;
```

```
Baudr     .equ     1200              ; Baudrate is 1200 Baud
FLLMPY    .equ     32                ; FLL multiplier for 1,048MHz
UARTCLK   .equ     FLLMPY*32768      ; MCLK is used for UARTCLK
;
; The content for the UMCTL and UBR registers are calculated.
; The two software macros do not use RAM or ROM, they only
; define the variables CUMCTL, CUBR1 and CUBR0 for the
; UART registers UMCTL, UBR1 and UBR0
;
          CALC_UMCTL                 ; Calc. Modulation Reg. content
          CALC_UBR                   ; Calculate UBR1/UBR0 contents
;
          .text                      ; Software start address
;
INIT      MOV      #STACK,SP         ; Initialize Stack Pointer
          CALL     #INITSR           ; Init. FLL and RAM
          ...                        ; Proceed with initialization
;
; Initialize the UART: Odd parity, 8 data bits, 2 stop bits
; MCLK for UART clock
;
          MOV.B    #CUMCTL,&UMCTL    ; Modulation Register
          MOV.B    #CUBR0,&UBR0      ; Baud Rate Register low
          MOV.B    #CUBR1,&UBR1      ; Baud Rate Register high
          BIS.B    #URXD+UTXD,&P4SEL ; Select RXD + TXD at Port4
          BIS.B    #UTXE+URXE,&ME2   ; Enable USART Moduls
          MOV.B    #PENA+SP_+CHAR,&UCTL ; USART Control Register
          MOV.B    #SSEL1+SSEL0,&UTCTL ; Transmit Control Reg. MCLK
          MOV.B    #0,&URCTL         ; Receive Control Register
          ...                        ; Continue with initialization
.;
MAINLOOP ...                         ; Start Mainloop
;
; UART parts within the mainloop.
; The software checks these two parts regularly.
; UART Receive part:
```

```
; check if a new character is received
; R7 contains the received information.
        BIT.B   #RXERR,&URCTL     ; Error during receive?
        JZ      L$3               ; No
        ...                       ; Error handling
        BIC.B   #FE+PE+OE+BRK+RXERR,&URCTL ; Clear error flags
        JMP     L$2               ; Continue in mainloop
;
L$3     BIT.B   #URXIFG,&IFG2     ; Character received?
        JZ      L$2               ; No, proceed in mainloop
        MOV.B   &URXBUF,R7,       ; Yes, move character to R7
L$2     ...                       ; Continue in mainloop
;
; UART Transmit part:
; check if the next character can be transmitted.
; R6 contains information to be transmitted.
;
        BIT.B   #UTXIFG,&IFG2     ; Transmit buffer empty?
        JZ      L$1               ; No, wait
        MOV.B   R6,&UTXBUF        ; Empty: move info to TX buffer
        MOV.B   src,R6            ; Next character to R6
L$1     ...                       ; Continue with mainloop
        BR      #MAINLOOP         ; End of mainloop
;
; Interrupt Vectors
;
        .sect   "INITVEC",0FFFEh  ; Reset Vector
        .word   INIT              ; Program Start Address
```

If the above software is to be used with the ACLK for the UART clock, then only the following two source lines need to be modified:

```
UARTCLK .equ    32768             ; ACLK is used for UARTCLK
;
        MOV.B   #SSEL0,&UTCTL     ; Transmit Control Register ACLK
```

All other necessary modifications are made automatically by the macros CALC_UMCTL and CALC_UBR.

6.9.3.2 Interrupt Processing

This is the normal mode for the use of the UART. Interrupt is requested if the general interrupt enable bit GIE (SR.3) is set and

❏ A character is transmitted and the transmit interrupt is enabled (IE2.1 = 1)
       or

❏ A character is received and the receive interrupt is enabled (IE2.0 = 1)

Note:

. If an error occurred during the reception of a character, then the error flags of the Receive control register (PE, FE, BRK, and RXERR) must be reset within the UART interrupt handler. Otherwise, the set error flags will block the next interrupt. This is not the case for the overrun error flag OE.

6.9.3.2.1 MCLK Used for the UART Clock

The following example is for when the MCLK is used for the generation of the UART clock or for external frequencies in the MCLK range (500 kHz to 3.8 MHz).

For high baud rates — higher than 38400 baud — dedicated CPU registers may be necessary to lower the interrupt overhead. The time for the saving and restoring of the register is not necessary. The software example shown in Section 6.9.3.2.2 uses dedicated registers.

Example 6–61. Full Duplex UART

Full duplex UART software using the two UART interrupts is shown. It is designed for:

❏ Baud rate: 19200 baud

❏ The MCLK (3.144 MHz) is used for the UART clock

❏ Seven data bits

❏ One stop bit

❏ Parity enabled with even parity

❏ Receive of errorfree characters only

Transmit Part — the start address xxxx is loaded into the pointer TXPOI and the number of characters to be output is loaded into the character count

TXCNT. The interrupt routine outputs the programmed character sequence starting at address xxxx.

Receive Part — the start address yyyy of a RAM buffer is loaded into the pointer RCPOI and the number of characters to be received is loaded into the character count RCCNT. The interrupt routine receives the characters and stores them into the buffer. Only error-free characters are accepted.

```
STACK    .equ    0600h              ; Stack start address
;
; Definitions for the UART part
;
Baudr    .equ    19200              ; Baudrate is 19200 Baud
FLLMPY   .equ    96                 ; FLL multiplier for 3,144MHz
UARTCLK  .equ    FLLMPY*32768       ; MCLK is used for UARTCLK
;
         .even                      ; Word boundary
         .bss    TXPOI,2            ; Pointer to transmit buffer
         .bss    RCPOI,2            ; Pointer to receive buffer
         .bss    TXCNT,1            ; Counter/status for transmit
         .bss    RCCNT,1            ; Counter/status for receive
;
; The content for the UMCTL and UBR registers are calculated
; The two software macros do not use RAM or ROM
;
         CALC_UMCTL                 ; Calculate Mod. Reg. content
         CALC_UBR                   ; Calculate UBR1/UBR0 contents
;
         .text                      ; Software start address
;
INIT     MOV     #STACK,SP          ; Initialize Stack Pointer
         CALL    #INITSR            ; Init. FLL and RAM
         ...                        ; Proceed with initialization
;
; Initialize the UART: Even parity, 7 data bits, 1 stop bit
; MCLK for UART clock, only errorfree characters to URXBUF
;
         MOV.B   #CUMCTL,&UMCTL     ; Modulation Register
```

```
        MOV.B    #CUBR0,&UBR0      ; Baud Rate Register low
        MOV.B    #CUBR1,&UBR1      ; Baud Rate Register high
        BIS.B    #URXD+UTXD,&P4SEL ; Select RXD + TXD at Port4
        BIS.B    #UTXE+URXE,&ME2   ; Enable USART Moduls
        MOV.B    #PENA+PEV,&UCTL   ; USART Control Register
        MOV.B    #SSEL1+SSEL0,&UTCTL ; Transmit Control Reg. MCLK
        MOV.B    #0,&URCTL         ; Receive Control Register
        BIS.B    #UTXIE+URXIE,&IE2 ; Enable USART interrupts
        CLR.B    TXCNT            ; Disable transmit
        CLR.B    RCCNT            ; Disable receive
        ...                       ; Continue with initialization
        EINT                      ; Enable interrupt
;
MAINLOOP ...                      ; Start of Mainloop
;
; Preparation for the reception of m bytes. The input
; buffer starts at address yyyy
;
        TST.B    RCCNT            ; Data input completed?
        JNZ      L$1              ; No, wait
        MOV      #yyyy,RCPOI      ; Buffer start address to RCPOI
        MOV.B    #m,RCCNT         ; Number of bytes to RCCNT
L$1     ...                       ; Continue in mainloop
;
; Stop the reception of data. The currently received character
; is input completely
;
        CLR.B    RCCNT            ; Status to zero
        ...                       ; Continue
;
; Preparation for the transmission of n bytes starting at
; address xxxx. A check is made if the last transmit operation
; is really completed.
;
        BIT.B    #TXEPT,&UTCTL    ; Transmit part ready?
        JZ       L$2              ; No, buffers are not yet empty
```

```
;
        MOV.B    #n-1,TXCNT        ; Ready, init. byte count
        MOV      #xxxx+1,TXPOI     ; Init. transmit buffer pointer
        MOV.B    xxxx,&UTXBUF      ; First info byte to TX buffer
L$2     ...                        ; Continue in background
;
; Stop the transmission of data. The currently sent character
; is transmitted completely
;
        CLR.B    TXCNT             ; Status to zero
        ...
;
; Interrupt Handlers
; Interrupt handler for the UART Receive part.
;
RCINT   TST.B    RCCNT             ; Reception allowed?
        JZ       RCRET             ; No, status is 0
        BIT.B    #RXERR,&URCTL     ; Error during receive?
        JNZ      RCERR             ; Yes
        DEC.B    RCCNT             ; No, Byte count -1
        PUSH     R5                ; Save R5
        MOV      RCPOI,R5          ; Pointer to buffer
        MOV.B    &URXBUF,0(R5)     ; Next byte to buffer
        INC      R5                ; To next buffer byte
        MOV      R5,RCPOI          ; Update pointer
        POP      R5                ; Restore R5
RCRET   RETI
;
RCERR   ...                        ; Error handling
        BIC.B    #FE+PE+OE+BRK+RXERR,&URCTL ; Clear error flags
        RETI
;
; Interrupt handler for the UART Transmit part.
;
TXINT   TST.B    TXCNT             ; Something to transmit?
        JZ       TXRET             ; No, buffer is empty
```

```
        DEC.B    TXCNT              ; Byte count -1
        PUSH     R5
        MOV      TXPOI,R5           ; Pointer to buffer
        MOV.B    @R5+,&UTXBUF       ; Next byte for output
        MOV      R5,TXPOI           ; Update pointer
        POP      R5
TXRET   RETI
; Interrupt Vectors
;
        .sect    "SCIVEC",0FFECh    ; USART Interrupt Vectors
        .word    TXINT              ; Transmit Vector
        .word    RCINT              ; Receive Vector
        .sect    "INITVEC",0FFFEh   ; Reset Vector
        .word    INIT               ; Program Start Address
```

6.9.3.2.2 ACLK Used for the UART Clock

The following example is for when the ACLK is used for the generation of the UART clock or for external frequencies lower than 100 kHz. It is very similar to that of Section 6.9.3.2.1. The ACLK can also be used as the UART clock. See that section for details.

This section shows another approach, however. The CPU is normally off and leaves the LPM3 only when the programmed number of received or transmitted characters is reached.

Example 6–62. Full Duplex UART With Interrupt

Full duplex UART software using the UART interrupt is shown. It is designed for:

❑ Baud rate: 2400 baud

❑ The ACLK (32,768 Hz) is used for the UART clock

❑ Eight data bits

❑ Two stop bit

❑ Parity enabled with odd parity

❑ Receive of errorfree characters only

❑ The CPU normally uses the low power mode 3 (LPM3)

Transmit Part — the start address xxxx of the output sequence is loaded into the pointer TXPOI and the number of characters *m* is loaded into the character count TXCNT. The interrupt routine outputs the character sequence and when TXCNT reaches 0 (output completed), it resets the CPUoff bit of the stored status register on the stack. This manipulation omits the return to LPM3 and initializes the next transmit sequence. R6 is exclusively used for the transmit part.

Receive Part — the start address yyyy of a RAM buffer is loaded into the pointer RCPOI and the number of characters *n* is loaded into the character count RCCNT. The interrupt routine receives the characters and stores them in the buffer until RCCNT reaches 0 (input completed). Then it resets the CPUoff bit of the stored status register on the stack. This manipulation omits the return to LPM3 and allows the processing of the received data. Only errorfree characters are accepted. R7 is exclusively used for the receive part.

```
STACK     .equ     0600h               ; Stack start address
;
; Definitions for the UART part
;
Baudr     .equ     2400                ; Baudrate is 2400 Baud
FLLMPY    .equ     64                  ; FLL multiplier for 2,096MHz
UARTCLK   .equ     32768               ; ACLK is used for UARTCLK
;
          .bss     TXCNT,1             ; Counter/status for transmit
          .bss     RCCNT,1             ; Counter/status for receive
          CALC_UMCTL                   ; Calculate Mod. Reg. content
          CALC_UBR                     ; Calculate UBR1/UBR0 contents
;
          .text                        ; Software start address
;
INIT      MOV      #STACK,SP           ; Initialize Stack Pointer
          CALL     #INITSR             ; Init. FLL and RAM
          ...                          ; Proceed with initialization
;
; Initialize the UART: Odd parity, 8 data bits, 2 stop bits
; ACLK used for the UART clock
;
          MOV.B    #CUMCTL,&UMCTL      ; Modulation Register
```

```
          MOV.B    #CUBR0,&UBR0      ; Baud Rate Register low
          MOV.B    #CUBR1,&UBR1      ; Baud Rate Register high
          BIS.B    #URXD+UTXD,&P4SEL ; Select RXD + TXD at Port4
          BIS.B    #UTXE+URXE,&ME2   ; Enable USART Moduls
          MOV.B    #PENA+SP_+CHAR,&UCTL ; USART Control Register
          MOV.B    #SSEL0,&UTCTL     ; Transmit Contr. Reg. ACLK
          MOV.B    #0,&URCTL         ; Receive Control Register
          BIS.B    #UTXIE+URXIE,&IE2 ; Enable USART interrupts
          CLR.B    TXCNT             ; Disable transmit
          CLR.B    RCCNT             ; Disable receive
          ...                        ; Continue with initialization
          EINT                       ; Enable interrupt (GIE = 1)
;
MAINLOOP ...                         ; Start Mainloop
;
; Preparation for the reception of m bytes. Buffer starts
; at address yyyy. R7 is a dedicated register for receive
;
          TST.B    RCCNT             ; Ready?
          JNZ      L$1               ; No, RCCNT > 0
          MOV      #yyyy,R7          ; Receive buffer start address
          MOV.B    #m,RCCNT          ; Number of bytes
L$1       ...
;
; Stop the reception of data. The actually received character
; is input completely
;
          CLR.B    RCCNT             ; Status is zero
          ...
;
; Preparation for the transmission of n bytes starting at
; address xxxx. R6 is a dedicated register for transmit.
; The check for the empty TX buffer is faster, but needs more
; ROM bytes.
;
          TST.B    TXCNT             ; Ready for next characters?
```

```
        JNZ      L$2                ; No, TXCNT > 0
        BIT.B    #UTXIFG,&IFG2      ; TX part also ready?
        JZ       L$2                ; No, busy
;
        MOV.B    #n-1,TXCNT         ; Ready, init. byte count
        MOV      #xxxx+1,R6         ; Init. transmit buffer pointer
        MOV.B    xxxx,&UTXBUF       ; First info byte to TX buffer
L$2     ...                         ; Continue in background
;
; Stop the transmission of data. The actually sent character
; is transmitted completely
;
        CLR.B    TXCNT              ; Status is zero
        ...
;
; After the completion of all tasks, the program enters LPM3
;
PLPM3   BIS      #CPUoff+GIE+SCG1+SCG0,SR   ; Enter LPM3
;
; An interrupt handler cleared the CPUoff bit on the stack.
; Checks are made if activity is needed:
: Receive:      receive input buffer full
; Transmit:     transmit buffer output completely
; ...           other interrupt handlers
;
        TST.B    RCCNT              ; Receive completed?
        JZ       PROCRC             ; Yes, process received data
        TST.B    TXCNT              ; Transmit completed?
        JZ       NXTTX              ; Yes, prepare next characters
        ...                         ; Other handlers?
        JMP      PLPM3              ; Back to LPM3
;
; Interrupt Handlers
; Interrupt handler for the UART Receive part. R7 is used
; only for the receive part
;
```

```
RCINT   TST.B   RCCNT               ; Reception allowed?
        JZ      RCRET               ; No, status is 0
        BIT.B   #RXERR,&URCTL       ; Error during receive?
        JNZ     RCERR               ; Yes
        DEC.B   RCCNT               ; Byte count -1
        MOV.B   &URXBUF,0(R7)       ; Next byte to buffer
        INC     R7                  ; To next buffer byte
        TST.B   RCCNT               ; Buffer filled?
        JNZ     RCRET               ; No, proceed
        BIC     #CPUoff+SCG1+SCG0,0(SP) ; Active Mode after RETI
RCRET   RETI
;
RCERR   ...                         ; Error handling
        BIC.B   #FE+PE+OE+BRK+RXERR,&URCTL ; Clear error flags
        RETI
;
; Interrupt handler for the UART Transmit part. R6 is used
; only for the transmit part
;
TXINT   TST.B   TXCNT               ; Something to transmit?
        JZ      TXRET               ; No, buffer is empty
        DEC.B   TXCNT               ; Byte count -1
        MOV.B   @R6+,&UTXBUF        ; Next byte for output
        TST.B   TXCNT               ; Buffer output?
        JNZ     TXRET               ; No, proceed
        BIC     #CPUoff+SCG1+SCG0,0(SP) ; Active Mode after RETI
TXRET   RETI
;
; Interrupt Vectors
;
        .sect   "SCIVEC",0FFECh     ; USART Interrupt Vectors
        .word   TXINT               ; Transmit Vector
        .word   RCINT               ; Receive Vector
        .sect   "INITVEC",0FFFEh    ; Reset Vector
        .word   INIT                ; Program Start Address
```

6.9.3.3 Subroutines and .MACROs

The subroutines and assembler .MACROs used with the previous examples are contained in this section.

6.9.3.3.1 Subroutines

The initialization subroutine INITSR — which is explained in detail in the section *Timer_A* — checks first if a power-up clear (PUC) or a power-on reset (POR) has occurred:

❑ Power-Up Clear — the supply voltage is switched on, the RAM is cleared

❑ Power-On Reset — a reset occurred (RST/NMI terminal or by watchdog) the RAM is not changed

The two situations are distinguished by the content of the word INITKEY. If it contains 0F05Ah, the power-on reset state is assumed. Otherwise the power-up clear state is assumed.

The subroutine selects the correct current switch FN_x for the system clock generator and waits 30000 clock cycles to ensure that it has locked at the correct oscillator tap.

```
; Common Initialization Subroutine
; Check the INITKEY value first:
; If value is 0F05Ah: a reset occurred, RAM is not cleared
; otherwise Vcc was switched on: complete initialization
;
INITSR   CMP     #0F05Ah,INITKEY   ; PUC or POR?
         JEQ     IN0               ; Key is ok, continue program
         CALL    #RAMCLR           ; Restart completely: clear RAM
         MOV     #0F05Ah,INITKEY ; Define "initialized state"
;
IN0      MOV.B   #FLLMPY-1,&SCFQCTL ; Define MCLK frequency
;
         .if     FLLMPY < 48       ; Use the right DCO current:
         MOV.B   #0,&SCFI0         ; MCLK < 1.5MHz: FN_x off
         .else
         .if     FLLMPY < 80       ; 1.5MHz < MCLK < 2.5MHz?
         MOV.B   #FN_2,&SCFI0      ; Yes, FN_2 on
```

```
        .else                   ;
        .if     FLLMPY < 112    ; 2.5MHz < MCLK < 3.5MHz?
        MOV.B   #FN_3,&SCFI0    ; Yes, FN_3 on
        .else
        MOV.B   #FN_4,&SCFI0    ; MCLK > 3.5MHz: FN_4 on
        .endif
        .endif
        .endif
;
        MOV     #10000,R5       ; Allow the FLL to settle
IN1     DEC     R5              ; at the correct DCO tap
        JNZ     IN1             ; during 30000 cycles
        RET                     ; Return from initialization
;
; Subroutine for the clearing of the RAM block
;
        .bss    INITKEY,2,0200h ; 0F05Ah: initialized state
RAMSTRT .equ    0200h           ; Start of RAM
RAMEND  .equ    05FEh           ; Highest RAM address (33x)
;
RAMCLR  CLR     R5              ; Prepare index register
RCL     CLR     RAMSTRT(R5)     ; 1st RAM address
        INCD    R5              ; Next word address
        CMP     #RAMEND-RAMSTRT+2,R5 ; RAM cleared?
        JLO     RCL             ; No, once more
        RET                     ; Yes, return
```

6.9.3.3.2 .MACROs

The following two software macros calculate the values for the UART baud rate generator that fit best. They do not use ROM or RAM — they only define the three variables CUBR1, CUBR0, and CUMCTL that are used during the initialization of the UART registers UBR1, UBR0, and UMCTL.

```
          .mnolist                    ; Do not list macro calls
;
; The values for the Modulation Registers UBR1/UBR0 are
; calculated: CUBR1 and CUBR0 contain the truncated result
; of the division UARTCLK/Baudr
;
CALC_UBR .macro
CUBR1     .equ    UARTCLK/(Baudr*256)       ; Baud Rate Reg. High
CUBR0     .equ    (UARTCLK/Baudr)-256*CUBR1 ; Baud Rate Reg. Low
          .endm
```

The calculation for the content of the Modulation Register UMCTL follows. Seven bits of resolution are used.

```
CALC_UMCTL        .macro
;
; Modulation Register content: the rounded fraction of
; CMOD = UARTCLK/Baudr    is calculated
; Binary format of CMOD: 0.xxxxxxx
; Then the 8 bits of UMCTL are built.
; Inputs:         UARTCLK, Baudr   ; Frequencies [Hz]
; Output:         CUMCTL           ; 8-bit UMCTL register value
;
CMOD .equ         ((((256*UARTCLK)/Baudr)-256*(UARTCLK/Baudr))+1)/2
M$00     .equ     CMOD+CMOD        ; Fraction x 2
         .if      M$00>127         ; Overflow to integer?
M$10     .equ     M$00-128+CMOD    ; Yes, subtract 1.000000
C$0      .equ     1                ; UMCTL.0 = 1
         .else
M$10     .equ     M$00+CMOD        ; No, add fraction
C$0      .equ     0                ; UMCTL.0 = 0
```

```
            .endif
            .if     M$10>127        ; Overflow to integer?
M$20        .equ    M$10-128+CMOD   ; Yes, subtract 1.000000
C$1         .equ    2               ; UMCTL.1 = 1
            .else
M$20        .equ    M$10+CMOD       ; No, add fraction
C$1         .equ    0               ; UMCTL.1 = 0
            .endif
            .if     M$20>127        ; Overflow to integer?
M$30        .equ    M$20-128+CMOD   ; Yes, subtract 1.000000
C$2         .equ    4               ; UMCTL.2 = 1
            .else
M$30        .equ    M$20+CMOD       ; No, add fraction
C$2         .equ    0               ; UMCTL.2 = 0
            .endif
            .if     M$30>127        ; Overflow to integer?
M$40        .equ    M$30-128+CMOD   ; Yes, subtract 1.000000
C$3         .equ    8               ; UMCTL.3 = 1
            .else
M$40        .equ    M$30+CMOD       ; No, add fraction
C$3         .equ    0               ; UMCTL.3 = 0
            .endif
            .if     M$40>127        ; Overflow to integer?
M$50        .equ    M$40-128+CMOD   ; Yes, subtract 1.000000
C$4         .equ    10h             ; UMCTL.4 = 1
            .else
M$50        .equ    M$40+CMOD       ; No, add fraction
C$4         .equ    0               ; UMCTL.4 = 0
            .endif
            .if     M$50>127        ; Overflow to integer?
M$60        .equ    M$50-128+CMOD   ; Yes, subtract 1.000000
C$5         .equ    20h             ; UMCTL.5 = 1
            .else
M$60        .equ    M$50+CMOD       ; No, add fraction
C$5         .equ    0               ; UMCTL.5 = 0
            .endif
```

```
        .if     M$60>127            ; Overflow to integer?
M$70    .equ    M$60-128+CMOD       ; Yes, subtract 1.000000
C$6     .equ    40h                 ; UMCTL.6 = 1
        .else
M$70    .equ    M$60+CMOD           ; No, add fraction
C$6     .equ    0                   ; UMCTL.6 = 0
        .endif
        .if     M$70>127            ; Overflow to integer?
C$7     .equ    80h                 ; UMCTL.7 = 1
        .else
C$7     .equ    0                   ; UMCTL.7 = 0
        .endif
CUMCTL  .equ    C$7+C$6+C$5+C$4+C$3+C$2+C$1+C$0 ; Add bits
        .endm
```

6.10 The 8-Bit Interval Timer/Counter

6.10.1 Introduction

The 8-Bit Interval Timer/Counter peripheral is included in all members of the MSP430x3xx family. This timer/counter — its block diagram is shown in Figure 6–86 — can work, like its name suggests, in two different modes: the timer mode and the counter mode. This section describes software routines usable for the UART mode (SCI, RS232) that use the timer mode of the 8-Bit Timer/Counter. The software examples shown in the subsequent sections adapt themselves to the needs defined by the user (number of data bits, number of stop bits, baud rate, error detection and handling, clock frequency, and so on). This self-adaptation is accomplished through the use of the *conditional assembly* feature of the MSP430 assembler.

The hardware of the 8-Bit Interval Timer/Counter module supports the receive and transmit of UART data on a bit basis: one data bit is received or transmitted between two interrupts, not a complete frame consisting of a start bit, data bits, a parity bit and stop bits. This means that the interrupt overhead is relatively large due to the interrupt request after each received or transmitted data bit. On the other hand, the advantage is the complete flexibility of the data format — only software defines the number and meaning of the transferred bits. Any protocol is possible.

Figure 6–86 shows the block diagram of the complete MSP430 8-Bit Interval Timer/Counter module.

The 8-Bit Interval Timer/Counter module allows only the half duplex mode. This means that the module can receive data or it can transmit data, but not receive and transmit data simultaneously. The user software must therefore determine which mode should be active. In the following software examples, this is accomplished by the initialization subroutines.
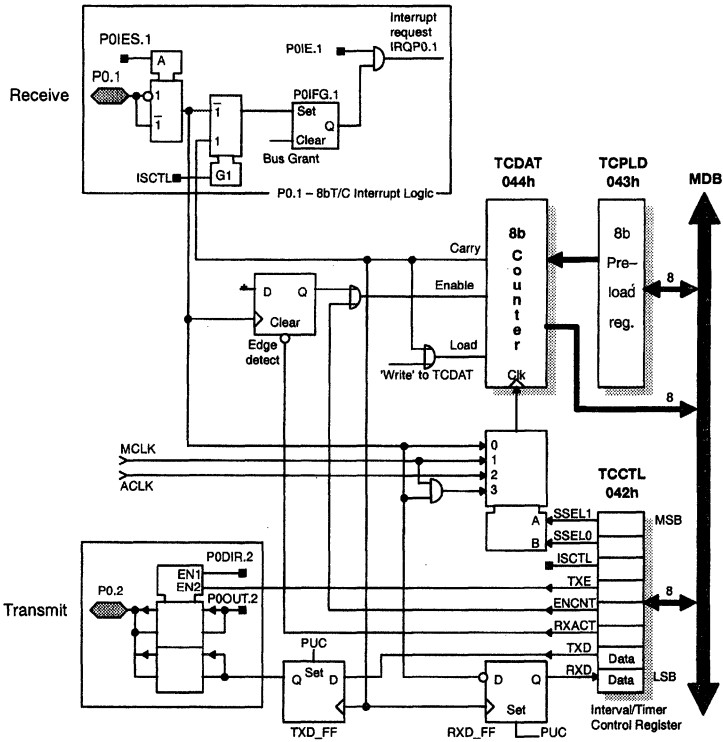
Figure 6–86. MSP430 8-Bit Interval Timer/Counter Module Hardware

6.10.1.1 Definitions Used With the Application Examples

The abbreviations used for the hardware definitions are consistent with the *MSP430 Architecture User's Guide*.

```
; HARDWARE DEFINITIONS
; 8-BIT TIMER/COUNTER
;
TCCTL    .equ    042h        ; T/C Control Register
RXD      .equ    001h        ; Receive signal at P0.1
TXD      .equ    002h        ; Next data bit for transmission
```

```
RXACT   .equ    004h        ; 1: detect start bit  0: off, reset FF
ENCNT   .equ    008h        ; Counter TCDAT enabled
TXE     .equ    010h        ; 1: TXD to P0.2     0: P0OUT.2 to P0.2
ISCTL   .equ    020h        ; Intrpt source:  0: P0.1  1: Carry TCDAT
SSEL0   .equ    040h        ; Clock source.   0: P0.1
SSEL1   .equ    080h        ; 1: MCLK  2: ACLK  3: P0.1 .and. MCLK
;
TCPLD   .equ    043h        ; T/C 8-Bit Pre-load Register
;
TCDAT   .equ    044h        ; T/C 8-Bit Counter
;
; OTHER DEFINITIONS
;
IE1     .equ    0           ; Interrupt Enable Register 1
P0IE1   .equ    8           ; P0.1 Interrupt Enable Bit (RCV)
P0IES   .equ    014h        ; P0 Interrupt Edge Select Register
;
SCG1    .equ    080h        ; Low Power Mode bit 1
SCG0    .equ    040h        ; Low Power Mode bit 0
CPUoff  .equ    010h        ; Switches CPU off
GIE     .equ    008h        ; General Interrupt Enable Bit
```

6.10.1.2 Attributes of a UART Implemented with the 8-Bit Timer/Counter

A short overview to the UART mode of the 8-Bit Timer/Counter module appears below:

❑ Half duplex mode — either transmit or receive mode is possible, but not both simultaneously.

❑ Any data length and format is possible. This is due to the software controlled data sequence.

❑ Error detection made by software:

■ Frame error — The stop bits have space potential or the start bit has mark potential in its middle

■ Parity error — Parity is enabled and the parity bit has the wrong value.

■ Overrun error — The next character is read in before the last one is read out by the software. This is not possible with the given software.

❑ Baud rate generation possible from the MCLK (500 kHz through 3.3 MHz) and from the ACLK signal (32,768 kHz crystal).

❑ Interrupt-driven transmit and receive functions.

❑ One interrupt vector for transmit and receive mode. Mode selection is made by software.

❑ Full functionality also during LPM3 (with ACLK only)

❑ Restricted baud rate range due to the length of the 8-Bit counter register TCDAT

❑ One full bit length (1/baud rate) is available for the read out or modification of the data. The time window for the reception and transmission of data is significantly enlarged compared to a pure software solution.

6.10.1.3 The Data Format

The data format used with the software examples is the RS232 format. Figure 6–87 shows how this format is seen at the MSP430 ports (P0.1 for receive and P0.2 for transmit) and Figure 6–88 shows how it is defined for the transmission line between the transmitter and the receiver.

The data format used with the Figures 6–87 and 6–88 is:

❑ Seven data bits. The least significant bit follows the start bit

❑ Parity enabled. The parity bit follows the most significant bit of the data

❑ No address bit. This is the normal case

❑ Two stop bits
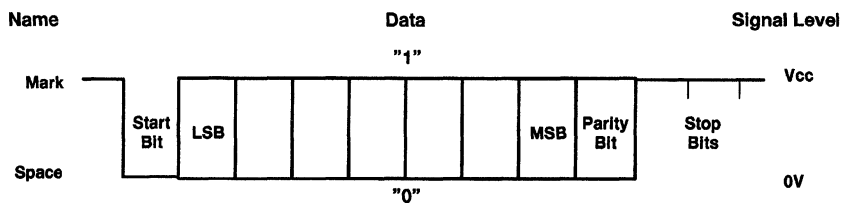
Figure 6–87. RS232 Format (Levels at the MSP430)

The signal on the transmission line has the inverted state as seen at the MSP430 ports and different voltage potentials. Figure 6–88 shows this.
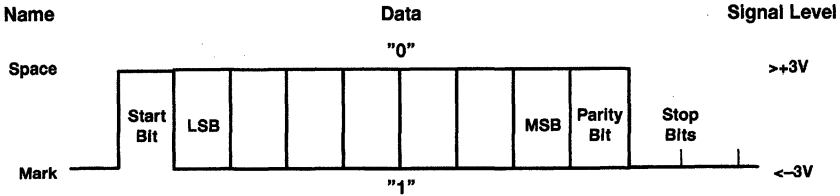
Figure 6–88. The RS232 Format (Levels on the Transmission Line)

6.10.2 Function of the UART Hardware

6.10.2.1 The Hardware Registers

The 8-Bit Timer/Counter module is controlled by one control register and two data registers. All are 8-bit registers and should therefore be accessed only with byte instructions. Figure 6–89 and Table 6–35 show an overview of these three registers, including the names, assembler mnemonics, hardware addresses, and the initial states. The detailed function of the control bits is described in the *MSP430 Architecture Guide and Module Library*.

Note:

When a write access to the Counter Register TCDAT is performed, then the information stored in the Preload Register TCPLD is loaded to TCDAT — and not the data addressed by the instruction.

The data contained in TCDAT can be read at address 044h.

Table 6–35.   UART Hardware Registers

REGISTER NAME	MNEMONIC	ACCESS	ADDRESS	INITIAL STATE
T/C Control Register	TCCTL	Read/Write	042h	Reset
T/C Preload Register	TCPLD	Read/Write	043h	Unchanged
T/C Counter Register	TCDAT	Read Only	044h	Unchanged

	7							0
TCCTL 042h	SSEL1	SSEL0	ISCTL	TXE	ENCNT	RXACT	TXD	RXD
	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	r(–1)

	7							0
TCPLD 043h								
	rw	rw	rw	rw	rw	rw	rw	rw

	7							0
TCDAT 044h								
	r(w)	r(w)	r(w)	r(w)	r(w)	r(w)	r(w)	r(w)
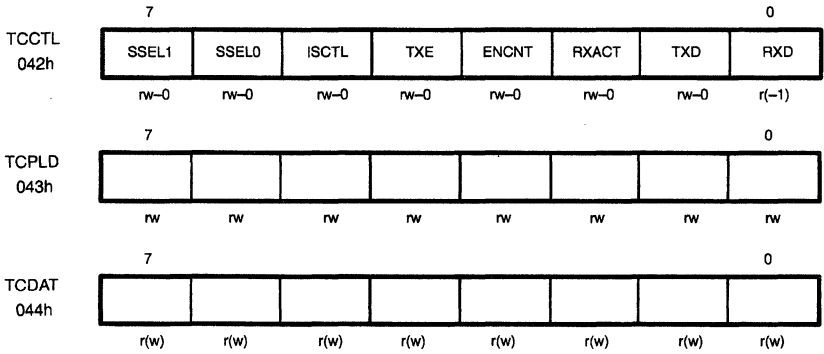
Figure 6–89. UART Hardware Registers

6.10.2.2 The Transmit Mode

If the 8-Bit Timer/Counter module is switched to the transmit mode — done by the initializing software of the module — then the hardware of figure 6–86 works as shown in Figure 6–90. Active data lines are drawn solid, nonactive data paths are drawn in gray color. The MCLK is selected for the UART timing.
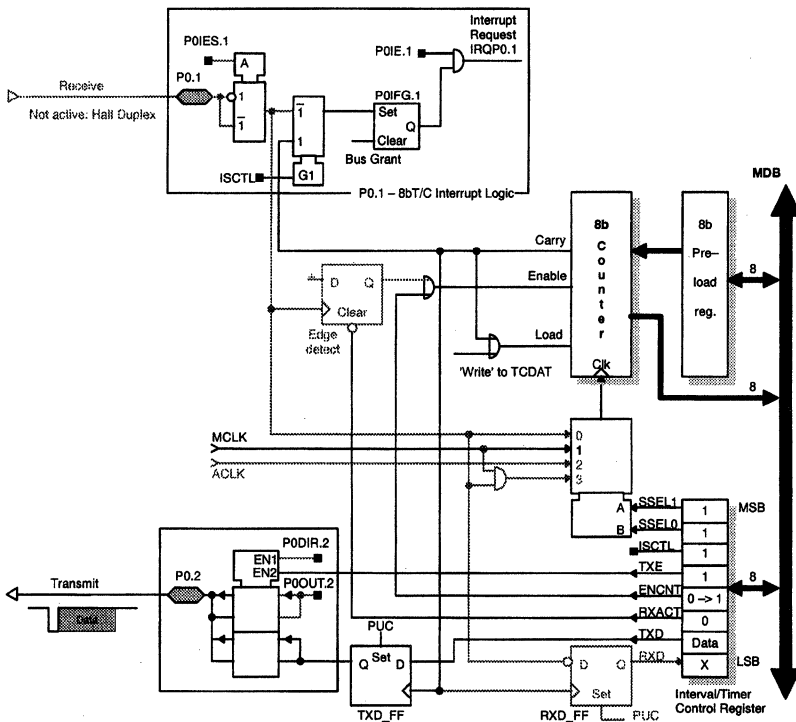
Figure 6–90. The 8-Bit Timer/Counter Transmit Mode

Initialization for the transmit mode is done by the subroutine TXINIT. The main steps for the transmission of a character are:

❑ Loading of the data word RTDATA with the character to be transmitted, including the address bit information (if defined)

❑ Initializing of the 8-Bit Timer/Counter and the RAM bytes RTERR and RTSTAT

   ■ Selecting of the clock frequency for the counter TCDAT (MCLK or ACLK) (SSELx bits)

   ■ Activation of the interrupt request by the carry of the 8-bit counter register TCDAT (ISCTL = 1)

   ■ Selecting of the TXD output data instead of the P0.2 output register data for the P0.2 pin (TXE = 1)

   ■ Setting of the TXD bit to mark (1) (TXD = 1). This value is transferred to the TXD output with the first counter interrupt. It guarantees a leading mark signal of at least one bit time.

   ■ Enabling of the 8-Bit Timer/Counter: the counter starts with the selected clock (ENCNT = 1)

   ■ Loading of the counter with one half of a bit time. After this time interval, the TXD output is set to mark (1) if not yet set

   ■ Loading of the pre-load register TCPLD with a full bit time interval (1/baud rate). This time interval is used for the leading mark before the start bit

   ■ Loading of the transmit status byte RTSTAT with the status for the start bit

   ■ Loading of the error byte RTERR with a start value (0 resp. 1) that delivers the correct parity bit of the complete character

   ■ Enabling of the interrupt for the 8-Bit Timer/Counter. Interrupt is requested now approximately after each time interval 1/baud rate. This time can change from bit to bit. See Section 6.10.3 *Baud Rate Generation and Correction.*

❑ Loading of the TXD bit during the interrupt handler with the information of the next but one bit to be output (start bit, data bits, address bit, parity bit, stop bits)

❑ Sampling of the information for the parity bit, if parity is enabled.

❑ Output of the nondata signals (start bit, parity bit, stop bits) dependent on the selected format

□ Turning off of the hardware after the complete output of a character, to save energy.
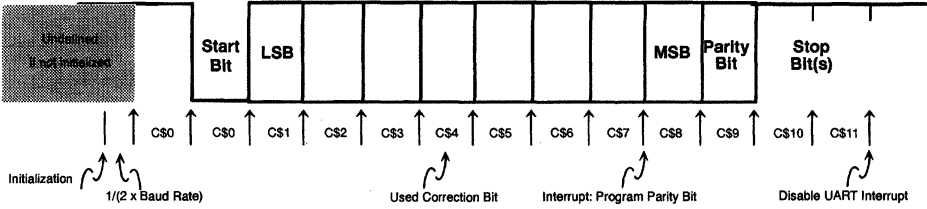
Figure 6–91. Interrupt Timing for the Transmit Mode

6.10.2.3 The Receive Mode

If the 8-Bit Timer/Counter module is switched to the receive mode — done by the initializing software of the module — then the hardware of Figure 6–86 works like shown in Figure 6–92. As with Figure 6–90, active data lines are drawn solid, nonactive data paths are drawn in gray color. The ACLK is used for the UART timing.
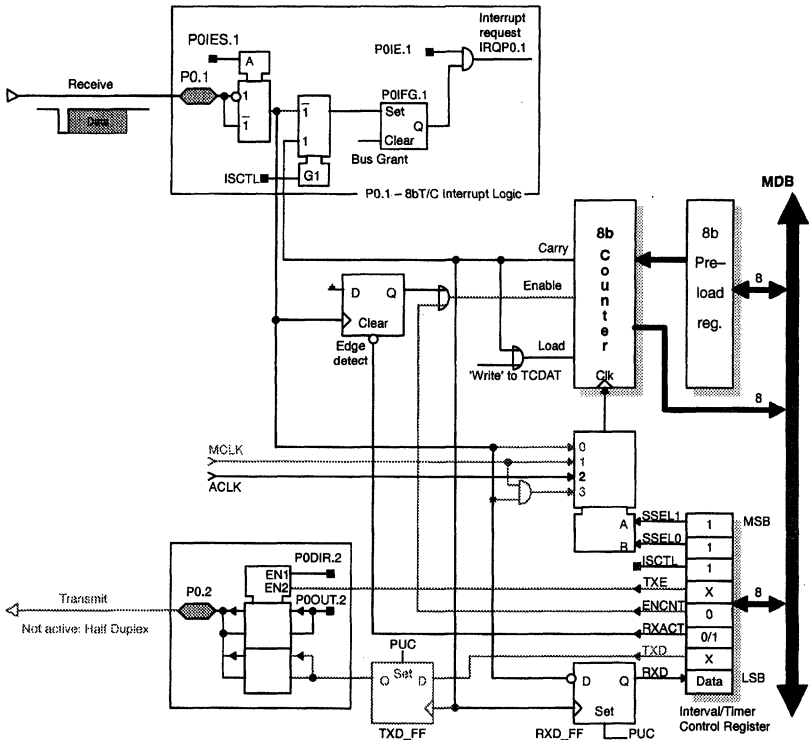
Figure 6–92. The 8–Bit Timer/Counter in Receive Mode

Initialization for the receive mode is done by the subroutine RCINIT. The main steps for the reception of a character are:

❑ Initializing of the 8-Bit Timer/Counter and the RAM bytes RTERR and RTSTAT:

   ■ Selecting the clock frequency for the counter (MCLK or ACLK) (SSELx bits)

   ■ Activation of the interrupt request by the carry of the 8-bit counter Register TCDAT (ISCTL = 1)

   ■ Reset of the edge-detect flip-flop (RXACT = 0)

   ■ Preparing of the 8-Bit Timer/Counter to start with the next negative transition of the P0.1 input signal from mark to space (1 to 0). The counter starts with the selected clock signal (ACLK or MCLK) after the next negative transition. (P0IES.1 = 1)

   ■ Loading of the counter with one half of a bit time. (If an input signal change at P0.1 occurs from mark to space, then after this time interval an interrupt is requested and the start signal is checked in its middle if it is still low (0).)

   ■ Loading of the pre-load Register TCPLD with a full bit time interval (1/baud rate). This time interval is used for the test in the middle of the LSB

   ■ Loading of the receive status byte RTSTAT with the status for the start bit

   ■ Loading of the error byte RTERR with a start value that delivers 0 if the parity of the complete character is correct

   ■ Enabling of the interrupt for the 8-Bit Timer/Counter. Interrupt is requested now approximately after the time interval 1/baud rate. This time changes slightly from bit to bit. See Section 6.10.3 *Baud Rate Generation and Correction.*

   ■ Setting the data word RTDATA to 0.

   ■ Activation of the edge-detect flip–flop: it detects the negative edge of the start bit and starts the counter (RXACT = 1).

   ■ Enabling of the UART interrupt (P0IE1 = 1).

❑ Reading of the RXD bit during the interrupt handler with the information of all bits (start bit, data bits, address bit, parity bit, stop bits). The read information is shifted into the data word RTDATA.

❑ Sampling of the information for the parity bit, if parity is enabled.

❑ Check of the nondata signals (start bit, parity bit, stop bits), dependent on the selected format

❑ Setting of the error bits TCPE and TCFE dependent on the bit check. If no error occurred, the error byte RTERR contains 0

❑ Turning off of the timer/counter hardware after the complete reception of a character: interrupt and clock are switched off.
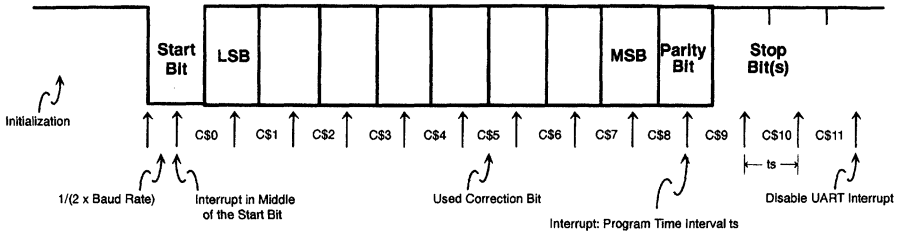
Figure 6–93. Interrupt Timing for the Receive Mode

6.10.3 The Baud Rate Generation and Correction

The short counter register TCDAT of the 8-bit timer/counter allows the use of the MCLK for only very few baud rates. For all other baud rates, the maximum value 255 for the quotient MCLK/baud rate is exceeded. Therefore, the use of the ACLK (32,768 Hz) is necessary for most of the usual baud rates. But the use of the ACLK frequency causes another problem:

Generating the desired baud rate from a relatively high frequency (1 MHz to 5 MHz) is a simple task. The resulting baud rate error is small due to the large integer part of the quotient compared to the truncated fractional part. This changes completely if the time base is a crystal of only 32 kHz. Then the error due to the truncated fractional part of the quotient grows large and leads to the loss of synchrony at the trailing bits of the frame. The MSP430 UART software therefore uses a correction to keep the baud rate error small. The baud rate correction calculates correction information (9 to 13 bits, dependent on the frame length) as to how to correct the baud rate of the received or transmitted UART signal. The calculated bits C$0 to C$12 define how the predivider information contained in the baud rate registers TCPLD and TCDAT is used:

❏ C$x = 0 — the calculated time interval is used as is.

❏ C$x = 1 — the calculated time interval is prolonged by one timer period (MCLK or ACLK) and used with this value.

The value C$0 is used for the start bit, the value C$1 for the LSB of the data, and so on. See Figure 6–94 for an explanation.

Example 6–63. Baud Rate Generation

A baud rate of 2400 baud is needed from a crystal frequency of 32,768 Hz. The frame length used is the minimum length: start bit, seven data bits, no address bit, no parity, and one stop bit. This results in a frame length of nine bits. The use of the ACLK is necessary due to two reasons:

❏ The UART also needs to run during the low power mode 3, when the MCLK is not available.

❏ The maximum MCLK frequency would be 255 × 2400 = 612 kHz. This frequency is too low for most of the applications (and cannot be guaranteed for the system clock generator).

With only the ACLK available, the theoretical division factor UBR — the truncated value is the base for one of the two contents of the Pre–Load Register TCPLD — is:

$$UBR = \frac{32768}{2400} = 13.653333$$

This means — because the register counts upward — that the pre-load register TCPLD normally contains −13 (0F3h). To get a rough value for the 9-bit baud rate correction C$0 to C$8, the fractional part (0.653333) of the above division is multiplied by 9 (the number of calculated bits for the baud rate correction):

$$Number\ of\ Ones\ =\ 0.653333 \times 9\ =\ 5.88000$$

The rounded result, 6, is the number of *1s* to be used with the baud rate correction. The resulting, corrected, baud rate with the 6 1s of the baud rate correction is (6 bits have a length of 14 ACLK periods, 3 have a length of 13 ACLK periods):

$$BaudRate\ =\ \frac{32768}{\left(\dfrac{6 \times 14 + 3 \times 13}{9}\right)}\ =\ 2397.6585$$

This results in an average baud rate error of:

$$Baud\ Rate\ Error\ =\ \frac{2397.6585 - 2400}{2400} \times 100\ =\ -0.0975\%$$

To get the bit sequence for the baud rate correction that fits best, the following algorithm can be used. The fractional part of the theoretical division factor UBR is summed nine times and if a carry to the integer part occurs, the current C$x bit is set. Otherwise, it is cleared. An example for the calculation of 9 bits with the above fraction (0.653333) follows:

Fraction Addition	Carry to next Integer	Correction Bits	
0.653333 + 0.653333 = 1.306667	Yes	C$0	1
1.306667 + 0.653333 = 1.959999	No	C$1	0
1.959999 + 0.653333 = 2.613332	Yes	C$2	1
2.613332 + 0.653333 = 3.266667	Yes	C$3	1
3.266667 + 0.653333 = 3.919999	No	C$4	0
3.919999 + 0.653333 = 4.573331	Yes	C$5	1
4.573331 + 0.653333 = 5.226664	Yes	C$6	1
5.226664 + 0.653333 = 5.879997	No	C$7	0
5.879997 + 0.653333 = 6.533333	Yes	C$8	1

The result of the calculated bits C$8...C$0 (1 0110 1101b) is 16Dh with six ones. The software example contains a macro loop (starting at label MODTAB) that uses the algorithm shown above and calculates, for every combination of the UART clock and the desired baud rate, the optimum value for the baud rate correction. For the above example (9 bit frame length), the macro also determines 16Dh with its six ones.

Example 6–64. 2400 Baud From ACLK

Figure 6–94 gives an example for a baud rate of 2400 baud generated with the ACLK frequency (32,768 Hz). The data format for figure 6–94 is:

Eight data bits, parity enabled, no address bit, and two stop bits. Figure 6–94 shows three different frames:

❏ The upper frame is the correct one with a bit length of 13.65333 ACLK cycles (32,768/2400 = 13.65333)

❏ The middle frame uses a rough estimation with 14 ACLK cycles for the bit length

❏ The lower frame uses a corrected frame with the best fit (C$11...C$0 = 0B6Dh) for the baud rate correction.

It can be seen that the approximation with 14 ACLK cycles accumulates an error of more than 0.3 bit length after the second stop bit. The error of the corrected frame is only 0.001 bit length. The error of the crystal clock is not yet included, and it adds to the above errors.
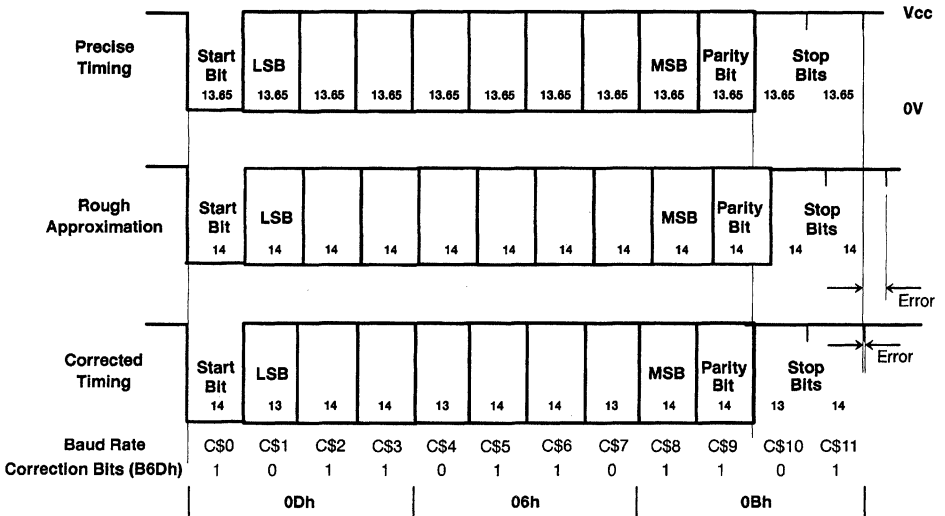
Figure 6–94. Baud Rate Correction

Tables 6–36 and 6–37 contain the average errors (full frame with maximum length, 13 bits) for the normally used baud rates resulting from the described baud rate generation. The software examples contain a looped macro. It calculates — dependent on the frame length used — for all the bits the optimum length.

6.10.3.1 Baud Rate Generation With the MCLK

Table 6–36 shows the optimum values for the 8-bit counter register TCDAT. The UART clock is the MCLK (1,048 MHz). The crystal error is not included. The mean error is calculated for a medium frame length of eleven bits: start bit, eight data bits, parity enabled, and one stop bit. Table 6–36 contains the following columns:

❏ **Baud Rate** — The baud rate for the data exchange (transmit and receive use the same baud rate)

❏ **Division Factor** — The quotient UARTCLK/baud rate. It indicates the number of MCLK cycles for a data bit

❏ **8-Bit Counter Register** — The truncated 8-bit hexadecimal result of the division factor (UARTCLK/baud rate). The value that is loaded into the hardware register TCDAT is (100h – table value). This is due to the upward count of the 8-bit counter.

❑ **Baud Rate Correction** — The 13-bit result that fits best for the baud rate correction. It is calculated by the software macro starting at label MOD-TAB. If frames with less than 13 bits are used, then the MSBs of this number are omitted.

❑ **Used Fraction** — The number of 1s in the baud rate correction sequence divided by eleven (the frame length used for the calculation). It is an approximation of the truncated fractional part of the division factor.

❑ **Mean Error** — The resulting error of a complete character caused by the approximation of the division factor

The length of the 8-bit counter register allows only a very limited range for the baud rate. An MCLK frequency of 1.048 MHz is assumed. For other frequencies, the baud rates change accordingly (e.g. for 2.096 MHz the usable baud rates are 9600 and 19200 baud). The reasons for this restriction are:

❑ From 110 baud to 2400 baud, the 8-bit counter register is too small to hold the necessary number for the result of the division MCLK/baud rate: the number contained in the column *8-Bit Counter Register* is greater than 0FFh.

❑ Beginning at 9600 baud, the CPU cycles between two UART interrupts are too few for correct handling (e.g. only 54 CPU cycles @ 1.048 MHz for 19200 baud). See Section 4.4. The maximum baud rate depends strongly on the amount of interrupt activity due to the other peripherals.

Note:

The assembler outputs an error message if the resulting value for the TCDAT register is greater than 255. This is an indication of a baud rate that is too low.

Note:

Baud rates that result in TCDAT register values lower than 100 make strictly *real time processing* rules necessary. Interrupt handlers must be as short as possible and interruptible. See Section 4.4 for hints how to speed-up the UART.

Table 6–36. Baud Rate Register TCDAT Contents (MCLK = 1,048 MHz)

BAUD RATE	DIVISION FACTOR	8-BIT COUNTER REGISTER	BAUD RATE CORRECTION	FRACTION USED	MEAN ERROR (%)
110	9532.51	253Ch	–		
300	3495.25	0DA7h	–		
600	1747.63	06D3h	–		
1200	873.81	0369h	–		
2400	436.91	01B4h	–		
4800	218.45	00DAh	14AAh	0.4545	–0.002
9600	109.23	006Dh	1088h	0.1818	+0.044
19200	54.61	0036h	–		
38400	27.31	001Bh	–		

6.10.3.2 Baud Rate Generation With the ACLK

With the relatively low ACLK frequency (32,768 Hz), the baud rate correction becomes much more important compared to the normally high MCLK frequency used for the UART timing. Table 6–37 shows the optimum values for the counter register TCDAT and the correction values for commonly used baud rates generated with the ACLK (32,768 Hz). The table values are calculated by the macro starting at the label MODTAB. The crystal is assumed to be without frequency error. The meaning of the table columns is explained in Section 6.10.3.1. As for Table 6–36, the mean error is calculated for a medium frame length of eleven bits: start bit, eight data bits, parity enabled, and one stop bit.

Table 6–37. Baud Rate Register TCDAT Contents (ACLK = 32,768 Hz)

BAUD RATE	DIVISION FACTOR	8-BIT COUNTER REGISTER	BAUD RATE CORRECTION	FRACTION USED	MEAN ERROR (%)
110	297.8909	0129h	–		
300	109.2267	006Dh	1088h	0.1818	+0.04
600	54.6133	0036h	15DAh	0.6363	–0.04
1200	27.3067	001Bh	1124h	0.2727	+0.12
2400	13.6533	000Dh	1B6Dh	0.6363	+0.12
4800	6.8267	0006h	1BEFh	0.8181	+0.13
9600	3.4133	0003h	094Ah	0.3636	+1.46
19200	1.7067	–	–	–	–
38400	0.8533	–	–	–	–

6.10.4 Software Routines

The following sections show proven software routines for the UART mode of the 8-Bit Timer/Counter.

Note:

The program sequences for the initialization of the UART software are important. The example code should not be modified. See the subroutines TXINIT and RCINIT.

Note:

Any protocol is possible due to the software control for the data sequence. It is only necessary to adapt the two tables RTTAB and MODTAB of the two software examples that follow.

The software routines are shown for interrupt use only. It makes no sense to use the noninterrupt solution (polling) because the time intervals between two signal bits are relatively short — a 100% loading of the CPU would be the result. This is due to the bit orientation of the 8-Bit Timer/Counter hardware.

The initialization subroutine INITSR and the RAM initialization subroutine RAMCLR are explained in detail in section *The Timer_A*, paragraph *Common Initialization Routine*.

6.10.4.1 MCLK Used for UART Clock

The following example is for use when MCLK used for the generation of the UART clock. For high baud rates — higher than 9600 baud @ 1 MHz — dedicated CPU registers may be necessary to lower the interrupt overhead. The time for the saving and restoring of the register is not necessary. See Section 6.10.4.4.

Example 6–65. Half Duplex UART with Interrupt

Half duplex UART software using the interrupt of the 8-Bit Timer/Counter is shown below. The software is designed for:

❑ Baud rate: 4800 baud

❑ The MCLK (1,048 MHz) is used for the UART clock

❑ The active mode of the CPU is used

❑ Seven data bits

❑ Parity enabled with even parity

❑ No address bit included

❑ One stop bit

❑ Reception of all characters (the error byte RTERR contains an error indication)

❑ UART signals like shown in figure 6–87 (mark = V_{CC}, space = V_{SS})

The following seven software switches and the value for the UARTCLK need to be defined for the UART operation (see also the examples in the software part). Functions that are not enabled, do not use memory space: the adjacent code is left out by *Conditional Assembly.*

UARTCLK   If the MCLK is used for the UART timing, then the MCLK frequency must be given here. Normally the MCLK is defined by multiplication of the crystal frequency with the FLL multiplier.

Baudr   Baud rate used [Hz]. For 1.048 MHz MCLK frequency, the range is from 4800 baud to 9600 baud. With special care, 19200 baud is also possible. The range of usable baud rates increases linearly with the MCLK frequency used.

CHARC   Number of data bits. The UART software allows 7 and 8 data bits, but the table structure of the software eases the adaptation to other bit counts.

ADDR   Inclusion of an address bit (1) or not (0). See the *MSP430 Architecture Guide* for an explanation of this feature.

PAR   Enables (1) or disables (0) a parity check. A parity error sets bit TCPE (RTERR.0).

PAREV   If parity is enabled (PAR = 1), even (1) or odd (0) parity is used for the data check.

STB   Defines the number of stop bits. Possible values are 1 or 2 stop bits

TCERRT   Defines the treatment of detected errors. If the received character is correct, the byte RTERR contains 0. The possible values for the switch TCERRT are:

TCERRT = 0:   the current, erroneous character is discarded and the receive function is initialized for a new start bit check. This means the software tries to find a valid start bit.

TCERRT = 1:   the error is indicated in byte RTERR, the reception of the current character continues.

Possible errors are:

TCPE (RTERR.0) = 1 — parity error. The sum of 1s contained in the data bits, the address bit, and the parity bit is not correct. It is not odd for odd parity *OR* even for even parity

TCFE (RTERR.1) = 1 — frame error. This means the middle of the start bit is high, or one of the stop bits is low. This error is normally caused by a software start inside of a character frame.

Transmit Mode: the data to be transmitted is loaded right-aligned into the RAM word RTDATA. The address bit — if enabled by ADDR = 1 — is included. No error is possible. Four examples for the data in RTDATA are shown in figure 6–95. The completion of the transmission is indicated by a value of (TX6–RTTAB) in the status byte RTSTAT. A relative number (TX6–RTTAB) is necessary due to the many possible data formats.

**7–Bit Data
No Address Bit**

15	7	0
0	0	7–Bit Data

**7–Bit Data
Address Bit**

15	7	0
0	ADR	7–Bit Data

**8–Bit Data
No Address Bit**

15	8 7	0
0	8–Bit Data	

**8–Bit Data
Address Bit**

15	8 7	0
0	ADR	8–Bit Data
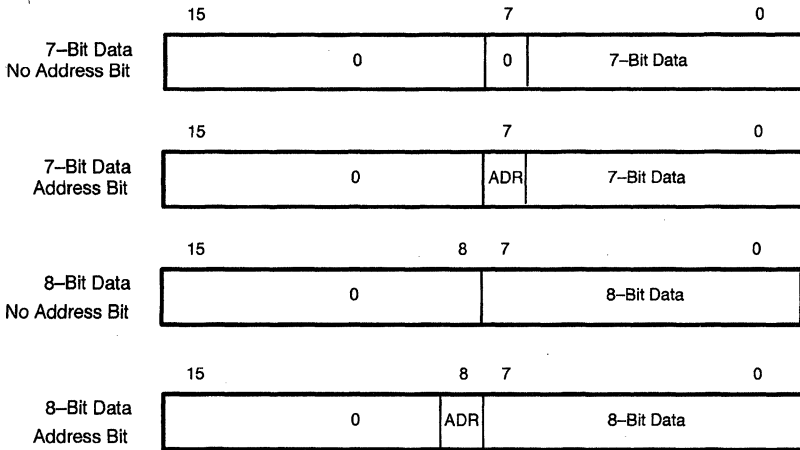
Figure 6–95. Transmitted Data Format

Receive Mode: the received data is loaded left-aligned into the RAM word RTDATA (see Figure 6–96). This means that, depending on the address bit and the number of data bits contained in the data word, a shift is necessary to get a single byte containing the received character. The input format used is necessary due to the address bit. The completion of the reception is indicated by a value of (RC6–RTTAB) in the status byte RTSTAT. A relative number is necessary due to the many possible data formats. If no error occurred, then the error byte RTERR contains 0, otherwise it contains the reason of the error in its LSBs:

❑ Bit TCPE (RTERR.0) is set: a parity error occurred

❑ Bit TCFE (RTERR.1) is set: a frame error occurred. This can be caused by a start bit having a mark signal (1) or a stop bit having a space signal (0).

	15	8	7	0
7–Bit Data No Address Bit	7–Bit Data	0	0	

	15	8	7	0
7–Bit Data Address Bit	ADR	7–Bit Data	0	

	15	8	7	0
8–Bit Data No Address Bit	8–Bit Data	0		

	15	8	7	0
8–Bit Data Address Bit	ADR	8–Bit Data	0	
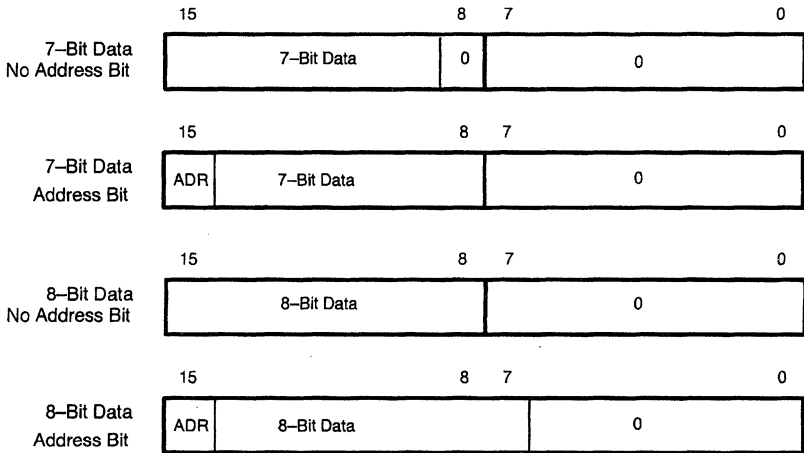
Figure 6–96. Received Data Format

```
; Definitions for the common part
;
STACK    .equ    0300h              ; Stack start address
FLLMPY   .equ    32                 ; FLL multiplier for 1,048MHz
;
; Definitions for the UART part
; Data format: 4800 Baud, even parity, 7 data bits, 1 stop bit
; MCLK for UART clock, also erroneous characters to input
; buffer
;
Baudr    .equ    4800               ; Baud rate is 4800 Baud
UARTCLK  .equ    FLLMPY*32768       ; MCLK is used for UARTCLK
CHARC    .equ    7                  ; Length:  7: 7 bits    8: 8 bits
ADDR     .equ    0                  ; Address bit: 1: yes   0: no
PAR      .equ    1                  ; Parity   0: disabled  1: enabled
PAREV    .equ    1                  ; Parity   0: odd       1: even
STB      .equ    1                  ; Stop bits: 1: one     2: two
TCERRT   .equ    1                  ; 0: error restart    1: indication
;
```

```
TCPE      .equ     1                      ; Parity error: RTERR.0 = 1
TCFE      .equ     2                      ; Frame error:  RTERR.1 = 1
CUBR      .equ     -(UARTCLK/Baudr)       ; Content 8-Bit Counter
;
          .even                           ; Word boundary
          .bss     RTDATA,2               ; Data for receive/transmit
          .bss     RTERR,1                ; Error byte
          .bss     RTSTAT,1               ; Status byte
;
          .text                           ; Software start address
;
INIT      MOV      #STACK,SP              ; Initialize Stack Pointer
          CALL     #INITSR                ; Init. FLL and RAM
          ...                             ; Proceed with initialization
          EINT                            ; Enable interrupts
;
MAINLOOP ...                              ; Mainloop starts here
          ...
;
; Prepare transmission of one character from RAM word RTDATA
; Info is contained right aligned in LSBs. No error possible
;
          MOV      #xxx,RTDATA            ; Character xxx to RTDATA
          CALL     #TXINIT                ; Initialize the transmit part
          ...                             ; Continue with background
                                          ; Check for completion:
          CMP.B    #TX6-RTTAB,RTSTAT      ; Character transmitted?
          JEQ      CHARTX                 ; Yes, prepare next one
          ...                             ; No, continue
;
; Prepare the reception of one character to RAM word RTDATA
; Info is contained left aligned in the LSBs. Errors in RTERR
;
          CALL     #RCINIT                ; Initialize the receive part
          ...                             ; Continue in background
                                          ; Check for completion:
```

```
          CMP.B     #RC6-RTTAB,RTSTAT ; One character received?
          JNE       NO_CHAR          ; No, continue
          TST.B     RTERR            ; Yes, error?
          JNZ       ERRHDL           ; Yes, check reason
          CLRC                       ; No, shift a 0 in MSB
          RRC       RTDATA           ; RTDATA+1 contains 7-bit data
          ...                        ; Process data in RTDATA+1
          BR        #MAINLOOP        ; Back to mainloop
; Common interrupt handler for transmit and receive functions.
; The carry of TCDAT is switched to the P0.1 interrupt request.
; Interrupt time interval of the 8-bit timer is: 1/Baud rate
; The single status byte RTSTAT contains the actual status:
;
; Idle:     RTSTAT = 0                        No UART activity
; Transmit: RTSTAT = 1...TX6-RTTAB-1          Active
;                    TX6-RTTAB                Character output
; Receive:  RTSTAT = RC-RTTAB...RC6-RTTAB-1   Active
;                    RC6-RTTAB                Char. received
;
TXRCINT PUSH      R5               ; Save R5
        MOV.B     RTSTAT,R5        ; Receive/transmit status
        MOV.B     RTTAB(R5),R5     ; Offset to handler address
        ADD       R5,PC            ; RTTAB+RTSTATx-RTTAB -> PC
RTTAB   .BYTE     RTSTAT0-RTTAB    ; Offset RTSTAT = 0 (inactive)
;
; Transmit states
;
TX      .BYTE     TXSTAT1-RTTAB    ; TX: Start bit
        .BYTE     TXSTAT2-RTTAB    ; TX: LSB
        .BYTE     TXSTAT2-RTTAB    ; TX: LSB+1
        .BYTE     TXSTAT2-RTTAB    ; TX: LSB+2
        .BYTE     TXSTAT2-RTTAB    ; TX: LSB+3
        .BYTE     TXSTAT2-RTTAB    ; TX: MSB-3
        .BYTE     TXSTAT2-RTTAB    ; TX: MSB-2
        .if       CHARC=8          ; Data length 7 or 8 bits?
        .BYTE     TXSTAT2-RTTAB    ; TX: MSB-1
```

```
        .endif
        .BYTE    TXSTAT2-RTTAB    ; TX: MSB
        .if      ADDR=1           ; Address bit?
        .BYTE    TXSTAT3-RTTAB    ; TX: Address bit
        .endif
        .if      PAR=1            ; Parity enabled?
        .BYTE    TXSTAT4-RTTAB    ; TX: Parity bit
        .endif
        .BYTE    TXSTAT5-RTTAB    ; TX: stop bit 1
        .if      STB=2            ; Two stop bits?
        .BYTE    TXSTAT5-RTTAB    ; TX: stop bit 2
        .endif
TX6     .BYTE    TXSTAT6-RTTAB    ; TX: Frame output completed
;
; Receive states: interrupt occurs in the middle of the bits
;
RC      .BYTE    RCSTAT1-RTTAB    ; RC: start bit
        .BYTE    RCSTAT2-RTTAB    ; RC: LSB
        .BYTE    RCSTAT2-RTTAB    ; RC: LSB+1
        .BYTE    RCSTAT2-RTTAB    ; RC: LSB+2
        .BYTE    RCSTAT2-RTTAB    ; RC: LSB+3
        .BYTE    RCSTAT2-RTTAB    ; RC: MSB-3
        .BYTE    RCSTAT2-RTTAB    ; RC: MSB-2
        .if      CHARC=8          ; Data length 7 or 8 bits?
        .BYTE    RCSTAT2-RTTAB    ; RC: MSB-1
        .endif
        .BYTE    RCSTAT2-RTTAB    ; RC: MSB
        .if      ADDR=1           ; Address bit?
        .BYTE    RCSTAT3-RTTAB    ; RC: Address bit
        .endif
        .if      PAR=1            ; Parity enabled?
        .BYTE    RCSTAT4-RTTAB    ; RC: Parity bit
        .endif
        .BYTE    RCSTAT5-RTTAB    ; RC: stop bit 1, parity check
        .if      STB=2            ; Two stop bits?
        .BYTE    RCSTAT6-RTTAB    ; RC: stop bit 2
```

```
        .endif
RC6     .BYTE   TXSTAT6-RTTAB      ; RC: Frame received
;
; Transmit software part. Interrupt after output bit.
; The bit length and data of the next but one bit is defined
;
TXSTAT1 BIC.B   #TXD,&TCCTL        ; Start bit: output space (0)
        JMP     TXRET              ; To common interrupt return
;
TXSTAT3 .equ    $                  ; Address bit (if defined)
TXSTAT2 RRA     RTDATA             ; Data bit: next one to carry
        .JC     TX1                ; Data is 1
TX0     BIC.B   #TXD,&TCCTL        ; Output data 0: reset TXD
        JMP     TXRET
;
TX1     .equ    $                  ; Output 1 with parity count
        .if     PAR=1              ; Parity enabled?
        XOR.B   #1,RTERR           ; Toggle LSB for parity
        .endif
TXSTAT5 .equ    $                  ; Stop bit: output 1 w/o parity
        BIS.B   #TXD,&TCCTL        ; Data is 1: set TXD
;
; Tasks are made, the next but one bit length is loaded to the
; pre-load register TCPLD. The bit length for the current bit was
; loaded with the current interrupt.
;
TXRET   MOV.B   RTSTAT,R5          ; Transmit status to R5
        MOV.B   MODTAB-1(R5),&TCPLD ; Next but one bit length
        JMP     RTRET              ; To common RETI part
;
        .if     PAR=1              ; Parity enabled?
TXSTAT4 BIT.B   #1,RTERR           ; Yes, check parity value
        JNZ     TX1                ; Output mark (1).  TCPE = 0
        JMP     TX0                ; Output space (0). TCPE = 0
        .endif
;
```

```
; One full character is received or transmitted. The UART
; hardware is switched off. The status for a completed
; character is:
; Receive Mode:  RC6-RTTAB
; Transmit Mode: TX6-RTTAB
;
TXSTAT6  BIC.B    #P0IE1,&IE1        ; Disable TCDAT carry interrupt
         BIC.B    #RXACT+ENCNT,&TCCTL ; Stop T/C, conserve power
         JMP      RTSTAT0            ; To RETI w/o status change
;
; Receive software part. Interrupt occurs in the middle of the
; bit. The bit length of the next but one bit is defined
;
RCSTAT1  BIT.B    #RXD,&TCCTL        ; Check middle of start bit
         JZ       RCRET             ; Start bit is 0: ok
         .if      TCERRT=1          ; Error, indication wished?
         BIS.B    #TCFE,RTERR        ; Frame error bit TCFE set
         .endif
         JMP      RCERR             ; Start bit is 1; error
;
RCSTAT4  .equ     $                  ; Parity bit is received normally
RCSTAT3  .equ     $                  ; Address bit too
RCSTAT2  BIT.B    #RXD,&TCCTL        ; Data bits: info to carry
         RRC      RTDATA            ; Shift data into MSB
         .if      PAR=1             ; Parity enabled?
         JN       RC1               ; Data is a 1: adjust parity info
         JMP      RCRET             ; Data is a 0: all done
RC1      XOR.B    #1,RTERR           ; Yes, adjust odd/even info
         .endif
;
; Tasks are made, the next but one bit length is loaded to the
; pre-load register TCPLD. The bit length for the next bit was
; loaded with the current interrupt.
;
RCRET    MOV.B    RTSTAT,R5          ; Transmit status to R5. Length
         MOV.B    MODTAB-(RC-TX)(R5),&TCPLD ; next but one bit
```

```
RTRET     INC.B    RTSTAT          ; To next receive status
RTSTAT0   POP      R5              ; Restore R5
          RETI
;
; Stop bit handling: RXD must be high. Parity is checked also:
; Parity bit RTERR.0 (TCPE) must be 0
;
RCSTAT5   .equ     $               ; Parity check during stop bit 1
          .if      PAR=1           ; Parity enabled?
          RLA      RTDATA          ; Shift out parity bit
          .if      TCERRT=0        ; Restart for error?
          BIT.B    #1,RTERR        ; Yes, check parity value TCPE
          JNZ      RCERR           ; Not 0: error. TCPE stays 1
          .endif
          .endif
RCSTAT6   BIT.B    #RXD,&TCCTL     ; Stop bit (1 or 2) high?
          JNZ      RCRET           ; Yes, Parity and stop bits ok
          .if      TCERRT=1        ; No, Error indication wished?
          BIS.B    #TCFE,RTERR     ; Yes, set frame error bit
          JMP      RCRET           ; Continue with frame
          .endif                   ; No, to error handler RCERR
;
; Error handling: two different ways can be selected:
; TCERRT = 0: restart, start bit check. Current char. is discarded
; TCERRT = 1: error indication in RTERR. Reception continues.
;
          .if      TCERRT=0        ; Error indication wished?
RCERR     BIC.B    #POIE1,&IE1     ; No, intrpt disabled: UART off
          EINT                     ; Allow nesting
          CALL     #RCINIT         ; Restart receive task
          JMP      RTSTAT0
          .else
RCERR     .equ     RCRET           ; Yes, continue
          .endif
;
; Table MODTAB contains the calculated bit lengths that fit
```

```
; best. Sequence: start bit, LSB...MSB, (address bit),
; (parity), stop bits + one bit more for the turn-off
; Only the necessary bytes - dependent on the frame length -
; are included. All bits are calculated individually.
; Resolution of the calculation is 10 bits
;
MODTAB    .equ      $                    ; Calculate fraction (UARTCLK/Baudr)
CMOD      .equ      (((1024*UARTCLK)/Baudr)-1024*(UARTCLK/Baudr))
          .eval     CMOD,M$00
          .mnolist
          .loop     9+(ADDR=1)+(PAR=1)+(CHARC=8)+(STB=2)+1 ; Bit #
          .eval     CMOD+M$00,M$00
          .if       M$00>1023            ; Carry to integer?
          .eval     M$00-1024,M$00       ; Yes
          .mlist
          .byte     CUBR-1               ; C$x = 1: Bit one cycle longer
          .mnolist
          .else
          .mlist
          .byte     CUBR                 ; C$x = 0: Bit normal length
          .mnolist
          .endif
          .endloop
          .even                          ; To word boundary
;
; Subroutines
; The subroutine prepares the 8-Bit Timer/Counter hardware to
; transmit data. Initialize control byte TCCTL:
; SSEL1/SSEL0: 1/0 for MCLK frequency
; ISCTL:       1   Carry of TCDAT register causes P0.1 intrpt
; TXE:         1   Output P0.2 to TXD, disable P0OUT.2
; TXD          1   Set TXD (P0.2) to high (mark)
; ENCNT:       1   Enable clock to the TCDAT register
;
TXINIT    MOV.B     #SSEL1+ISCTL+TXE+TXD+ENCNT,&TCCTL
          MOV.B     #TX-RTTAB,RTSTAT ; Transmit status for start bit
```

```
            JMP      RTINIT            ; To common part
;
; The subroutine prepares the 8-Bit Timer/Counter hardware to
; receive data. Initialize control byte TCCTL:
; SSEL1/SSEL0: 1/0 for MCLK frequency
; ISCTL:      1   Carry of TCDAT register causes P0.1 intrpt
; TXE:        1   Enable output buffer for P0.2
; TXD:        1   Set TXD (P0.2) to high (mark)
; RXACT:      0   Reset Edge Detect Flip-Flop
;
RCINIT  MOV.B    #SSEL1+ISCTL+TXD+TXE,&TCCTL
        MOV.B    #RC-RTTAB,RTSTAT  ; Receive status start bit
        CLR      RTDATA            ; Clear data word
        BIS.B    #2,&P0IES         ; Neg. edge detect for P0.1
;
; Common part for transmit and receive. The parity bit RTERR.0
; is initialized in a way, that always zero is returned, if
; the parity is ok.
;
RTINIT  MOV.B    #CUBR/2,&TCPLD    ; Half bit time to 1st intrpt
        MOV.B    #0,&TCDAT         ; Load half bit time to TCDAT
        MOV.B    MODTAB,&TCPLD     ; Bit time for 1st bit
        .if      (PAR=1)&(PAREV=0) ; Odd Parity enabled?
        MOV.B    #1,RTERR          ; Odd parity: RTERR.0 = 1
        .else
        MOV.B    #0,RTERR          ; No parity .or. even parity
        .endif
        BIS.B    #P0IE1,&IE1       ; TCDAT carry intrpt enabled
        BIS.B    #RXACT,&TCCTL     ; Receive: enable edge detect.
        RET
; Interrupt Vectors
;
        .sect    "SCIVEC",0FFF8h   ; HW/SW UART Vectors
        .word    TXRCINT           ; Common TX/RC Vector
        .sect    "INITVEC",0FFFEh  ; Reset Vector
        .word    INIT              ; Program Start Address
```

6.10.4.2 ACLK Used for the UART Clock

With the ACLK used for the UART clock, two different methods are possible.

❑ ACLK used with the active mode — the only difference to the last section is the use of the ACLK instead of the MCLK.

❑ ACLK used with the low power mode 3 — The CPU is switched off normally (LPM3) but the UART activity continues. This method is necessary for low power applications.

The two different methods are described in the next two sections.

6.10.4.2.1 ACLK With the Active Mode

The ACLK can be used for the UART clock in very much the same way as the MCLK (see Section 6.10.4.1 for details). The use of the ACLK may be necessary if the needed baud rate is too low for the MCLK frequency in use. For example, with an MCLK of 1.048 MHz, the lowest (usual) baud rate is 4800 baud.

To use the ACLK with the active mode, it is only necessary to change two parts of the software example of Section 6.10.4.1:

❑ The definition line for the UART clock:

```
UARTCLK  .equ          32768    ; ACLK is used for UARTCLK
```

❑ The initialization subroutines TXINIT and RCINIT. Instead of the MCLK, the ACLK needs to be defined with the initialization subroutines (SSEL0 = 1, SSEL1 = 0). The simplest way is to use the subroutines of this Section (6.10.4.2).

6.10.4.2.2 ACLK With the Low Power Mode 3

This section shows another approach. With this example, the CPU is normally off and leaves the LPM3 only for the interrupt handling and after a complete character is received or transmitted.

Example 6–66. Half duplex UART With Interrupt

Half duplex UART software using the UART interrupt is shown. It is designed for:

❑ Baud rate: 2400 baud

❑ The ACLK (32,768 Hz) is used for the UART clock

❑ Eight data bits

❑ Parity enabled with odd parity

❑ Address bit included

❑ Two stop bits

❑ Reception of correct characters only (no error indication, restart instead)

❑ The CPU normally uses the low power mode 3 (LPM3)

❑ UART signals like shown in figure 6–87 (mark = V_{CC}, space = V_{SS})

The software switches have the same function as described in Section 6.10.4.1. The UARTCLK is defined with the crystal frequency.

Also, this example uses a looped calculation for the correction of the bits. Not only eight different bits are calculated, but all of the bits of a frame (9 to 13) are calculated individually. See the software part starting at the label MODTAB.

Transmit Mode: the data to be transmitted is loaded right-aligned into the RAM word RTDATA. The address bit — if enabled by ADDR = 1 — is included. No error is possible. Four examples for the data in RTDATA are shown in figure 6–95. The completion of the transmission is indicated by the value (TX6–RTTAB) in the status byte RTSTAT. The interrupt routine outputs the character and resets after the completion the CPUoff bit and the SCG1 and SCG0 bits of the stored status register on the stack. This manipulation omits the return to LPM3 and initializes the next transmit sequence.

Receive Mode: the received data is loaded left-aligned into the RAM word RTDATA. This means that depending on the address bit and the number of data bits contained in the data word, a shift is necessary to get a single byte containing the received character. Examples for the data are shown in figure 6–96. The input format used is necessary due to the address bit. The completion of the reception is indicated by the value (RC6–RTTAB) in the status byte RTSTAT. After the reception of a complete character, the interrupt handler resets the CPUoff bit and the SCG1 and SCG0 bits of the stored status register on the stack. This manipulation omits the return to LPM3 and allows the processing of the received data. The error handling is the same as shown for the example in Section 6.10.4.1.

```
; Definitions for the common part
;
STACK    .equ    0300h          ; Stack start address
FLLMPY   .equ    32             ; FLL multiplier for 1,048MHz
;
; Definitions for the UART. Data format:
; odd parity, 8 data bits, address bit, 2 stop bits
; ACLK for UART clock, only correct characters to input buffer
;
Baudr    .equ    2400           ; Baud rate is 2400 Baud
UARTCLK  .equ    32768          ; ACLK is used for UARTCLK
CHARC    .equ    8              ; Length:   7: 7 bits    8: 8 bits
ADDR     .equ    1              ; Address bit: 1 yes    0 no
PAR      .equ    1              ; Parity   0: disabled  1: enabled
PAREV    .equ    0              ; Parity   0: odd       1: even
STB      .equ    2              ; Stop bits:   1: one   2: two
TCERRT   .equ    0              ; 0: error restart   1: indication
;
TCPE     .equ    1              ; Parity error: RTERR.0 = 1
TCFE     .equ    2              ; Frame error:  RTERR.1 = 1
CUBR     .equ    -(UARTCLK/Baudr)  ; Content 8-Bit Counter
;
         .even                  ; Word boundary
         .bss    RTDATA,2       ; Data for receive/transmit
         .bss    RTERR,1        ; Error byte
         .bss    RTSTAT,1       ; Status byte
;
         .text                  ; Software start address
;
INIT     MOV     #STACK,SP      ; Initialize Stack Pointer
         CALL    #INITSR        ; Init. FLL and RAM
         ...                    ; Proceed with initialization
         EINT                   ; Enable interrupts
         ...
;
; Prepare the transmission of one character from RAM word
```

```
; RTDATA. Info is contained right aligned in the LSBs. No error
; is possible
;
        MOV      #xxx,RTDATA      ; Character xxx to RTDATA
        CALL     #TXINIT          ; Initialize the transmit part
        ...                       ; Continue with background
;
; Prepare the reception of one character to RAM word RTDATA
;
        CALL     #RCINIT          ; Initialize the receive part
        ...                       ; Continue in background
;
; After the completion of all background tasks, enter LPM3
;
PLPM3   BIS      #CPUoff+GIE+SCG1+SCG0,SR   ; Enter LPM3
;
; An interrupt handler cleared the CPUoff, SCG1 and SCG0 bits
; of the SR on the stack. Checks are made if activity is
; needed:
; Receive Mode:     one character is received
; Transmit Mode:    one character is output completely
; ...               other interrupt handlers
;
        CMP.B    #RC6-RTTAB,RTSTAT ; One character received?
        JEQ      CHAR_RC          ; Yes, process character
        CMP.B    #TX6-RTTAB,RTSTAT ; One character transmitted?
        JEQ      CHAR_TX          ; Yes, prepare next one
        ...                       ; Check other reasons
        JMP      PLPM3            ; Back to LPM3
;
; Common interrupt handler for transmit and receive functions.
; The carry of TCDAT is switched to the P0.1 interrupt request
; Interrupt time interval of the 8-bit timer is: 1/Baud rate
; The single status byte RTSTAT contains the actual status:
; Idle:      RTSTAT = 0                       No activity
; Transmit:  RTSTAT = 1...TX6-RTTAB-1         Active
```

```
;                      TX6-RTTAB               Character output
; Receive:   RTSTAT = RC-RTTAB...RC6-RTTAB-1  Active
;                      RC6-RTTAB               Char. received
;
TXRCINT  PUSH    R5              ; Save R5
         MOV.B   RTSTAT,R5       ; Receive/transmit status
         MOV.B   RTTAB(R5),R5    ; Offset to handler -> R5
         ADD     R5,PC           ; RTTAB+RTSTATx-RTTAB -> PC
RTTAB    .BYTE   RTSTAT0-RTTAB   ; Offset RTSTAT = 0 (inactive)
;
         ...                     ; Like shown for MCLK version
```

Note:

The interrupt handler for the UART when using the ACLK for the the UART clock is the same as the handler for when the MCLK is used. Only the small software part after the completion of a received or sent character (at label TXSTAT6) is slightly different. It resets the CPUoff, SCG1, and also SCG0 bits (SR.4 to SR.6) to allow a software activity after the return from interrupt RETI. Also, the first instructions of the initialization subroutines are different. These parts are shown below.

```
;
; One full character is received or transmitted. The UART
; hardware is switched off, the LPM3 is terminated to wake-up
; the CPU after the RETI. The status for a completed character
; is:
; Receive Mode:  RC6 - RTTAB
; Transmit Mode: TX6 - RTTAB
;
TXSTAT6  BIC.B   #P0IE1,&IE1        ; Disable TCDAT carry interrupt
         BIC.B   #RXACT+ENCNT,&TCCTL ; Stop T/C, conserve power
         BIC     #SCG1+SCG0+CPUoff,2(SP) ; Terminate LPM3
         JMP     RTSTAT0            ; To RETI
;
; Subroutines
; The subroutine prepares the 8-Bit Timer/Counter hardware to
; transmit data. Initialize control byte TCCTL:
; SSEL1/SSEL0: 0/1 for ACLK frequency
```

```
; ISCTL:     1   Carry of TCDAT register causes P0.1 intrpt
; TXE:       1   Enable output buffer for P0.2
; TXD        1   Set TXD (P0.2) to high (mark)
; ENCNT:     1   Enable clock to the TCDAT register
;
TXINIT   MOV.B   #SSEL0+ISCTL+TXE+TXD+ENCNT,&TCCTL
         MOV.B   #TX-RTTAB,RTSTAT  ; Transmit status, start bit
         JMP     RTINIT            ; To common part
;
; The subroutine prepares the 8-Bit Timer/Counter hardware to
; receive data. Initialize control byte TCCTL:
; SSEL1/SSEL0: 0/1 for ACLK frequency
; ISCTL:     1   Carry of TCDAT register causes P0.1 intrpt
; TXE:       1   Enable output buffer for P0.2
; TXD:       1   Set TXD (P0.2) to high (mark)
; RXACT:     0   Reset the Edge Detection Flip-Flop
;
RCINIT   MOV.B   #SSEL0+ISCTL+TXD+TXE,&TCCTL ; Control byte
         MOV.B   #RC-RTTAB,RTSTAT  ; Receive status, start bit
         CLR     RTDATA            ; Clear data word
         BIS.B   #2,&P0IES         ; Neg. edge detection on P0.1
;
; Common part for transmit and receive. The parity bit RTERR.0
; is initialized in a way, that always zero is returned, if the
; parity is ok.
;
RTINIT   MOV.B   #CUBR/2,&TCPLD    ; Half bit time to 1st intrpt
         MOV.B   #0,&TCDAT         ; Load half bit time to TCDAT
         MOV.B   MODTAB,&TCPLD     ; Bit time for 1st bit
         .if     (PAR=1)&(PAREV=0) ; Odd Parity enabled?
         MOV.B   #1,RTERR          ; Odd parity: RTERR.0 = 1
         .else
         MOV.B   #0,RTERR          ; No parity .or. even parity
         .endif
         BIS.B   #P0IE1,&IE1       ; TCDAT carry intrpt enabled
         BIS.B   #RXACT,&TCCTL     ; Receive: enable edge detect.
```

```
        RET
;
; Interrupt Vectors
;
        .sect    "SCIVEC",0FFF8h   ; HW/SW UART Vectors
        .word    TXRCINT           ; Common TX/RC Vector
        .sect    "INITVEC",0FFFEh  ; Reset Vector
        .word    INIT              ; Program Start Address
```

6.10.4.3 CPU Loading and Memory Space

6.10.4.3.1 CPU Loading

The CPU loading due to the UART activity can be calculated with simple formulas. The formulas are slightly different for the transmit and the receive mode, because they have different medium cycles per bit. The numbers are given for a frame with 8 data bits, parity enabled, no address bit, and two stop bits. This results in 13 interrupts per frame (the turn off of the 8-Bit Timer/Counter is included). The transmitted [resp.] received character is 0AAh with its sequence of ones and zeros.

The cycle count includes:

6	cycles to get to the 1st instruction of the interrupt handler
n	cycles for the interrupt handler itself
5	cycles for the RETI instruction

Not included are: the initialization subroutines, the data preparation for the transmit mode, and the data processing for the receive mode.

Transmit Mode — the sum of cycles for a complete frame is 708 cycles. The medium cycle count per transmitted bit is 708/13 = 54.46 cycles.

Receive Mode — the sum of cycles for a complete frame is 699 cycles. The medium cycle count per received bit is 699/13 = 53.77 cycles.

The formula to calculate the percentage for the CPU load due to the UART activity is:

$$CPULoad = \frac{BaudRate \times c}{fMCLK} \times 100$$

Where:

CPULoad	Loading of the MSP430 CPU by the UART	[%]
fMCLK	system clock used for the UART	[Hz]
BaudRate	Used baud rate of the UART	[Hz]
c	MCLK cycles per bit used by the interrupt handler	

If MCLK = 1.048 MHz and the baud rate = 4800 Hz, then the CPU loading is approximately 24.7%.

6.10.4.3.2 Memory Space

The memory space needed by the 8-Bit Timer/Counter UART depends on the UART format used and the enabled options. The minimum version is shown first and the additional bytes due to the enabled functions afterward. The numbers given include the interrupt handler TXRCINT and the two initialization subroutines TXINIT and RCINIT.

Minimum Version: (7 data bits, no address bit, no parity, one stop bit, error indication).

	202 ROM bytes, 4 RAM bytes
8 data bits	+ 4 bytes
Address bit included	+ 2 bytes
Parity enabled	+30 bytes
Two stop bits	+ 2 bytes
Error restart enabled	+16 bytes

Maximum Version: 256 ROM bytes, 4 RAM bytes.

6.10.4.4 UART Speed-Up Possibilities

The following ideas on how to speed up the UART come from Mark Buccini TI/Atlanta. It must be determined for each application if these possibilities can be used.

6.10.4.4.1 Dedicated CPU Register for the Status

The use of a dedicated CPU register for the status makes the saving and restoring of the needed register unnecessary. If it is incremented by two, it can step through a word table with minimum overhead.

```
; Initialization for transmit
;
TXINIT    ...                    ; Like described before
          MOV    #TX-RTTAB,R5    ; Initialize transmit status
          ...
```

```
;
; Interrupt handler: R5 contains the status in steps of two
;
TXRCINT  MOV      RTTAB(R5),PC    ; Start of handler to PC
RTTAB    .WORD    RTSTAT0         ; Address for R5 = 0 (inactive)
                                  ; Transmit states:
TX       .WORD    TXSTAT1         ; TX: Start bit
         .WORD    TXSTAT2         ; TX: LSB
         . . .
                                  ; Return from interrupt
RTRET    INCD     R5              ; To next status (steps of 2)
RTSTAT0  RETI
```

The autoincrement addressing mode may also be used to speed up the interrupt handler:

```
; Initialization for transmit
;
TXINIT   . . .                    ; Like described before
         MOV      #TX,R5          ; Initialize transmit status
         . . .
;
; R5 contains the address of the current table word
;
TXRCINT  MOV      @R5+,PC         ; Start of handler to PC
RTTAB    .WORD    RTSTAT0         ; Address = RTTAB (inactive)
                                  ; Transmit states:
TX       .WORD    TXSTAT1         ; TX: Start bit
         .WORD    TXSTAT2         ; TX: LSB
         . . .
                                  ; Return from interrupt
RTRET    RETI                     ; Next status yet in R5
RTSTAT0  DECD     R5              ; Completed: last status
         RETI
```

6.10.4.4.2 No Baud Rate Correction

No baud rate correction is needed if the MCLK is used for the baud rate Generation. This allows a shorter interrupt handler with fewer cycles and less program space.

6.10.4.4.3 Word Table Instead of a Byte Table

If a word table instead of the byte table is used for the distribution at the start of the interrupt handler, then more program space is needed, but the execution is faster. See Section 6.10.4.4.1.

6.10.4.4.4 Mixture of the Methods

The two sources for the UART clock are detailed in sections 6.10.4.1 (MCLK) and 6.10.4.2 (ACLK) may be mixed to get the best of both worlds:

Transmit Mode — the program normally uses the LPM3. If a character needs to be output, then the active mode with its MCLK is used. The software is identical to the transmit mode shown in Section 6.10.4.1.

Receive Mode — the program normally uses the LPM3 with the interrupt of the P0.1 pin activated on negative edges (start bit).

❑ The initialization subroutine is the same as shown in Section 6.10.4.1 with the exception of:

■ The bit ISCTL in the control register TCCTL is reset to enable the interrupt at pin P0.1 for negative edges.

❑ The next start bit wakes up the MSP430, which starts the following activities:

■ The control loop of the system clock generator is closed to get a controlled MCLK frequency (SCG0 = 0)

■ The interrupt source is switched from the input pin P0.1 to the carry of the 8-bit counter (ISCTL = 1)

❑ The MCLK stays active until the complete character is received. The LPM3 is activated again after the processing of the received data.

6.11 The Comparator_A

The Comparator_A module is contained in some members of the MSP430x1xx family. It can be used for precise analog measurements. Figure 6–97 shows the versatile hardware of the module.
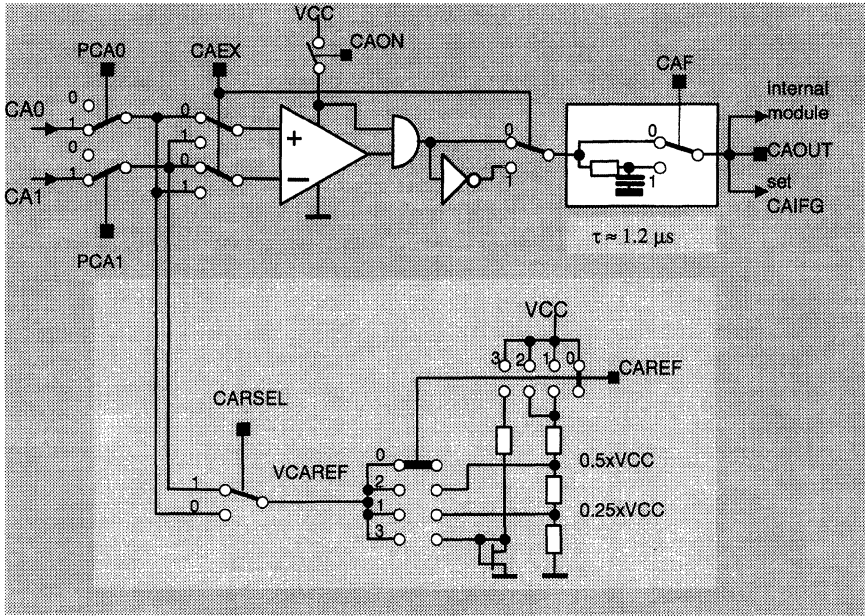
Figure 6–97. Comparator_A Hardware

6.11.1 Definitions Used With the Application Examples

The abbreviations used for the hardware definitions are consistent with the *MSP430 Architecture User's Guide.*

```
; HARDWARE DEFINITIONS
; COMPARATOR_A
;
CACTL1    .equ    059h         ; Control Register 1
CAIFG     .equ    001h         ; Interrupt Flag
CAIE      .equ    002h         ; Interrupt Enable Flag
CAIES     .equ    004h         ; Edge Select 0: rising 1: falling
CAON      .equ    008h         ; Supply      0: off    1: off
CAREF0    .equ    010h         ; 00: off        01: 0.5xVcc
CAREF1    .equ    020h         ; 10: 0.25xVcc 11: Vref
CARSEL    .equ    040h         ; Reference to: 0: CA0  1: CA1
CAEX      .equ    080h         ; 0: CA0 -> +   1: CA1 -> +
;
CACTL2    .equ    05Ah         ; Control Register 2
CAOUT     .equ    001h         ; CA Output
CAF       .equ    002h         ; Output Filter 0: off   1: on
P2CA0     .equ    004h         ; Switch CA0  0: off     1: CA0 on
P2CA1     .equ    008h         ; Switch CA1  0: off     1: CA1 on
CACTL24   .equ    010h         ; Software Bits
CACTL25   .equ    020h
CACTL26   .equ    040h
CACTL27   .equ    080h
;
CAPD      .equ    05Bh         ; Control Register 3
CAPD0     .equ    001h         ; Input Buffer Switches Port 2
CAPD1     .equ    002h         ; 0: Input Buffer enabled
CAPD2     .equ    004h         ; 1: Input Buffer disabled
CAPD3     .equ    008h
CAPD4     .equ    010h         ; Avoid current through input buffers
CAPD5     .equ    020h         ; with analog signals
CAPD6     .equ    040h
CAPD7     .equ    080h
```

6.11.1.1 Attributes of the Comparator_A

The hardware allows all combinations of comparisons. The bit CAOUT (CACTL.0) contains the result of the comparison:

☐ Comparison of two external inputs

☐ Comparison of each external input with $0.25 \times V_{CC}$ or $0.5 \times V_{CC}$

☐ Comparison of each external input with an internal reference voltage

☐ An analog filter can be switched to the CAOUT output

☐ The module has interrupt capability for the leading and the trailing edge of the output signal CAOUT

6.11.2 Fast Comparator Input Check

Often a very fast sampling of sequential input values is necessary. The following measurement sequence is the fastest way to do this with the Comparator_A inputs. After the n input checks, a majority test — or something equivalent — can be made for a decision. Figure 6–98 shows the hardware used for the example. The software samples the voltage generated by the current Imeas through resistor Rm. A voltage drop higher than $0.25 \times V_{CC}$ sets CAOUT, a lower voltage drop resets CAOUT. After n samples, the number of sampled 1s is checked. Any other input combination may also be used.
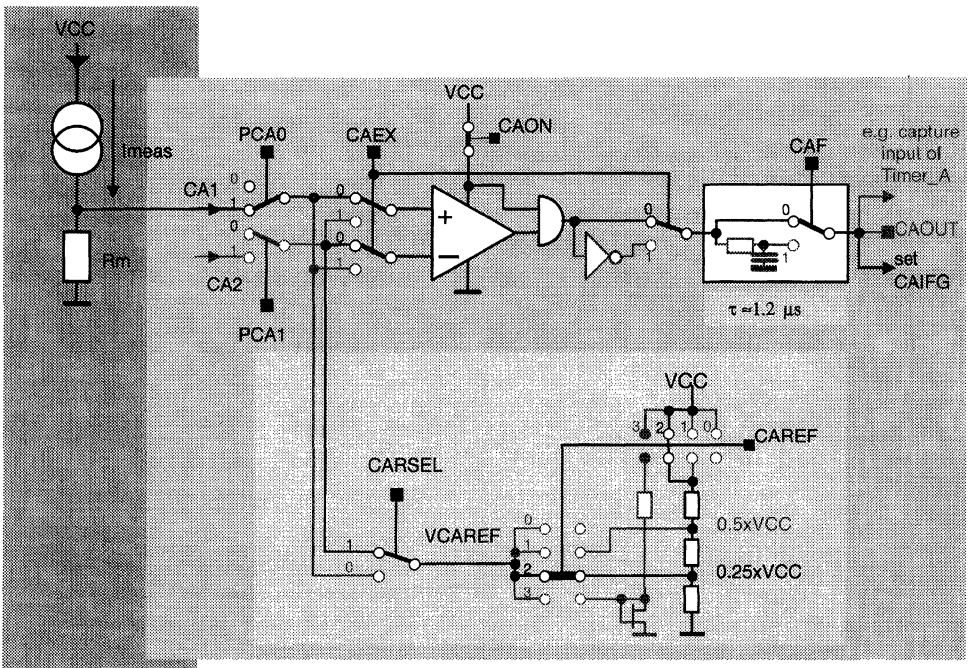
Figure 6–98. Fast Comparator Input Check Circuitry

```
; Fast test for the state of the Comparator_A input
;
        MOV.B   #CARSEL+CAREF1+CAON,&CACTL1 ; Define Comp_A mode
        MOV.B   #PCA0,&CACTL2
        MOV     #CACTL2,R15     ; Prepare pointer to reg. CACTL2
        ...
        MOV.B   @R15,R5         ; Sample CAOUT (CAOUT = CACTL2.0)
        ADD.B   @R15,R5         ; Add next sample
        ...                     ; Add following samples
        ADD.B   @R15,R5         ; Add sample n
;
; Test if CAOUT showed more than n/2 times a positive result
;
        SUB     #n*PCA0,R5      ; Correct result
        CMP.B   #1+(n/2),R5     ; R5 - (1+n/2)
        JHS     POS             ; More samples are 1
        ...                     ; More samples are 0
```

or an even faster decision:

```
; Test if CAOUT showed more than n/2 times a positive result
;
        CMP.B   #n*PCA0+1+(n/2),R5 ; R5-n*PCA0+(1+n/2)
        JHS     POS             ; More samples are 1
        ...                     ; More samples are 0
```

6.11.3 Voltage Measurement

Figure 6–99 shows hardware that can be used for the measurement of external voltages. The supply voltage is used for reference. The measurement principle is the same one as shown in section *Voltage Measurement with the Universal Timer Port/Module.*

$$0.25 \times Vcc \times \frac{R1 + R2 + R3}{R2 + R3} < \quad Vin \quad < Vcc \times \frac{R1 + R2 + R3}{R2 + R3}$$
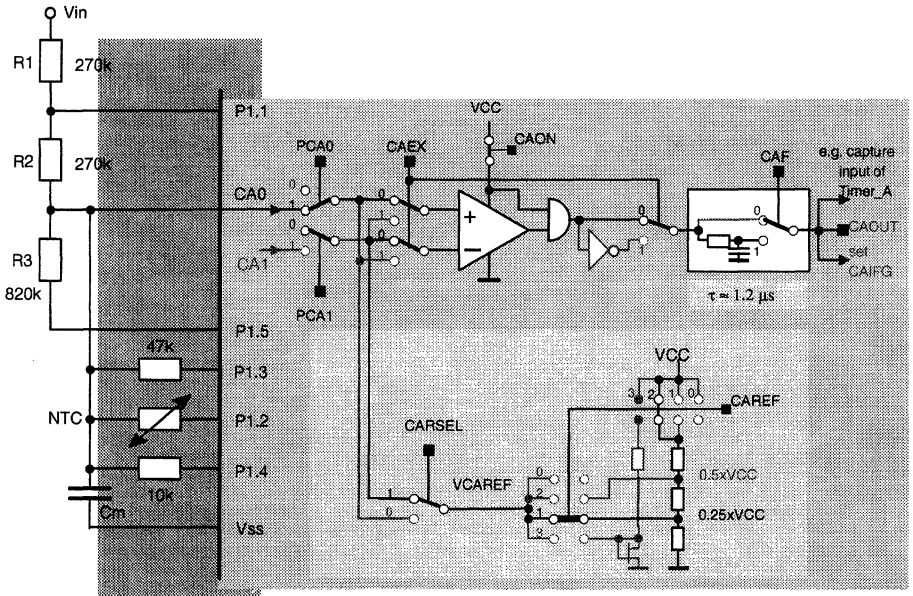
Figure 6–99. Voltage Measurement

Hints and Recommendations

7.1 Hints and Recommendations

During the software development for the first MSP430 projects, a lot of experience was acquired. The following hints and recommendations are for all programmers and system designers having more experience with 4-bit and 8-bit microcomputers than with 16-bit systems. Also mentioned are deviations the MSP430 family shows when compared to other 16-bit architectures (e.g., the function of the carry bit as an inverted zero bit with some instructions).

❑ **Frequently Used Bits**: these bits should always be located in bit positions 0, 1, 2, 3, 7, or 15. The first four bits can be set, reset, and tested with constants coming from the constant generator (1, 2, 4, 8), and the last two can be easily tested with the conditional jump instructions JN and JGE:

```
TST.B  RSTAT       ; TEST Bit7 (OV <- 0)
JGE    BIT7LO      ; JUMP IF MSB OF BYTE IS 0
TST    MSTAT       ; TEST Bit15 (OV <- 0)
JN     BIT15HI     ; JUMP IF MSB OF WORD IS 1
```

❑ **Use of BCD arithmetic**: If simple up/down counters are used and are to be displayed. This saves time and ROM space because no unnecessary binary–BCD conversion is needed.

EXAMPLE: Counter1 (four BCD digits) is incremented. Counter2 (eight BCD digits) is decremented by one.

```
CLRC                      ; DADD adds Carry bit too!
DADD   #0001,COUNTER1     ; INCREMENT COUNTER1 DECIMALLY
CLRC
DADD   #9999h,COUNTER2    ; DECREMENT 8 DIGIT COUNTER2
DADD   #9999h,COUNTER2+2  ;DECIMALLY
```

❑ **Conditional Assembly**: This feature of the MSP430 assembler allows more than one version out of one source of code. This drastically reduces the effort to maintain software. Only one version needs to be updated if changes are necessary. See Section 9.2.1, *Conditional Assembly* and Floating Point Software Examples.

❑ **Use of Bytes**: Use bytes wherever appropriate. The MSP430 allows the use of every instruction with bytes. The only exceptions are the instructions SWPB, SXT, and CALL.

❑ **Use of Status Bytes or Words**: Use status bytes or words, not flags for the storage of states. This allows extremely fast branching with only one instruction to the appropriate handler. Otherwise, a time (and ROM) consuming skip chain is necessary (also see Section 9.2).

❑ **Computing Software**: Use integer routines if speed is essential. Use the floating point package if complex or very accurate calculations are needed.

❑ **Bit Handling Instructions**:

With the bit handling instructions (BIS, BIT and BIC) more than one bit can be handled simultaneously. Up to 16 bits can be handled with a single instruction.

The BIS instruction is equivalent to the logical OR and can be used this way.

The BIC instruction is equivalent to the logical AND with the inverted source and can be used this way.

❑ **Use of the Addressing Modes**:

Use the symbolic mode for random accesses

Use the absolute mode for fixed hardware addresses such as peripheral addresses

Use the indexed mode for random accesses in tables

Use the register mode for time critical processing and as the normal one

Use assigned registers for extremely critical purposes. If a register always contains the same information, then it is not necessary to save it and to load it afterwards. The same is true for the restoring of the register when the task is done.

❑ **Stack Operations**:

All items on the stack can be accessed directly with the indexed mode. This allows completely new applications compared with architectures that have only simple hardware stacks.

The stack size is limited only by the available RAM, not by hardware register limitations.

Note:

The previously mentioned possibilities make strict housekeeping necessary. Every program part that uses the stack has to ensure that only relevant information remains on the stack and that all irrelevant data is removed. If this rule is not used consequently, the stack will overflow or underflow. If complex stack handling is used, it is advised to draw the stack with its items and the stack pointer as shown with the examples *Argument Transfer with Subroutine Calls* in Chapter 9. The drawn stack allocation gives a good overview.

❑ **The Program Counter (PC)**: The PC can be accessed as every other register with all instructions and all addressing modes. Be very careful when using this feature. Do not use byte instructions when accessing the PC, due to the clearing of the upper byte when used.

❑ **The Status Register (SR)**: The SR can be accessed in register mode only. Every status bit can be set or reset alone or together. This feature can be used for status transfer in subroutines. The FPP uses this type of status transfer.

❑ **Enabling of the General Interrupt**: The instruction that follows the enabling of the interrupt is executed before an interrupt is accepted:

```
EINT              ; Enable interrupt (GIE)
CLRC              ; This instruction is executed before
ADC R5            ; the 1st interrupt is accepted
```

❑ **High–Speed Multiplication**: If the fastest possible speed is necessary for multiplications and the hardware multiplier is not available, then the loop overhead can be omitted.

Straight through programming: the effort used for the looping can be saved if the shifts and adds are programmed straight through. The routine ends at the known MSB of the multiplicand (here, at bit 13 due to an ADC result (14 bits) that is multiplied):

```
;
; EXECUTION TIMES FOR REGISTER USE (CYCLES, CALL not included):
;
; TASK        CYCLES     EXAMPLE
;-------------------------------------------------------------
; MINIMUM     80         00000h x 00000h = 000000000h
; MEDIUM      96         0A5A5h x 05A5Ah = 03A763E02h
; MAXIMUM     112        0FFFFh x 0FFFFh = 0FFFE0001h
; Fast Multiplication Routine: Part used by signed and unsigned
; Multiplications. R5 x R4 -> R8|R7
;
MACUF  CLR   R6      ; MSBs MULTIPLIER
;
       RRA   R4      ; LSB to carry
       JNC   L$01    ; IF ZERO: SKIP ADD
       ADD   R5,R7   ; IF ONE: ADD MULTIPLIER TO RESULT
       ADDC  R6,R8
L$01   RLA   R5      ; MULTIPLIER x 2
       RLC   R6 ;
;
```

```
         RRA    R4     ; LSB+1 to carry
         JNC    L$02   ;
         ADD    R5,R7  ;
         ADDC   R6,R8
L$02     RLA    R5     ;
         RLC    R6     ;
  ....                 ; same way for bits 2 to 12
;
         RRA    R4     ; MSB to carry (here bit13)
         JNC    L$014  ;
         ADD    R5,R7  ;
         ADDC   R6,R8  ; No shift for multiplier necessary
L$014    RET           ; Return with result in R8|R7
;
```

□ **Emulation of Jump if Positive**: No jump if positive is provided, only a jump if negative. But after several instructions it is possible to use the jump if greater than or equal (JGE) for this purpose. But it must be certain that the instruction preceding the JGE resets the overflow bit V. The following instructions ensure this:

TST, SXT, RRA, BIT, AND.

The use of this emulation should be noted in the comment field to ease software modifications.

□ **Special Use of the Carry Bit**: The following instructions have a special feature that is valuable during serial to parallel conversion. The carry acts as an inverted zero bit. This means if the result of an operation is zero then the carry is reset and vice versa. The instructions having this feature are:

XOR, SXT, INV, BIT, AND.

Without using this feature a typical sequence for the conversion of an I/O-port bit to a parallel word would look like the following:

```
RLA    R5         ; Free bit 0 for next info
BIT    #1,&IOIN   ; P0.0 high ?
JZ     L$111
INC    R5         ; Yes, set bit 0
L$111  ...         ; Info in bit 0
```

Using this feature, the previous sequence is shortened to two instructions:

```
BIT    #1,&IOIN   ; P0.0 high ? .NOT.Zero -> carry
RLC    R5         ; Shift bit (in Carry) into R5
```

☐ **The Carry Bit Used for Increments**: The carry bit can be used if increments by one are necessary.

EXAMPLE: If the RAM word COUNT is greater than or equal to the value 1000 then a word COUNTER is to be incremented by one.

```
CMP    #1000,COUNT      ; COUNT >= 1000
ADC    COUNTER          ; If yes, (C = 1) incr. COUNTER
```

☐ **Immediate Addition of the Carry Bit**: The carry bit can be added immediately. No conditional jumps are necessary for counters longer than 16 bits. A 48-bit counter is incremented.

```
ADD    R5,COUNT         ; Low part of COUNT
ADC    COUNT+2          ; Medium part
ADC    COUNT+4          ; High part of 48-bit counter
```

☐ **Fall Through Programming**: ROM space is saved if a subroutine call that is located immediately before a RET instruction is changed. The called subroutine is located after the instruction before the CALL, and the program falls through it. This saves 6 bytes of ROM: The CALL itself and the RET instruction. The I2C handler uses this mode.

```
; Normal way: SUBR2 is called, afterwards returned
;
SUBR1  ...
       MOV    R5,R6
       CALL   #SUBR2    ; Call subroutine
       RET
;
; "Fall Through" solution: SUBR2 is located after SUBR1
;
SUBR1  ...
       MOV R5,R6         ; Fall through to SUBR2
;
SUBR2  ...              ; Start of  subroutine SUBR2
       RET
```

☐ **Shift Operations for 32–Bit Numbers**: If shifts with numbers greater than 16 bits are necessary, the shift operations for the upper words must be RLC or RRC. If RLA or RRA are used then only zeroes are shifted in

```
RLA    R11      ; MSB of low byte to carry
RLC    R12      ; RLA is wrong here!
RRA    R12      ; LSB of high byte to carry
RRC    R11      ; RRA is wrong here!
```

☐ **Interrupt Handlers**: The length of interrupt handlers should be kept as short as possible. All necessary calculations should be made in the background program (main program). The activation and control can be made easily with status bytes.

❑ **Decimal Subtraction**: No instruction is provided, but a simple way is possible. The copy instruction is only necessary if the minuend may not be modified.

EXAMPLE: OP1 is subtracted from OP2 decimally

```
MOV    OP1,R5        ; Copy Op1
ADD    #6666h,R5     ; OP1 into range 6666h to FFFFh
INV    R5            ; Build 9999 complement
SETC
DADD   R5,OP2        ; OP2 - OP1 -> OP2 (dec.)
```

❑ **Timer Wake–Up Out of Low Power Modes**: The two 8-bit counters of the universal timer/port can also be used during the low power modes 3 and 4. If a counter is incremented by an external signal (inputs CIN, CMP, or TPIN.5) from 0FFh to 0h, then the appertaining RCxFG-flag is set. If an interrupt is enabled, the CPU wakes up.

7.2 Design Checklist

Several steps are necessary to complete a system consisting of an MSP430 and its peripherals with the necessary performance. Typical and recommended development steps are shown in the following. All of the tasks mentioned should be done carefully in order to prevent trouble later on.

1) Definition of the tasks to be performed by the MSP430 and its peripherals

2) Selection of the MSP430 version that fits best

3) Worst case timing considerations for all of the tasks to be done (interrupt timing, calculation times, I/O etc.)

4) Drawing of a complete hardware schematic. Deciding which hardware options are used (supply voltage, pull-downs at the I/O-ports, etc.)

5) Worst case design for all of the external components

6) Organization of the RAM and — if present — of the external EEPROM

7) Flowcharting of the complete software

8) Coding of the software with an editor

9) Assembling of the program with the ASM430 Assembler

10) Removing of the logical errors found by the ASM430 Assembler

11) Testing of the software with the SIM430 Simulator and an emulation board

12) Repetition of the steps 7 to 10 until the software is free of errors

7.3 Most Frequent Software Errors

During software development, the same errors are made by nearly all assembler programmers. The following list contains the errors which are most often heard of and experienced.

❏ **Missing Housekeeping During Stack Operations**: If items are removed from or placed onto the stack during subroutines or interrupt handlers, it is mandatory to keep track of these operations. Any wrong positioning of the stack pointer will lead to a program crash, due to the wrong data written into the program counter.

❏ **Missing Initialization of the Stack Pointer**: The stack pointer needs to be initialized before the EINT instruction is executed or a CALL is used. The normal instruction to be used is:

```
MOV    #0300h,SP       ; Locate stack at high RAM
```

❏ **Use of the Wrong Jump Instructions**: The conditional jump instructions JLO and JLT, or JHS, and JGE, give different results if used for numbers above 07FFFh. It is therefore necessary to always distinguish between signed and unsigned jump instructions.

❏ **Wrong Completion Instructions**. Despite their virtual similarity, subroutines and interrupt handlers need completely different actions for completion.

■ Subroutines end with the RET instruction. Only the address of the next instruction (the one following the subroutine call) is popped from the stack.

■ Interrupt handlers end with the RETI instruction. Two items are popped from the stack, first the status register is restored and afterwards the address (the address of the next instruction after the interrupted one) is popped from the stack to the program counter.

■ If RETI and RET are used incorrectly, a wrong item is written into the PC. This means that the software will continue at random addresses and will hang–up.

❏ **Addition and Subtraction of Numbers With Differently Located Decimal Points**: if numbers with virtual decimal points are used the addition or subtraction of numbers with different fractional bits leads to errors. It is necessary to shift one of the operands in a way to achieve fractional parts of equal length. See "Rules for the Integer Subroutines."

❏ **Byte Instructions Applied to Working Registers**: byte instructions always clear the upper byte of the used working register (except CMP.B, TST.B, BIT.B). It is necessary therefore to use word instructions if operations in working registers can exceed the byte range.

❏ **Use of Byte Instructions With the Program Counter as Destination Register:** if the PC is the destination register byte instructions do not make sense. The clearing of the PC high byte is certainly wrong in any case. Instead, a register is to be used before the modification of the PC with the byte information. See 9.2.5.

❏ **Use of Falsely Addressed Branches and Subroutine Calls:** The destination of branches and calls is used indirectly, and this means the content of the destination is used as the address. These errors occur most often with the symbolic mode and the absolute mode:

```
CALL   MAIN          ; Subroutine's address is stored in MAIN
CALL   #MAIN         ; Subroutine starts at address MAIN
```

The real behavior is easily seen when looking to the branch instruction. It is an emulated instruction using the MOV instruction:

```
BR     MAIN          ; Emulated instruction BR
MOV    MAIN,PC       ; Emulation by MOV instruction
```

The addressing for the CALL instruction is exactly the same as for the BR instruction.

❏ **Counters and Timers Longer Than 16-Bits:** If counters or timers longer than 16 bits are modified by the foreground (interrupt routines) and used by the background, it is necessary to disable the timer interrupt (most simply with the GIE bit in SR) during the reading of these words. If this is not done, the foreground can modify these words between the reading of two words. This would mean that one word read contains the old value and the other one the modified value.

EXAMPLE: The timer interrupt handler increments a 32–bit timer. The background software uses this timer for calculations. The disabling of the interrupts prevents the timer interrupt that occurs between the reading of TIMLO and TIMHI, which can falsify the read information. This can be the case if TIMLO overflows from 0FFFFh to 0000h during the interrupt routine. TIMLO is read with the old information 0FFFFh and TIMHI contains the new information x+1.

```
BT_HAN   INC    TIMLO          ; Incr. LO word
         ADC    TIMHI          ; Incr. HI word
         RETI

         . . .

;
; Background part copies TIMxx for calculations
;
         DINT                  ; GIE <- 0
```

```
        NOP                         ; DINT needs 2 cycles
        MOV     TIMLO,R4            ; Copy LSDs
        MOV     TIMHI,R5            ; COPY MSDs
        EINT                        ; Enable interrupt again
```

❏ **Counters Used by Foreground and Background**: If counters are modified by the foreground and read and cleared by the background, care is to be taken that no counts are lost. With the following example, it is possible to loose a count if the interrupt occurs between the MOV and the CLR instruction. The additional count is not recognized because CNTR (with its content 1) is cleared.

```
; First the WRONG sequence is shown:
;
INT_HAN  INC CNTR                   ; Incr. counter CNTR    WRONG!
         RETI                       ; by interrupts
         ...                        ; Background program
         MOV     CNTR,STORE         ; Read CNTR
         CLR     CNTR               ; A count may be lost!
         ...
```

To avoid loosing a count, the following solutions are possible for the background part.

```
; Background part switches off the interrupt during reading
;
        DINT                        ; GIE <- 0 (inactive after MOV)
        MOV     CNTR,STORE          ; Read CNTR
        CLR     CNTR                ; Clear unmodified CNTR
        EINT                        ; Enable interrupt again
;
; Background part uses difference of contents. If interrupt occurs
; after the PUSH instruction, 1 remains in CNTR.
;
        PUSH    CNTR                ; Copy CNTR
        SUB     @SP,CNTR            ; Subtract read number from CNTR
        POP     STORE               ; Place read info to STORE
;
; Simplified version of above: if CNTR is yet changed it contains
; despite correct value (1)
;
        MOV     CNTR,STORE          ; Copy CNTR to STORE
        SUB     STORE,CNTR          ; Subtract STORE of current CNTR
```

❏ **Use of the PUSH Instruction**: When using sophisticated stack processing, it is often overlooked that the PUSH instruction decrements the stack pointer first and moves the item afterwards.

EXAMPLE: The return address stored at TOS is to be moved one word down to free space for an argument.

```
PUSH   @SP      ; WRONG! 1st free word (TOS-2) is copied
                ; on itself
PUSH   2(SP)    ; Correct, old TOS is pushed 1 word down
```

EXAMPLE: The stored SP does not point to the same stack address after the restoring. It points to the (address –2) afterwards.

```
PUSH  SP        ; Store SP-2 on stack
POP   SP        ; Restore SP-2 to SP !!
```

❑ **Use of the Autoincrement Mode:** The source register is incremented immediately after the reading of the source operand. This means if the source register is also used for the addressing of the destination operand, it contains the incremented value when used.

❑ **Register Overflow:** If registers do not have the necessary length, negative numbers (MSB = 1) or too small numbers (register is reset to zero by overflow) can result. The length of registers needs to be evaluated with *worst case* methods.

❑ **Interrupt Blocking:** Long interrupt routines should be avoided. If they are necessary, the GIE bit located in the SR should be set (instruction EINT) at the start of these routines. Otherwise, the disabled interrupt blocks all other interrupt sources.

❑ **Real Time Processing:** If the algorithm used is longer than the time slot that is available, errors will occur. *Worst case* evaluations are necessary to ensure the algorithm fits into the time slot.

❑ **Write-Only Registers:** The complete information always needs to be written to these registers. Otherwise, the bits not included in the source of the instruction are reset. The crystal buffer control register (CBCTL) is an example for this register type.

❑ **Port Select Registers:** I/O ports with dual functions – like the Port3 for the Timer_A (MSP430C33x) – must be switched to the second function. Otherwise, the normal port function is active. To switch Port3 completely to the Timer_A functions, the following code is needed during the initialization routine. If the Basic Timer is not initialized, the LCD will not work correctly.

```
MOV.B   #TACLK+TA4+TA3+TA2+TA1+TA0,&P3SEL   ; Init. Timer_A
```

❑ If the basic timer is not initialized, the LCD will not work properly.

7.4 Most Frequent Hardware Errors

❏ **Crystal Connection**: The crystal oscillator is connected to AVCC and AVSS. This means that these two terminals must be connected to DVCC and DVSS, otherwise the crystal will not oscillate.

❏ **Open Inputs**: Every input must have a defined potential. Otherwise, hum and noise will influence the program flow. In addition, the supply current increases. See Section 4.9.4, *Correct Termination of Unused Terminals*.

❏ **Crystal Turnon Time**: If woken-up from the low power mode 4, the crystal needs a relatively long time until it runs with the correct frequency. This can last up to four seconds. Correct timing is not possible until the crystal reached its nominal frequency. Until this, the DCO steps down to its lowest frequency ($\approx$ 500 kHz). See Section 6.5, *The System Clock Generator*.

❏ **Frequency–locked loop considerations**

■ **FLL Turnon Time:** If woken-up from LPM3, the FLL needs approximately 6 cycles to reach the nominal frequency. This also means, the 1st instruction of an interrupt handler is executed with the correct frequency.

■ **Setting Time:** The FLL needs a certain non-interrupted time to set the control value of the digitally-controlled oscillator (DCO). If this time is not provided, no control for the DCO is possible. It remains at the same tap. This time is best spent during initialization by a software loop with a worst case length of 28 x 32 x 30.5 μs = 27.3 ms. To allow the system clock the adaptation of the DCO to the eventually changed tap, the FLL-loop should be closed during longer calculations. This is done simply with the instruction:

```
BIC      #SCG0,SR        ; Turn on FLL-loop control
```

❏ **Supply Voltage for Battery-Powered Systems**: if certain batteries are used the supply voltage may fall below the lower voltage limit during Active Mode (especially if the ADC is used) due to the high internal resistance of these batteries. A capacitor is necessary then in parallel with the battery.

❏ **Supply Voltage for AC-Powered Systems**: No hum, noise, or spikes are allowed. If these are present, the reliability of the system and the accuracy of the ADC will decrease.

❏ **EEPROM Clocking**: For some EEPROMs, the minimum clock duration is longer than one MSP430 instruction. This means that NOPs have to be included into the clock timing. See the specification of the EEPROM used.

7.5 Checklist for Problems

7.5.1 Hardware Related Problems

1) Initialization circuit connected? An RC circuit is not sufficient in most cases.

2) Fan–out of bus or outputs taken into account?

3) Open inputs (interrupt, init, inputs etc.): every input must be connected to a defined voltage level. Otherwise, undefined signals (normally the ac frequency) are seen at the inputs. See Section 4.9.4, *Correct Termination of Unused Terminals.*

4) Crystal turn–on time not taken into account (may be up to seconds for low–power devices)?

5) Correct levels at all inputs? (low and high levels)

6) Input signals in specified limits: thresholds, frequency and edges?

7) Supply voltage in specified limits, no spikes, no noise etc.

8) External interrupt signals too short (no response from interrupted system)

9) External EEPROM, clock out of the MSP430 too fast or too short? See EEPROM specification.

10) RESET signals with spikes or false voltage levels? This is an often occurring reason for problems.

7.5.2 Software Related Problems

1) Register overflow (registers, memory and peripheral registers) causes negative numbers or sawtooth characteristic of results (numbers are too small then)

2) PWM applications: loading of the pulse length register needs to be synchronized to the output change. Otherwise, undefined pulses are output during the change of this register

3) Output frequency too high? (register load time longer than pulse length?)

4) Real-time applications: is used algorithm shorter than the available time interval also under *worst case* conditions?

5) Conditional jumps: signed and unsigned jumps used correctly? For example JHS and JGE are completely different instructions. The same is true for JLO and JLT.

6) Missing *housekeeping* during stack operations? If the return values for the SR and the PC – stored on the stack during an interrupt – are overwritten with data: this can cause the setting of the OSCoff bit in the SR during the RETI instruction. Which means, the program execution ceases.

7) Read–out of two–word–registers without disabling the interrupt? (if overflow occurs one word may contain the old number, the other one the new number)

8) Multiple word shifts: correct shift instruction used for the MSBs (no arithmetic shifts: they shift in always zeroes)

9) SP Initialization: forgotten or made after the *interrupt enabling* with EINT?

10) Interrupt handlers: long lasting parts without enabling the interrupt again (blocks all other interrupt activities)?

11) If the second function of a port register is used: are the select bits in the PxSEL register set?

12) Are all the peripherals initialized?

7.6 Run Time Estimation

To get a quick overview concerning the speed of a given piece of software, the following estimations may be used:

❏ If the code contains all addressing modes then the estimation for the needed runtime t_{run} is:

$$t_{run} \approx 0.75 \text{ cycles/byte}$$

❏ If the code contains only or predominant register mode addressing then the estimation for the needed runtime t_{run} is:

$$t_{run} \approx 0.5 \text{ cycles/byte}$$

Architecture and Instruction Set

8.1 Introduction

The instruction set of the *ultra low power*-microcomputer MSP430 family differs strongly from the instruction sets used by other 8-bit and 16-bit microcomputers. The reasons why this instruction set is appreciated though, are explained in the following pages in detail. It is the return to clarity and especially the return to orthogonality, an attribute of microcomputer architectures that has disappeared more and more during the last 20 years. A customer commented that it is an instruction set to fall in love with.

The MSP430 Family was developed to fulfill the ever increasing requirements of Texas Instruments Ultra Low Power microcomputers. It was not possible to increase the computing power and the real-time processing capability of the MSP430 predecessor (TSS400) as far as was needed. Therefore, a complete new 16-bit architecture was developed to stay competitive and be viable for several years.

The benchmark numbers shown in relation to competition's products (bytes used, number of program lines) are taken from an unbiased comparison executed by a British software consultant.

8.2 Reasons for the Popularity of the MSP430

The following sections are intended to explain the different reasons why the MSP430 instruction set, which closely mirrors the architecture, has become so popular.

8.2.1 Orthogonality

This notation of computer science means that a single operand instruction can use any addressing mode or that any double operand instruction can use any combination of source and destination addressing modes. Figure 8–1 shows this graphically: the existing combinations fill the complete possible space.
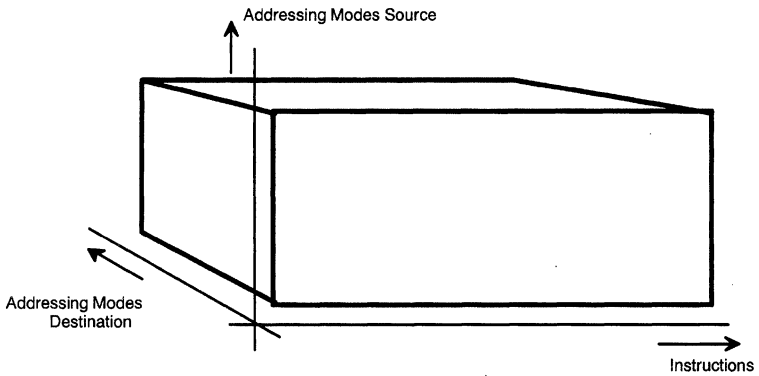
Figure 8–1. Orthogonal Architecture (Double Operand Instructions)

The opposite of orthogonal, a non-orthogonal architecture is shown in Figure 8–2. Any instruction can use only a part of the existing addressing modes. The possible combinations are arranged like small blocks in the available space.
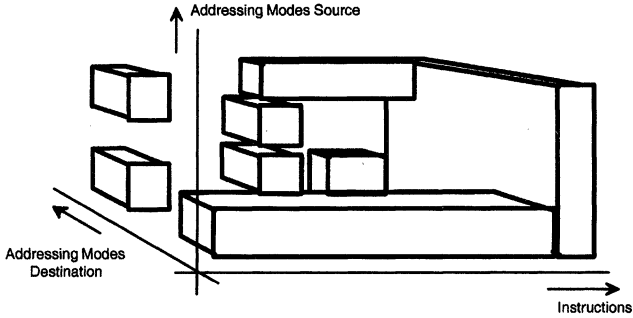
Figure 8–2. Non-Orthogonal Architecture (Dual Operand Instructions)

8.2.2 Addressing Modes

The MSP430 architecture has seven possibilities to address its operands. Four of them are implemented in the CPU, two of them result from the use of the program counter (PC) as a register, and a further one is claimed by indexing a register that always contains a zero (status register).

The single operand instructions can use all of the seven addressing modes, the double operand instructions can use all of them for the source operand, and four of them for the destination operand. Figure 8–3 shows this context:

Double Operand Instructions		**Single Operand Instructions**
Mnemonic Source,Destination		Mnemonic Destination

Register	Register	Register
Indexed	Indexed	Indexed
Absolute	Absolute	Absolute
Symbolic	Symbolic	Symbolic
Immediate		Immediate
Register indirect		Register indirect
Register indirect autoincrement		Register indirect autoincrement

12 Instructions	28 Combinations	7 Instructions 7 Addressing Modes

Figure 8–3. Addressing Modes

8-4

8.2.2.1 Register Addressing

The operand is contained in one of the registers R0 to R15. This is the fastest addressing mode and the one that needs the least memory. Example:

```
; Add the contents of R7 to the contents of R8
;
        ADD    R7,R8        ; (R7) + (R8) → (R8)
```

8.2.2.2 Indirect Register Addressing

The register used contains the address of the operand. The operand can be located anywhere in the entire memory space (64K). Example:

```
; Add the byte addressed by R8 to the contents of R9
;
        ADD.B  @R8,R9       ; ((R8)) + (R9) → (R9)
```

8.2.2.3 Indirect Register Addressing With Autoincrement

The register used contains the address of the operand. This operand can be located anywhere in the entire memory space (64K). After the access to the operand the used register is incremented by two (word instruction) respective one (byte instruction). The increment occurs immediately after the reading of the source operand. Example:

```
; Copy the byte operand addressed by R8 to R9
; and increment the pointer register R8 by one afterwards
;
        MOV.B  @R8+,R9      ; ((R8)) → (R9), (R8) + 1 → (R8)
```

8.2.2.4 Indexed Addressing

The address of the operand is the sum of the index and the contents of the register used. The index is contained in an additional word located after the instruction word. Example:

```
; Compare the 2nd byte of a table addressed by R5 with the
; low Byte of R15. Result to the Status Register SR
;
        CMP.B  1(R5),R15    ; (R15) − (1 + (R5))
;
```

If the register in use is the program counter then two additional, important addressing modes result:

8.2.2.5 Immediate Addressing

Any 16-bit or 8-bit constant can be used with an instruction. The PC points to the following word after reading the instruction word. By the use of the *register indirect autoincrement* addressing mode, this word (the immediate value) can be read and the PC is incremented by two afterwards. The word after the instruction word is treated this way as an 8-bit or a 16-bit immediate value. Example:

```
; Test bit 8 in the 3rd word of a table R10 points to.
; Start address of the table is 0(R10), 3rd word is 4(R10)
;
          BIT       #0100h,4(R10)    ; Bit 8 = 1?
;
; The assembler inserts for the instruction above:
;
          BIT       @PC+,4(R10)      ; Executed instruction
          .WORD     0100h            ; Source constant 0100h
          .WORD     0004h            ; Index 4 of the destination
```

8.2.2.6 Symbolic Addressing

This is the normal addressing mode for the random access to the entire 64K memory space. The word located after the instruction word contains the difference in bytes to the destination address relative to the PC. This difference can be seen as an index to the PC. Any address in the 64K memory map is addressable this way, both as a source and as a destination.

Example: $ = address the PC points to

```
; Subtract the contents of the ROM word EDE from the contents
; of the RAM word TONI
;
          SUB     EDE,TONI          ; (TONI) - (EDE) → (TONI)
;
; The assembler inserts for the instruction above:
;
          SUB     X(PC),Y(PC)       ; Executed instruction
          .WORD   X                 ; Index X = EDE-$
          .WORD   Y                 ; Index Y = TONI-$
```

8.2.2.7 Absolute Addressing

Addresses that are fixed (e.g., the hardware addresses of the peripherals like ADC, UART) can be addressed absolutely. The absolute addressing mode is a special case of the indexed addressing mode. The register used (SR) always contains a zero in this case (without loosing its former information!). Example:

```
; Set the Power Down Bit in the ADC Control Register ACTL
;
        BIS     #PD,&ACTL       ; Power Down Bit ADC <- 1
;
; The assembler inserts for the instruction above:
;
        BIS     @PC+,X(SR)      ; Executed instruction
        .WORD   01000h          ; PD Bit Hardware Address
        .WORD   00114h          ; X: Hardware Address of ACTL
```

8.2.3 RISC Architecture Without RISC Disadvantages

Classic RISC architectures provide several addressing modes only for the LOAD and STORE instructions; all other instructions can only access the (numerous) registers. The MSP430 can be programmed this way too. An example of this programming style is the floating point package FPP4. The registers are loaded during the initialization, the calculations are made exclusively in the registers, and the result is placed onto the stack.

In real time applications, this kind of programming is less usable, here it is important to access operands at random addresses without any delays. An example of this is the incrementing of a counter during an interrupt service routine:

```
; Pure RISC program sequence for the incrementing of a counter
;
INT_HND PUSH   R5              ; Save register
        LOAD   R5,COUNTER      ; Load COUNTER to register
        INC    R5              ; Increment this register
        STORE  R5,COUNTER      ; Store back the result
        POP    R5              ; Restore used register
        RETI                   ; Return from interrupt
;
; The MSP430 program sequence for the incrementing of a counter
;
INT_HND INC    COUNTER         ; Increment COUNTER
        RETI                   ; Return from interrupt
```

As shown in the previous example, the pure RISC architecture is not optimal in cases with few calculations, but necessary for fast access to the memory.

Here, the MSP430 architecture is advantageous due to the random access to the entire memory (64K) with any instruction, seven source addressing modes and four destination addressing modes.

8.2.4 Constant Generator

One of the reasons for the high code efficiency of the MSP430 architecture is the constant generator. The constants, appearing most often in assembler software, are small numbers. Out of these, six were chosen for the constant generator:

Table 8-1. Constants implemented in the Constant Generator

CONSTANT	HEX REPRESENTATION	USE
−1	0FFFFh	Constant, all bits are one
0	00000h	Constant, all bits are zero
+1	00001h	Constant, increment for byte addresses
+2	00002h	Constant, increment for word addresses
+4	00004h	Constant, value for bit tests
+8	00008h	Constant, value for bit tests

These six constants can also be used for byte processing. Only the lower byte is in use then.

The use of numbers out of the constant generator has two advantages:

Memory Space: The constant does not need an additional 16 bit word as it is the case with the normal immediate mode. Two useless addressing modes of the status registers SR and all four addressing modes of the otherwise unserviceable register R3 are used.

Speed: The constant generator is implemented inside the CPU which results in an access time similar to a general purpose register (shortest access time).

Most of the emulated instructions use the constant generator. See Chapter *The MSP430 Instruction Set* for examples.

8.2.5 Status Bits

The influence of the instructions to the status bits contained in SR is not as uniform as the instructions appear. Dependent on the main use of the instruction, the status bits are influenced in one of the following three ways shown:

1) Not at all, the status bits are not affected. This is, for example, the case with the instructions bit clear, bit set and move.

2) Normal: the status bits reflect the result of the last instruction. This is used with all arithmetical and logical instructions (except bit set and bit clear)

3) Normal, but the carry bit contains the inverted zero bit. The logical instructions XOR (exclusive or), BIT (bit test) and AND use the carry bit for the non-zero information. This feature can save ROM space and time. No preparations or conditional jumps are necessary. The tested information, which is contained in the carry bit, is simply shifted into a register or a RAM word respective byte.

8.2.6 Stack Processing

The stack processing capability of the MSP430 allows any nesting of interrupts, subroutines, and user data. It is only necessary to have enough RAM space. Due to the function of the SP as a general purpose register, it is possible to use all seven of the addressing modes for the SP. This allows any needed manipulation of data on the stack. Any word or byte on the stack can be addressed and may therefore be read and written. (The addressing modes implemented for the MSP430 were chosen primarily for the addressing of the stack; but they proved to be very effective also for the other registers).

8.2.7 Usability of the Jumps

Remarkable is the uncommonly wide reach of the jumps which is ±512 words. This value is eight times the reach of other architectures that use normally ±128 bytes. Inside program parts it is, therefore, necessary only very rarely to use the branch instruction with its normal two memory words and longer execution time. The implemented eight jumps are classified in three categories:

1) Signed jumps: Numbers range from –32768 to +32767 (word instructions) respective –128 to +127 (byte instructions)

2) Unsigned jumps: Numbers range from 0 to 65535 respective 0 to 255

3) Unconditional jump: (replaces the branch instruction normally)

8.2.8 Byte and Word Processor

Any MSP430 instruction is implemented for byte and word processing. Exceptions are only the instructions where a byte instruction would not make sense (subroutine call CALL, return from interrupt RETI) or instructions that are used as an interface between words and bytes (swap bytes SWPB, sign extension SXT). The addressable memory of the MSP430 is divided into bytes and words as shown in Figure 8–4.

15		0

Byte Address n+1	Byte Address n	Word Address n
Byte Address n+3	Byte Address n+2	Word Address n+2
Byte Address n+5	Byte Address n+4	Word Address n+4
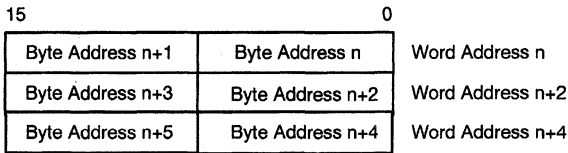
Figure 8–4. Word and Byte Addresses of the MSP430

This way, the entire 64K address space is organized. The planned memory extension will be addressed in the same clear manner. Due to this memory organization, any table can be allocated in the most favorable manner. Dependent on the maximum value of the operands, the table can be implemented as a byte table or a word table. Any general purpose register from R4 to R15 can be used as a pointer to the tables. The implemented addressing modes indexed, indirect, and indirect with autoincrement are intended for table processing.

8.2.9 High Register Count

In addition to the PC and the SP, which are usable for several purposes, twelve identical general purpose registers (R4 to R15) are available. Anyone of these registers can be used as a data register, as an address pointer, as an auto-incrementing address pointer, and as an index register. The bottleneck of the accumulator architectures, which have to pass any operation through the accumulator (with corresponding LOAD and STORE instructions), does not exist for the MSP430.

15 0	
PC	R0 Program Counter
SP	R1 Stack Pointer
SR	R2 Status Register and CG1
CG2	R3 Constant Generator 2
R4	R4 General-Purpose Register R4
	General-Purpose Registers R5 to R14
R15	R15 General-Purpose Register R15
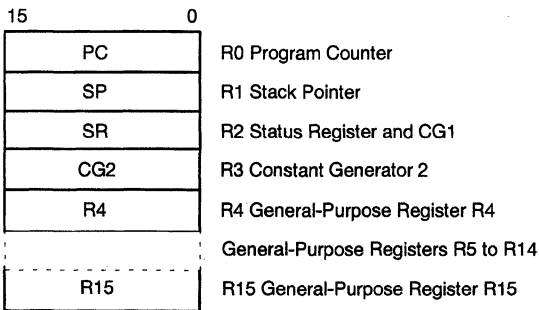
Figure 8–5. Register Set of the MSP430

8.2.10 Emulation of Non-Implemented Instructions

The 27 implemented instructions allow the emulation of additional 24 instructions. This is normally reached with the help of the constant generator, but other ways are used also. As the constants used are taken from the constant generator, no additional memory space is needed.

The assembler automatically inserts the correct instructions if emulated instructions are used. The emulation of the 24 instructions led to a remarkable smaller central processing unit. The MSP430 CPU is even smaller than some 4-bit CPUs. The emulated instructions are completely listed in Section 8.4.2, *Emulated Instructions*.

8.2.11 No Paging

The 16-bit architecture of the MSP430 allows the direct addressing of the entire 64K memory bytes without paging of the memory. This feature greatly simplifies the development of software.

8.3 Effects and Advantages of the MSP430 Architecture

The reasons for the popularity of the MSP430 instruction set (and by it its architecture) shown in Section 8.2, have effects and advantages that also result in money saved. These effects and advantages are shown and explained in the following.

8.3.1 Less Program Space

The direct access to any address, without loading of the previous address into a register, together with the constant generator results in program lengths for a given task that are between 55% and 90% compared to other microcomputers. This means that with the MSP430, a 4K version may be sufficient where with other microcomputers a 6K or 8K version is needed.

8.3.2 Shorter Programs

Any necessary code line is a source of errors. The less code lines that are necessary for a given task, the simpler a program is to read, understand, and service. The MSP430 needs between 33% and 55% of the code lines compared to its competition's products. The reason for this is the same as described previously. Any address can be accessed directly and that both for the source operand and for the destination operand. It is not necessary to create troublesome 16-bit addresses, handle the operands byte-wise, and store the final result afterwards indirectly via a composed destination address. All this happens with only one MSP430 instruction.

8.3.3 Reduced Development Time

The clearly smaller program length and the less troublesome access to ROM, RAM, and other peripherals reduce the necessary development time. In addition to that advantage, the considerations omit completely how the actual problem can be solved at all with the given architecture. A part that can take up to one third of the development time with other architectures. (Whoever has developed with 4-bit microcomputers knows what is meant).

8.3.4 Effective Code Without Compressing

The clear assembler language of the MSP430 allows, from the start, the writing of dense and legible code. If the developed program is well prepared and coded clearly, it is nearly impossible to reduce the program length seriously afterwards by compressing. This is no disadvantage, it simply means that optimized code was developed from the start.

8.3.5 Optimum C Code

The C compiler of a microcomputer can use only the instructions that have a regular structure. Typical CISC (complex instruction set computer) instructions, which normally show strong addressing mode restrictions, are not used by the compilers. Instead, the compilers emulate the complex instructions with several of the simple instructions, resulting in a use of only 30% (!) of the implemented instructions.

This is completely different with the MSP430. As the instructions (apart from the executed operation) are completely uniform, 100% of them are used by the compiler and not just 30%. As logical and arithmetical operations are executed directly and not by composed instructions, the execution time of the compiled code is shorter and less memory space is needed. Therefore, the same advantages that are valid for assembler programming are valid also for high-level language programming.

8.4 The MSP430 Instruction Set

In the following are all implemented and emulated instructions.

Description of the used abbreviations:

@	Logical NOT-Function
*	Status Bit is affected by the instruction
−	Status Bit is not affected by the instruction
0	Status Bit is cleared by the instruction
1	Status Bit is set by the instruction
V	Overflow Bit in Status Register SR
N	Negative Bit in Status Register SR
Z	Zero Bit in Status Register SR
C	Carry Bit in Status Register SR
src	Source Operand with 7 Addressing Modes
dst	Destination Operand with 4 Addressing Modes
xx.B	Byte Operation of the Instruction xx
Label	Label of the source or destination

8.4.1 Implemented Instructions

The instructions that are implemented in the CPU follow.

8.4.1.1 Two Operand Instructions

				Status Bits V	N	Z	C
ADD	ADD.B	src,dst	Add src to dst	*	*	*	*
ADDC	ADDC.B	src,dst	Add src + Carry to dst	*	*	*	*
AND	AND.B	src,dst	src .and. dst → dst	0	*	* @Z	
BIC	BIC.B	src,dst	@src .and. dst → dst	−	−	−	−
BIS	BIS.B	src,dst	src .or. dst → dst	−	−	−	−
BIT	BIT.B	src,dst	src .and. dst → SR	0	*	* @Z	
CMP	CMP.B	src,dst	Compare src and dst (dst − src)	*	*	*	*
DADD	DADD.B	src,dst	Add src + Carry to dst (dec.)	*	*	*	*
MOV	MOV.B	src,dst	Copy src to dst	−	−	−	−
SUB	SUB.B	src,dst	Subtract src from dst	*	*	*	*
SUBC	SUBC.B	src,dst	Subtract src with Carry from dst	*	*	*	*
XOR	XOR.B	src,dst	src .xor. dst → dst	*	*	* @Z	

8.4.1.2 Single Operand Instructions

The operand (src or dst) can be addressed with all seven addressing modes.

				Status Bits			
				V	N	Z	C
CALL		dst	Subroutine call	–	–	–	–
PUSH	PUSH.B	src	Copy operand onto stack	–	–	–	–
RETI			Interrupt return	*	*	*	*
RRA	RRA.B	dst	Rotate dst right arithmetically	0	*	*	*
RRC	RRC.B	dst	Rotate dst right through Carry	*	*	*	*
SWPB		dst	Swap bytes	–	–	–	–
SXT		dst	Sign extension into high byte	0	*	* @Z	

8.4.1.3 Conditional Jumps

The status bits are not affected by the jumps. With the signed jumps (JGE, JLT), the overflow bit is evaluated also, so that the jumps are executed correctly even in the case of overflow. Some jumps are the same (JC/JHS, JZ/JEQ, JNC/JLO, JNE/JNZ) but two mnemonics are used to get a better understanding of the program code. In case of a comparison JHS gives a better understanding of the code than JC.

JC	Label	Jump if Carry = 1
JHS	Label	Jump if dst is higher or same than src (C = 1)
JEQ	Label	Jump if dst equals src (Z = 1)
JZ	Label	Jump if Zero Bit = 1
JGE	Label	Jump if dst is greater than or equal to src (N .xor. V = 0)
JLT	Label	Jump if dst is less than src (N .xor. V = 1)
JMP	Label	Jump unconditionally
JN	Label	Jump if Negative Bit = 1
JNC	Label	Jump if Carry = 0
JLO	Label	Jump if dst is lower than src (C = 0)
JNE	Label	Jump if dst is not equal to src (Z = 0)
JNZ	Label	Jump if Zero Bit = 0

8.4.2 Emulated Instructions

The emulated instructions use implemented instructions together with constants coming out of the constant generator.

				Status Bits			
				V	N	Z	C
ADC	ADC.B	dst	Add Carry to dst	*	*	*	*
BR		dst	Branch indirect dst	–	–	–	–
CLR	CLR.B	dst	Clear dst	–	–	–	–
CLRC			Clear Carry Bit	–	–	–	0
CLRN			Clear Negative Bit	–	0	–	–
CLRZ			Clear Zero Bit	–	–	0	–
DADC	DADC.B	dst	Add Carry to dst (decimally)	*	*	*	*
DEC	DEC.B	dst	Decrement dst by 1	*	*	*	*
DECD	DECD.B	dst	dst − 2 → dst	*	*	*	*
DINT			Disable interrupts	–	–	–	–
EINT			Enable interrupts	–	–	–	–
INC	INC.B	dst	Increment dst by 1	*	*	*	*
INCD	INCD.B	dst	dst + 2 → dst	*	*	*	*
INV	INV.B	dst	Invert dst	*	*	* @Z	
NOP			No operation	–	–	–	–
POP	POP.B	dst	Pop top of stack to dst	–	–	–	–
RET			Subroutine return	–	–	–	–
RLA	RLA.B	dst	Rotate left dst arithmetically	*	*	*	*
RLC	RLC.B	dst	Rotate left dst through Carry	*	*	*	*
SBC	SBC.B	dst	Subtract Carry from dst	*	*	*	*
SETC			Set Carry Bit	–	–	–	1
SETN			Set Negative Bit	–	1	–	–
SETZ			Set Zero Bit	–	–	1	–
TST	TST.B	dst	Test dst	0	*	*	1

8.5 Benefits

The specification for the architecture of the MSP430 CPU contains the following requirements in order of importance:

1) High processing speed
2) Small CPU area on-chip
3) High ROM efficiency
4) Easy software development
5) Usable into the future
6) High flexibility
7) Usable for modern programming techniques

The following shows the finding of the optimum architecture out of the previous list of priorities. Several of the listed solutions affect more than one item of the list of priorities; these are shown at the item where they have the biggest impact.

8.5.1 High Processing Speed

To increase the processing speed to a multiple of the speed of 4-bit or 8-bit microcomputers, software and hardware related attributes were chosen.

Hardware related attributes

❑ Using 16-bit words, the analog-to-digital converter result can be processed immediately. Two operands (source and destination) are possible in one instruction.

❑ No microcoding: every instruction is decoded separately and allows one-cycle instructions. This is the case for register-to-register addressing, the normal addressing mode used for time critical software parts.

❑ Interrupt capability for anyone of the 8 I/O-Ports: The periodical polling of the inputs is not necessary.

❑ Vectored interrupts: This allows the fastest reaction to interrupts.

Software related attributes

❑ Implementation of the constant generator: The six most often used constants (–1, 0, 1, 2, 4, 8) are retrieved from the CPU with the highest possible speed.

❑ High register count: Twelve commonly usable registers allow the storage of all time critical values to achieve the fastest possible access.

8.5.2 Small CPU Area

To get low overall cost for the MSP430, the smallest CPU without limiting its processing capability was achieved:

❑ Use of a RISC structure: With few but strong instructions, any algorithm can be processed. Together with the constant generator, all commonly used instructions, not contained in the implemented instructions, are executable.

❑ Use of 100% orthogonality: Every instruction inside one of the three instruction formats is completely similar to the other ones. This results in a strongly simplified CPU.

❑ Only three instruction formats: Restriction to dual operand instructions, single operand instructions, and conditional jumps.

8.5.3 High ROM Efficiency

To solve a given task with a small ROM, the following steps were taken:

❑ Implementation of seven addressing modes: The possibility to select out of seven addressing modes for the source and out of four addressing modes for the destination allows direct access to all operands without any intermediate operations necessary.

❑ Placing of PC, SP, and SR inside of the register set: The possibility to address these as registers saves ROM space.

❑ Wide reach of the conditional jumps: Due to the eightfold jump distance of the MSP430 compared to other microcomputers, in most cases a branch instruction, that normally needs two words, is not necessary.

❑ Use of a byte/word structure: ROM and RAM are addressable both as bytes and as words. This allows the selection of the most favorable format.

8.5.4 Easy Software Development

Nearly all of the previously mentioned attributes of the architecture ease the development of software for the MSP430.

8.5.5 Usability on Into the Future

The chosen *von-Neumann*-architecture allows a simple system expansion far beyond the currently addressable 64K bytes. If necessary, memory expansion up to 16M bytes is possible.

8.5.6 Flexibility of the Architecture

To ensure that all intended peripheral modules, including currently unknown ones, can be connected easily to the MSP430 system. The following definitions were made:

❑ Placing of the peripheral control registers into the memory space (memory mapped I/O). The use of the normal instructions for the peripheral modules makes special peripheral instructions superfluous.

❑ All of the control registers and data registers of the peripheral modules can be read and written to.

8.5.7 Usable for Modern Programming Techniques

Programming techniques like *position independent coding* (PIC), *reentrant coding*, *recursive coding*, or the use of high-level languages like C force the implementation of a stack pointer. The system SP is therefore implemented as a CPU register.

8.6 Conclusion

This section demonstrates that the instruction set and the architecture of the MSP430 are easy to understand and that it is easy too to write software for the MSP430. Everyone who has written large program parts with the MSP430 assembler language has an antipathy to adapt again to the more or less unstructured architectures of the other 4-bit, 8-bit, and 16-bit microcomputers.

CPU Registers

9.1 CPU Registers

All of the MSP430 CPU registers can be used with all instructions.

9.1.1 The Program Counter PC

One of the main differences from other microcomputer architectures relates to the Program Counter (CPU register R0) that can be used as a normal register with the MSP430. This means that all of the instructions and addressing modes can be used with the Program Counter too. A branch, for example, is made by simply moving an address into the PC:

```
MOV    #LABEL,PC    ; Branch to address LABEL
MOV    LABEL,PC     ; Branch to the address contained in address LABEL
MOV    @R14,PC      ; Branch indirect, indirect R14
```

> **Note:**
>
> The Program Counter always points to even addresses. This means that the LSB is always zero. The software has to ensure that no odd addresses are used if the Program Counter is involved. Odd PC addresses will result in non-predictable behavior.

9.1.2 Stack Processing

9.1.2.1 Use of the System Stack Pointer (SP)

The system stack pointer (CPU register R1) is a normal register like the others. This means it can use the same addressing modes. This gives good access to all items on the stack, not only to the one on the top of the stack.

The system stack pointer (SP) is used for the storage of the following items:

❑ Interrupt return addresses and status register contents
❑ Subroutine return addresses
❑ Intermediate results
❑ Variables for subroutines, floating point package etc.

When using the system stack, remember that the microcomputer hardware also uses the stack pointer for interrupts and subroutine calls. To ensure the error-free running of the program it is necessary to do exact *housekeeping* for the system stack.

> **Note:**
>
> The Stack Pointer always points to even addresses. This means the LSB is always zero. The software has to ensure that no odd addresses are used if the Stack Pointer is involved. Odd SP addresses will end up in non-predictable results.

If bytes are pushed on the system stack, only the lower byte is used, the upper byte is not modified.

```
PUSH      #05h         ; 0005h -> TOS
PUSH.B    #05h         ; xx05h -> TOS
MOV.B     1(SP),R5     ; Address odd byte
```

9.1.2.2 Software Stacks

Every register from R4 to R15 can be used as a software stack pointer. This allows independent stacks for jobs that have a need for this. Every part of the RAM can be used for these software stacks.

EXAMPLE: R4 is to be used as a software stack pointer.

```
MOV    #SW_STACK,R4  ; Init. SW stack pointer
...
DECD   R4            ; Decrement stack pointer
MOV    item,0(R4)    ; Push item on stack
...                  ; Proceed
MOV    @R4+,item2    ; Pop item from stack
```

Software stacks can be organized as byte stacks also. This is not possible for the system stack, which always uses 16-bit words. The example shows R4 used as a byte stack pointer:

```
MOV #SW_STACK,R4     ; Init. SW stack pointer
...
DEC R4               ; Decrement stack pointer
MOV.B          item,0(R4)   ; Push item on stack
...                  ; Proceed
MOV.B @R4+,item2     ; Pop item from stack
```

9.1.3 Byte and Word Handling

Every memory word is addressable by three addresses as shown in the Figure 9–1:

❑ The word address: An even address N
❑ The lower byte address: An even address N
❑ The upper byte address: An odd address N+1

If byte addressing is used, only the addressed byte is affected. No carry or overflow can affect the other byte.

Note:

Registers R0 to R15 do not have an address but are treated in a special way. Byte addressing always uses the lower byte of the register. The upper byte is set to zero if the instruction modifies the destination. Therefore, all instructions clear the upper byte of a destination register except CMP.B, TST.B, BIT.B and PUSH.B. The source is never affected.

The way an instruction treats data is defined with its extension:

❑ The extension .B means byte handling
❑ The extension .W (or none) means word handling

EXAMPLES: The next two software lines are equivalent. The 16-bit values read in absolute address 050h are added to the value in R5.

```
ADD     &050h,R5        ; ADD 16-BIT VALUE TO R5

ADD.W   &050h,R5        ; ADD 16-BIT VALUE TO R5
```

The 8-bit value read in the lower byte of absolute address 050h is added to the value contained in the lower byte of R5. The upper byte of R5 is set to zero.

```
ADD.B   &050h,R5        ; ADD 8-BIT VALUE TO R5
```

Bit   15   8 7   0

Upper Byte	Lower Byte

Odd Address N+1   Even Address N

Word Address N
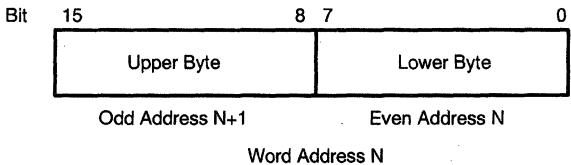
Figure 9–1. Word and Byte Configuration

If registers are used with byte instructions the upper byte of the destination register is set to zero for all instructions except CMP.B, TST.B, BIT.B and PUSH.B. It is therefore necessary to use word instructions if the range of calculations can exceed the byte range.

EXAMPLE: The two signed bytes OP1 and OP2 have to be added together and the result stored in word OP3.

```
MOV.B   OP1,R4         ; Fetch 1st operand
SXT     R4             ; Change to word format
MOV.B   OP2,OP3        ; Fetch 2nd operand
SXT     OP3            ; Change to word format
ADD.W   R4,OP3         ; 16-bit result to OP3
```

9.1.4 Constant Generator

A statistical look at the numbers used with the Immediate Mode shows that a few small numbers are in use most often. The six most often used numbers can be addressed with the four addressing modes of R3 (constant generator 2) and with the two not usable addressing modes of R2 (status register). The six constants that do not need an additional 16-bit word when used with the immediate mode are:

Table 9–1. Constant Generator

NUMBER	EXPLANATION	HEXADECIMAL	REGISTER	FIELD AD
+0	Zero	0000h	R3	00
+1	Positive one	0001h	R3	01
+2	Positive two	0002h	R3	10
+4	Positive four	0004h	R2	10
+8	Positive eight	0008h	R2	11
−1	Negative one	FFFFh	R3	11

The assembler inserts these ROM-saving addressing modes automatically when one of the previously described immediate constants is encountered. But, only immediate constants are replaceable this way, not (for example) index values.

If an immediate constant out of the constant generator is used, the execution time is equal to the execution time of the register mode.

The most often used bits of the peripheral registers are located in the bits addressable by the constant generator whenever possible.

9.1.5 Addressing

The MSP430 allows seven addressing modes for the source operand and four addressing modes for the destination. The addressing modes used are:

Table 9–2. Addressing Modes

ADDRESS BITS		SOURCE MODES	DESTINATION MODES	EXAMPLE
src	dst			
00	0	Register	Register	R5
01	1	Indexed	Indexed	TAB(R5)
01	1	Symbolic	Symbolic	TABLE
01	1	Absolute	Absolute	&BTCTL
10	–	Indirect	–	@R5
11	–	Indirect autoincrement	–	@R5+
11	–	Immediate	–	#TABLE

The three missing addressing modes for the destination operand are not of much concern for the programming. The reason is:

Immediate Mode: Not necessary for the destination; immediate operands can always be placed into the source. Only in a very few cases it is necessary to have two immediate operands in one instruction

Indirect Mode: If necessary, the Indexed Mode with an index of zero is usable. For example:

```
ADD    #16,0(R6)        ; @R6 + 16 -> @R6
CMP    R5,0(SP)         ; R5 equal to TOS?
```

The second previously shown example can be written in the following way, saving 2 bytes of ROM:

```
CMP    @SP,R5           ; R5 equal to TOS? (R5-TOS)
```

Indirect Autoincrement Mode: With table processing, a method that saves ROM space and reduces the number of used registers to one can be used:

EXAMPLE: The content of TAB1 is to be written into TAB2. TAB1 ends at the word preceding TAB1END.

```
        MOV    #TAB1,R5                ; Initialize pointer
LOOP    MOV.B  @R5+,TAB2-TAB1-1(R5)    ; Move TAB1 -> TAB2
        CMP    #TAB1END,R5             ; End of TAB1 reached?
        JNE    LOOP                    ; No, proceed
...                                    ; Yes, finished
```

The previous example uses only one register instead of two and saves three words due to the smaller initialization part. The normally written, longer loop is shown in the following

```
        MOV    #TAB1,R5         ;Initialize pointers
        MOV    #TAB2,R6
LOOP    MOV.B  @R5+,0(R6)       ;Move TAB1 -> TAB2
        INC    R6
        CMP    #TAB1END,R5      ;End of TAB1 reached?
        JNE    LOOP             ;No, proceed
...                             ;Yes, finished
```

In other cases it can be possible to exchange source and destination operands to have the auto increment feature available for a pointer.

Each of the seven addressing modes has its own features and advantages:

Register Mode: Fastest mode, least ROM requirements

Indexed Mode: Random access to tables

Symbolic Mode: Access to random addresses without overhead by loading of pointers

Absolute Mode: Access to absolute addresses independent of the current program address

Indirect Mode: Table addressing via register; code saving access to often referenced addresses

Indirect Autoincrement Mode: Table addressing with code saving automatic stepping; for transfer routines

Immediate Mode: Loading of pointers, addresses or constants within the instruction,

With the use of the symbolic mode an interrupt routine can be as short as possible. An interrupt routine is shown that has to increment a RAM word COUNTER and to do a comparison if a status byte STATUS has reached the value 5. If this is the case, the status byte is cleared. Otherwise, the interrupt routine terminates:

```
INTRPT INC    COUNTER      ;Increment counter
       CMP.B  #5,STATUS    ;STATUS = 5?
       JNE    INTRET       ;
       CLR.B  STATUS       ;STATUS = 5: clear it
INTRET RETI
```

No loading of pointers or saving and restoring of registers is necessary. The action is done immediately, without any overhead.

9.1.6 Program Flow Control

9.1.6.1 Computed Branches and Calls

The branch instruction is an emulated instruction that moves the destination address into the program counter:

```
MOV    dst,PC       ; EMULATION FOR BR @dst
```

The ability to access the program counter in the same way as all other registers provides interesting options:

1) The destination address can be taken from tables: see Section 9.2.5
2) The destination address can be calculated
3) The destination address can be a constant. This is the usual method of getting the address.

9.1.6.2 Nesting of Subroutines

Due to the stack orientation of the MSP430, one of the main problems of other architectures does not play a role here at all. Subroutine nesting can proceed as long as RAM is available. There is no need to keep track of the subroutine calls as long as all subroutines terminate with the *Return from Subroutine* in-

struction RET. If subroutines are left without the RET instruction, some house-keeping is necessary; popping of the return address or addresses from the stack.

9.1.6.3 Nesting of Interrupts

Nesting of interrupts gives no problem at all, provided there is enough RAM for the stack. For every occurring interrupt, two words on the stack are needed for the storage of the status register and the return address. To enable nested interrupts, it is necessary to only include an EINT instruction into the interrupt handler. If the interrupt handlers are as short as possible (a good real-time practice), nesting may not be necessary.

EXAMPLE: The basic timer interrupt handler is woken-up with 1 Hz only, but has to do a lot of things. The interrupt nesting is therefore used. The latency time is 8 clock cycles only.

```
; Interrupt handler for Basic Timer: Wake-up with 1Hz
;
BT_HAN    EINT                ; Enable interrupt for nesting
          INC.B  SECCNT       ; Counter for seconds +1
          CMP.B  #60,SECCNT   ; 1 minute elapsed?
          JHS    MIN1         ; Yes, do necessary tasks
          RETI                ; No return to LPM3
;
; One minute elapsed: Return is removed from stack, a branch to
; the necessary tasks is made. There it is decided how to proceed
;
MIN1   INC    MINCNT       ; Minute counter +1
       CLR    SECCNT       ; 0 -> SECCNT
       ...                 ; Start of necessary tasks
       RETI                ; Tasks completed
```

9.1.6.4 Jumps

Jumps allow the conditional or unconditional leaving of the linear program flow. Jumps cannot reach every address of the address space. But they have the advantage of needing only one word and only two MCLK cycles. The 10-bit offset field allows jumps of 512 words maximum forward and 511 words, maximum, backwards. This is four to eight times the normal reach of a jump. Only in a few cases, the 2-word branch is necessary.

Eight Jumps are possible with the MSP430; four of them have two mnemonics to allow better readability:

Table 9–3. Jump Usage

MNEMONIC	CONDITION	APPLICATIONS
JMP label	Unconditional Jump	Program control transfer
JEQ label	Jump if Z = 1	After comparisons: src = dst
JZ label	Jump if Z = 1	Test for zero contents
JNE label	Jump if Z = 0	After comparisons: src # dst
JNZ label	Jump if Z = 0	Test for nonzero contents
JHS label	Jump if C = 1	After unsigned comparisons: dst ≥ src
JC label	Jump if C = 1	Test for a set carry
JLO label	Jump if C = 0	After unsigned comparisons dst < src
JNC label	Jump if C = 0	Test for a reset carry
JGE label	Jump if N .XOR. V = 0	After signed comparisons: dst ≥ src
JLT label	Jump if N .XOR. V = 1	After signed comparisons: dst < src
JN label	Jump if N = 1	Test for the sign of a result: dst < 0

Note:

It is important to use the appropriate conditional jump for signed and unsigned data. For positive data (0 to 07FFFh or 0 to 07Fh) both signed and unsigned conditional jumps operate similarly. This changes completely when used with negative data (08000h to 0FFFFh or 080h to 0FFh): the signed conditional jumps treat negative data as smaller numbers than the positive ones, and the unsigned conditional jumps treat them as larger numbers than the positive ones.

No *Jump if Positive* is provided, only a *Jump if Negative*. But after several instructions, it is possible to use the *Jump if Greater Than or Equal* for this purpose. It must be ensured that only the instruction preceding the JGE resets the overflow bit V. The following instructions ensure this:

```
AND    src,dst      ; V <- 0
BIT    src,dst      ; V <- 0
RRA    dst          ; V <- 0
SXT    dst          ; V <- 0
TST    dst          ; V <- 0
```

If this feature is used, it should be noted within the comment for later software modifications. For example:

```
MOV    ITEM,R7      ; FETCH ITEM
TST    R7           ; V <- 0, ITEM POSITIVE?
JGE    ITEMPOS      ; V=0: JUMP IF >= 0
```

> **Note:**
>
> If addresses are computed only the unsigned jumps are adequate. Addresses are always unsigned, positive numbers.

No *Jump if Overflow* is provided. If the status of the overflow bit is needed from the software, a simple bit test can be used (the BIT instruction clears the overflow bit, but its state is read correctly before):

```
OV      .EQU    0100h       ; Bit address in SR
                            ;
        BIT     #OV,SR      ; Test Overflow Bit and clear it
        JNZ     OVFL        ; If OV = 1 branch to label OVFL
    ...                     ; If OV = 0 continue here
```

9.2 Special Coding Techniques

The flexibility of the MSP430 CPU together with a powerful assembler allows coding techniques not available with other microcomputers. The most important ones are explained in the following sections.

9.2.1 Conditional Assembly

For a detailed description of the syntax please refer to *MSP430 Family Assembler Language Tools User's Guide.*

Conditional assembly provides the ability to compile different lines of source into the object file depending on the value of an expression that is defined in the source program. This makes it easy to alter the behavior of the code by modifying one single line in the source.

The following example shows how to use of conditional assembly. The example allows easy debugging of a program that processes input from the ADC by pretending that the input of the ADC is always 07FFh. The following is the routine used for reading the input of the ADC. It returns the value read from ADC input A0 in R8.

```
DEBUG .set  1                     ;1= debugging mode; 0= normal mode
ACTL  .set  0114h
ADAT  .set  0118h
IFG2  .set  3
ADIFG .set  4

; get_ADC_value:
;
        .IF    DEBUG=1
        MOV    #07FFh,R8
        .ELSE
        BIC    #60,&ACTL          ; Input channel is A0
        BIC.B  #ADIFG,&IFG2
        BIS    #1,&ACTL           ; Start conversion
WAIT    BIT.B  #ADIFG,&IFG2
        JZ     WAIT               ; Wait until conversion is ready
        MOV    &ADAT,R8
        .ENDIF
        RET
;
```

With a little further refining of the code, better results can be achieved. The following piece of code shows more built-in ways to debug the written code. The second *debug code*, where debug=2, returns 0700h and 0800h alternating.

```
DEBUG .SET  1                      ; 1= debug mode 1; 2= deb. mode 2; 0=
                                   ; normal mode
ACTL  .SET        0114h
ADAT  .SET        0118h
IFG2  .SET        3
ADIFG .SET        4

; get_ADC_value:
;
VAR   .SECT       "VAR"'0200h
OSC   .WORD.      0700h

      .IF         DEBUG=1    ; Return a   constant value
      MOV         #07FFh,R8
      .ELSEIF     DEBUG=2    ; Return alternating values
      MOV         #0F00h,R8
      SUB         OSC,R8
      MOV         R8,OSC
      .ELSE
      BIC         #60h,&ACTL  ; Input channel is A0
      BIC         #ADIFG,&IFG2
      BIS         #1,&ACTL    ; Start conversion
WAIT  BIT         #ADIFG,&IFG2
      JZ          WAIT        ; Wait until conversion is ready
      MOV         &ADAT,R8
      .ENDIF
      RET
```

Conditional assembly is not restricted to the debug phase of software develop-ment. The main use is normally to get different software versions out of one source. For every version only the necessary software parts are assembled and the unneeded parts are left out by conditional assembly. The big advan-tage is the single source that is maintained.

An example of this is the MSP430 floating point package with two different number lengths (32 and 48 bits) contained in one source. Before assembly the desired length is defined by an .EQU directive. See Section 5.6, *The Floating Point Package* for details.

9.2.2 Position Independent Code

The architecture of the MSP430 allows the easy implementation of position independent code (PIC). This is a code, which may run anywhere in the ad-dress space of a computer without any relocation needed. PIC is possible with the MSP430 because of the allocation of the PC inside of the register bank. The addressability of the PC is often used. Links to other PIC blocks are pos-sible only by references to absolute addresses (pointers).

EXAMPLE: Code is transferred to the RAM from an outside storage (EPROM, ROM, or EEPROM) and executed there at full speed. This code needs to be PIC. The loaded code may have several purposes:

❑ Application specific software that is different for some versions
❑ Additional code that was not anticipated before mask generation
❑ Test routines for manufacturing purposes

9.2.2.1 Referencing of Code Inside Position Independent Code

The referenced code or data is located in the same block of PIC as the program resides.

Jumps

Jumps are position independent anyway: their address information is an offset to the destination address.

Branches

```
ADD    @PC,PC            ; Branch to label DESTINATION
.WORD DESTINATION-$      ; Address pointer
```

Subroutine Calls

```
; Calling a subroutine starting at the label SUBR:
;
SC   MOV  PC,Rn           ; Address SC+2 -> Rn
     ADD  #SUBR-$,Rn      ; Add offset (SUBR - (SC+2))
     CALL Rn              ; SC+2+SUBR-(SC+2)) = SUBR
```

Operations on Data

The symbolic addressing mode is position independent. An offset to the PC is used. No special addressing is necessary

```
MOV    DATA,Rn           ; DATA is addressed
CMP    DATA1,DATA2       ; symbolically
```

Jump Tables

The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: +512 words, –511 words

```
ADD    Rstatus,PC        ; Rstatus = (2x status)
JMP    STATUS0           ; Code for status = 0
JMP    STATUS2           ; Code for status = 2
...
JMP    STATUSn           ; Code for status = 2n
```

Branch Tables

The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: complete 64K

```
        ADD    TABLE(Rstatus),PC        ; Rstatus = status
  TABLE .WORD STATUS0-TABLE             ; Offset for status = 0

        .WORD STATUS2-TABLE             ; Offset for status = 2
        ...
        .WORD STATUSn-TABLE             ; Offset for status = 2n
```

9.2.2.2 *Referencing of Code Outside of PIC (Absolute)*

The referenced code or data is located outside the block of PIC. These addresses can be absolute addresses only (e.g. for linking to other blocks or peripheral addresses).

Branches

Branching to the absolute address DESTINATION:

```
  BR    #DESTINATION            ; #DESTINATION -> PC
```

Subroutine Calls

Calling a subroutine starting at the absolute address SUBR:

```
  CALL  #SUBR                   ; #SUBR -> PC
```

Operations on Data

Absolute mode (indexed mode with status register SR = 0). SR does not loose its information!

```
  CMP   &DATA1,&DATA2           ; DATA1 + 0 = DATA1
  ADD   &DATA1,Rn
  PUSH  &DATA2                  ; DATA2 -> stack
```

Branch Tables

The status contained in Rstatus decides where the SW continues. Rstatus steps in increments of 2. The table is located in absolute address space:

```
        MOV   TABLE(Rstatus),PC ; Rstatus = status
        ...
        .sect xxx               ; Table in absolute address space
  TABLE .WORD STATUS0           ; Code for status = 0
```

```
          .WORD  STATUS2            ; Code for status = 2
          . . .
          .WORD  STATUSn            ; Code for status = 2n
```

Table is located in PIC address space, but addresses are absolute:

```
          MOV    Rstatus,Rhelp      ; Rstatus contains status
          ADD    PC,Rhelp           ; Status + L$1 -> Rhelp
L$1       ADD    #TABLE-L$1,Rhelp   ; Status+L$1+TABLE-L$1
          MOV    @Rhelp,PC          ; Computed address to PC
TABLE .WORD  STATUS0            ; Code for status = 0
          .WORD  STATUS1            ; Code for status = 2
          . . .
          .WORD  STATUSn            ; Code for status = 2n
```

The previously shown program examples can be implemented as MACROs if needed. This would ease the usage and enhance the legibility.

9.2.3 Reentrant Code

If the same subroutine is used by the background program and interrupt routines, then two copies of this subroutine are necessary with normal computer architectures. The stack gives a method of programming that allows many tasks to use a single copy of the same routine. This ability of sharing a subroutine for several tasks is called reentrancy.

Reentrancy allows the calling of a subroutine despite the fact that the current task has not yet finished using the subroutine.

The main difference of a reentrant subroutine from a normal one is that the reentrant routine contains only pure code. That is, no part of the routine is modified during the usage. The linkage between the routine itself and the calling software is possible only via the stack (i.e. all arguments during calling and all results after completion have to be placed on the stack and retrieved from there). The following conditions must be met for reentrant code:

☐   No usage of dedicated RAM; only stack usage
☐   If registers are used, they need to be saved on the stack and restored from there.

EXAMPLE: A conversion subroutine *Binary to BCD* needs to be called from the background and the interrupt part. The subroutine reads the input number from TOS and places the 5-digit result also on TOS (two words). The subroutines save all registers used on the stack and restore them from there or compute directly on the stack.

```
          PUSH  R7              ; R7 CONTAINS THE BINARY VALUE
          CALL  #BINBCD         ; TO BE CONVERTED TO BCD
          MOV   @SP+,LSD        ; BCD-LSDs FROM STACK
          MOV   @SP+,MSD        ; BCD-MSD  FROM STACK
          . . .
```

9.2.4 Recursive Code

Recursive subroutines are subroutines that call themselves. This is not pos-sible with typical architectures; stack processing is necessary for this often used feature. A simple example for recursive code is a line printer handler that calls itself for the inserting of a *form feed* after a certain number of printed lines. This self-calling allows the use all of the existent checks and features of the handler without the need to write it more than once. The following conditions must be met for recursive code:

❏ No use of dedicated RAM; only stack usage

❏ A termination item must exist to avoid infinite nesting (e.g., the lines per page must be greater than 1 with the above line printer example)

❏ If registers are used, they need to be saved and restored on the stack

EXAMPLE: The line printer handler inserts a form feed after 70 printed lines

```
;
LPHAND       PUSH  R4             ; Save R4
             . . .
             CMP   #70,LINES      ; 70 lines printed?
             JLT   L$500          ; No, proceed
             CALL  #LPHAND        ;
             .BYTE CR,FF          ; Yes, output Carriage Return
             . . .                ; and Form Feed
L$500        . . .
```

9.2.5 Flag Replacement by Status Usage

Flags have several disadvantages when used for program control:

❏ Missing transparency (flags may depend on other flags)
❏ Possibility of nonexistent flag combinations if not handled very carefully
❏ Slow speed: the flags can be tested only serially

The MSP430 allows the use of a status (contained in a RAM byte or register), which defines the current program part to be used. This status is very descrip-

tive and prohibits nonexistent combinations. A second advantage is the high speed of the decision. Only one instruction is needed to get to the start of the appropriate handler (see Branch Tables).

The program parts that are used currently define the new status dependent on the actual conditions. Normally the status is only incremented, but it can be changed to be more random too.

EXAMPLE: The status contained in register Rstatus decides where the software continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n)

```
; Range: Complete 64K
;
        MOV    TABLE(Rstatus),PC ;Rstatus = status
TABLE .WORD STATUS0            ; Address handler for status = 0

        .WORD STATUS2           ; Address handler for status = 2
        ...
        .WORD STATUSn           ; Address handler for status = 2n
;
STATUS0     ....                ; Start handler status 0
        INCD  Rstatus           ; Next status is 2
        JMP   HEND              ; Common end
```

The previous solution has the disadvantage of using words even if the distances to the different program parts are small. The next example shows the use of bytes for the branch table. The SXT instruction allows backward references (handler starts at lower addresses than TABLE4).

```
; BRANCH TABLES WITH BYTES: Status in R5 (0, 1, 2, ..n)
; Usable range: TABLE4-128 to TABLE4+126

        PUSH.B    TABLE4(R5)       ; STATUSx-TABLE4 -> STACK
        SXT       @SP      ..... ; Forward/backward references
        ADD       @SP+,PC          ; TABLE4+STATUSx-TABLE4 -> PC
TABLE4    .BYTE     STATUS0-TABLE4   ; DIFFERENCE TO START OF
                                       HANDLER
        .BYTE     STATUS1-TABLE4
        ....
        .BYTE     STATUSn-TABLE4   ; Offset for status = n
```

If only forward references are possible (normal case), the addressing range can be doubled. The next example shows this:

```
; Stepping is forward only (with doubled forward range)
; Status is contained in R5 (0, 1, ..n)
```

```
; Usable range: TABLE5 to TABLE5+254


            PUSH.B      TABLE5(R5)   ; STATUSx-TABLE -> STACK
            CLR.B       1(SP)       ..... ; Hi byte <- 0
            ADD         @SP+,PC              ; TABLE+STATUSx-TABLE -> PC
TABLE5      .BYTE       STATUS0-TABLE5   ; DIFFERENCE TO START OF
                                              HANDLER
            .BYTE       STATUS1-TABLE5

            ....
            .BYTE       STATUSn-TABLE5   ; Offset for status = n
      ;
```

The previous example can be made shorter and faster if a register can be used:

```
; Stepping is forward only (with doubled forward range)
; Status is contained in R5 (0, 1, 2..n)
; Usable range: TABLE5 to TABLE5+254
;
            MOV.B       TABLE5(R5),R6    ; STATUSx-TABLE5 -> R6
            ADD         R6,PC       ..... ; TABLE5+STATUSx-TABLE5 -> PC
TABLE5      .BYTE       STATUS0-TABLE5   ; DIFFERENCE TO START OF
                                              HANDLER
            .BYTE       STATUS1-TABLE5

            ....
            .BYTE       STATUSn-TABLE5   ; Offset for status = n
```

The addressable range can be doubled once more with the following code. The status (0, 1, 2, ..n) is doubled before its use.

```
; The addressable range may be doubled with the following code:
; The "forward only" version with an available register (R6) is
; shown: Status 0, 1, 2 ...n
; Usable range: TABLE6 to TABLE6+510
;
            MOV.B TABLE6(R5),R6      ; (STATUSx-TABLE6)/2
            RLA   R6                 ; STATUSx-TABLE6
            ADD   R6,PC              ; TABLE6+STATUSx-TABLE6 -> PC
TABLE6      .BYTE (STATUS0-TABLE6)/2      ; Offset for Status = 0
            .BYTE (STATUS1-TABLE6)/2      ;
            ...
            .BYTE (STATUSn-TABLE6)/2      ; Offset for Status = n
```

9.2.6 Argument Transfer With Subroutine Calls

Subroutines often have arguments to work with. Several methods exist for the passing of these arguments to the subroutine:

❑ On the stack
❑ In the words (bytes) after the subroutine call
❑ In registers
❑ The address is contained in the word after the subroutine call

The passed information itself may be numbers, addresses, counter contents, upper and lower limits etc. It only depends on the application.

9.2.6.1   Arguments on the Stack

The arguments are pushed on the stack and read afterwards by the called sub-routine. The subroutine is responsible for the necessary housekeeping (here, the transfer of the return address to the top of the stack).

❑ Advantages:

■ Usable generally; no registers have to be freed for argument passing

■ Variable arguments are possible

❑ Disadvantages:

■ Overhead due to necessary housekeeping

■ Not easy to understand

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before being called. No information is given back and a normal return from subroutine is used.

```
        PUSH   argument0    ; 1st ARGUMENT FOR SUBROUTINE
        PUSH   argument1    ; 2nd ARGUMENT
        CALL   #SUBR        ; SUBROUTINE CALL
        . . .
SUBR    MOV    4(SP),Rx     ; COPY ARGUMENT0 TO Rx
        MOV    2(SP),Ry     ; COPY ARGUMENT1 TO Ry
        MOV    @SP,4(SP)    ; RETURN ADDRESS TO CORRECT LOC.
        ADD    #4,SP        ; PREPARE SP FOR NORMAL RETURN
        . . .               ; PROCESSING OF DATA
        RET                 ; NORMAL RETURN
```

After the subroutine call, the stack looks as follows: After the RET, it looks like this:

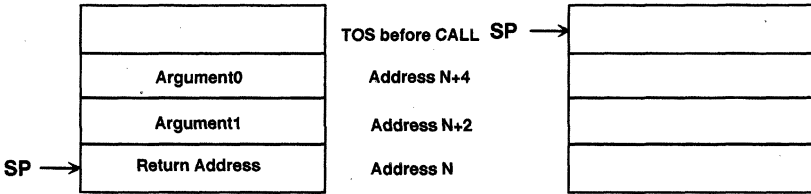|			
		TOS before CALL SP →	
Argument0		**Address N+4**	
Argument1		**Address N+2**	
SP → **Return Address**		**Address N**	

Figure 9–2. Argument Allocation on the Stack

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before being called. Three result words are returned on the stack. It is the responsibility of the calling program to pop the results from the stack.

```
          PUSH   argument0          ; 1st ARGUMENT FOR SUBROUTINE
          PUSH   argument1          ; 2nd ARGUMENT
          CALL   #SUBR              ; SUBROUTINE CALL
          POP    R15               ; RESULT2 -> R15
          POP    R14               ; RESULT1 -> R14
          POP    R13               ; RESULT0 -> R13
          . . .
   SUBR   MOV    4(SP),Rx          ; COPY ARGUMENT0 TO Rx
          MOV    2(SP),Ry          ; COPY ARGUMENT1 TO Ry
          . . .                    ; PROCESSING CONTINUES
          PUSH   2(SP)             ; SAVE RETURN ADDRESS
          MOV    RESULT0,6(SP)     ; 1st RESULT ON STACK
          MOV    RESULT1,4(SP)     ; 2nd RESULT ON STACK
          MOV    RESULT2,2(SP)     ; 3rd RESULT ON STACK
          RET
```

After the subroutine call, the stack looks as follows: After the RET, it looks like this:

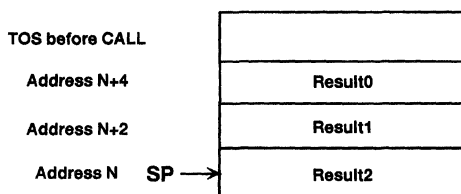|		
	TOS before CALL	
Argument0	**Address N+4**	**Result0**
Argument1	**Address N+2**	**Result1**
SP → **Return Address**	**Address N** SP →	**Result2**
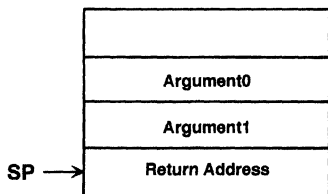
Figure 9–3. Argument and Result Allocation on the Stack

> **Note:**
>
> If the stack is involved during data transfers, it is very important to have in mind that only data at or above the top of stack (TOS, the word the SP points to) is protected against overwriting by enabled interrupts. This does not allow the SP to move above the last item on the stack. Indexed addressing is needed instead.

9.2.6.2 Arguments Following the Subroutine Call

The arguments follow the subroutine call and are read by the called subroutine. The subroutine is responsible for the necessary housekeeping (here, the adaptation of the return address on the stack to the 1st word after the arguments).

☐ Advantages:

■ Very clear and easily readable interface

☐ Disadvantages:

■ Overhead due to necessary housekeeping

■ Only fixed arguments possible

EXAMPLE: The subroutine SUBR gets its information from two arguments following the subroutine call. Information can be given back on the stack or in registers.

```
CALL   #SUBR              ; SUBROUTINE CALL
.WORD  START              ; START OF TABLE
.BYTE  24,0               ; LENGTH OF TABLE, FLAGS
```

```
        ...                     ; 1st instruction after CALL
SUBR    MOV     @SP,R5          ; COPY ADDRESS 1st ARGUMENT TO R5
        MOV     @R5+,R6         ; MOVE 1st ARGUMENT TO R6
        MOV     @R5+,R7         ; MOVE ARGUMENT BYTES TO R7
        MOV     R5,0(SP)        ; ADJUST RETURN ADDRESS ON STACK
        ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

9.2.6.3 Arguments In Registers

The arguments are moved into defined registers and used afterwards by the subroutine.

❏ Advantages:

■ Simple interface and easy to understand
■ Very fast
■ Variable arguments are possible

❏ Disadvantages:

■ Registers have to be freed

EXAMPLE: The subroutine SUBR gets its information from two registers which are loaded before the calling. Information can be given back, or not with the same registers.

```
        MOV     arg0,R5         ; 1st ARGUMENT FOR SUBROUTINE
        MOV     arg1,R6         ; 2nd ARGUMENT
        CALL    #SUBR           ; SUBROUTINE CALL
        ...
SUBR    ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

9.2.7 Interrupt Vectors in RAM

If the destination address of an interrupt changes with the program run, it is valuable to have the ability to modify the pointer. The vector itself (which resides in ROM) cannot be changed but a second pointer residing in RAM can be used for this purpose.

EXAMPLE: The interrupt handler for the basic timer starts at location BTHAN1 after initialization and at BTHAN2 when a certain condition is met (for example, when a calibration is made).

```
; BASIC TIMER INTERRUPT GOES TO ADDRESS BTVEC. THE INSTRUCTION
```

```
; "MOV @PC,PC" WRITES THE ADDRESS IN BTVEC+2 INTO THE PC:
; THE PROGRAM CONTINUES AT THAT ADDRESS
;
       .sect "VAR",0200h       ; RAM START
BTVEC .word 0                  ; OPCODE "MOV @PC,PC"
       .word 0                 ; ACTUAL HANDLER START ADDR.

; THE SOFTWARE VECTOR BTVEC IS INITIALIZED:
;
INIT   MOV    #04020h,BTVEC    ; OPCODE "MOV @PC,PC
       MOV    #BTHAN1,BTVEC+2  ; START WITH HANDLER BTHAN1
       ...                     ; INITIALIZATION CONTINUES
;
; THE CONDITION IS MET: THE BASIC TIMER INTERRUPT IS HANDLED
; AT ADDRESS BTHAN2 STARTING NOW

       MOV    #BTHAN2,BTVEC+2  ; CONT. WITH ANOTHER HANDLER
       ...
;
; THE INTERRUPT VECTOR FOR THE BASIC TIMER CONTAINS THE RAM
; ADDRESS OF THE SOFTWARE VECTOR BTVEC:

       .sect "BTVect",0FFE2h   ; VECTOR ADDRESS BASIC TIMER
       .WORD BTVEC             ; FETCH ACTUAL VECTOR THERE
```

9.3 Instruction Execution Cycles

9.3.1 Double Operand Instructions

With the following scheme, it is relatively easy to remember how many cycles a double operand instruction will need to execute. Figure 9–4 shows the number of cycles for all 28 possible combinations of the source and destination addressing modes. All similar addressing modes are condensed.

	Rdst	X(Rdst) SYMBOLIC &ABSOLUT
Rsrc	1†	4
@Rsrc, @Rsrc+, #N	2†	5
X(Rsrc), SYMBOLIC, &ABSOLUT	3	6

†: Add one cycle if Rdst is PC

Figure 9–4. Execution Cycles for Double Operand Instructions

EXAMPLE: the instruction `ADD #500h,16(R5)` needs 5 cycles for the execution.

9.3.2 Single Operand Instructions

The simple and clear scheme of the double operand instructions is not applicable to the six single operand instructions. They differ too much. Figure 9–5 gives an overview.

	SWPB SXT RRx	PUSH	CALL
Rdst	1	3	4
@Rdst	3	4	4
@Rdst+, #N	3	4	5
X(Rdst), SYMBOLIC, &ABSOLUT	4	5	5

Figure 9–5. Execution Cycles for Single Operand Instructions

EXAMPLE: the instruction `PUSH #500h` needs 4 cycles for the execution.

9.3.3 Jump Instructions

All seven conditional jump instructions need two cycles for execution, independent if the jump condition is met or not. The same is true for the unconditional jump instruction, JMP.

9.3.4 Interrupt Timing

An enabled interrupt sequence needs eleven cycles overhead:

❏ Six cycles for the storage of the PC and the SR on the stack until the first instruction of the interrupt handler is started

❏ Five cycles for the return from interrupt—by the instruction RETI—until the first instruction of the interrupted program is started.

If the interrupt is requested during the low power modes 3 or 4, then additional two cycles are needed.

TI Worldwide Technical Support

Internet

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/general/distrib.htm

Product Information Centers

Americas
Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Internet	www.ti.com/sc/ampic

Europe, Middle East, and Africa
Phone	
Belgium (English)	+32 (0) 27 45 55 32
France	+33 (0) 1 30 70 11 64
Germany	+49 (0) 8161 80 33 11
Israel (English)	1800 949 0107
Italy	800 79 11 37
Netherlands (English)	+31 (0) 546 87 95 45
Spain	+34 902 35 40 28
Sweden (English)	+46 (0) 8587 555 22
United Kingdom	+44 (0) 1604 66 33 99
Fax	+44 (0) 1604 66 33 34
Email	epic@ti.com
Internet	www.ti.com/sc/epic

Japan
Phone	
International	+81-3-3344-5311
Domestic	0120-81-0026
Fax	
International	+81-3-3344-5317
Domestic	0120-81-0036
Intenet	
International	www.ti.com/sc/jpic
Domestic	www.tij.co.jp/pic

r1.2

Asia
Phone			
International	+886-2-23786800		
Domestic		Local Access Code	TI Number
Australia		1-800-881-011	-800-800-1450
China		10810	-800-800-1450
Hong Kong		800-96-1111	-800-800-1450
India		000-117	-800-800-1450
Indonesia		001-801-10	-800-800-1450
Korea		080-551-2804	-
Malaysia		1-800-800-011	-800-800-1450
New Zealand		000-911	-800-800-1450
Philippines		105-11	-800-800-1450
Singapore		800-0111-111	-800-800-1450
Taiwan		080-006800	-
Thailand		0019-991-1111	-800-800-1450
Fax	886-2-2378-6808		
Email	tiasia@ti.com		
Internet	www.ti.com/sc/apic		

© 2000 Texas Instruments Incorporated

TEXAS
INSTRUMENTS