AN429

MC68332 QSPI interface for the MCM2814 EEPROM

By Mark Maiolani
Motorola Semiconductors Ltd
East Kilbride
Glasgow

INTRODUCTION

This application note describes the software and hardware necessary to use the MCM2814 serial EEPROM with the MC68332 Queued Serial Peripheral Interface, or QSPI.

As well as giving specific details on accessing the MCM2814 EEPROM with the QSPI, this application note can also be used to provide general information on configuring and using the QSPI with SPI compatible devices.

The main areas covered are hardware configuration, the general operation of the QSPI and a description of software which allows programming and reading data from the MCM2814. Information on interfacing the software with high level 'C' language programs is also covered, with a short demonstration program included.

HARDWARE CONFIGURATION

Figure 1 shows a simple system with four MCM2814 EEPROMs connected directly to the MC68332 QSPI. The MCM2814 is a serially accessed 256 byte EEPROM, which can be used in either IIC or SPI protocol systems. In this application SPI mode is selected by pulling the MCM2814 MODE pin to the +5V supply level, Vdd. As the MCM2814 generates its programming voltage (Vpp) internally, only a single 5V supply is necessary.

In the simplest configuration, selection of the individual EEPROMs is accomplished by connecting the QSPI slave select lines, PCS0-3, directly to the MCM2814 SPI slave select lines, SPISS. With this configuration a maximum of four EEPROMs or other SPI devices can be individually selected.

If more than four devices are to be connected to the SPI bus, a decoder can be added to select a maximum of 15 devices.

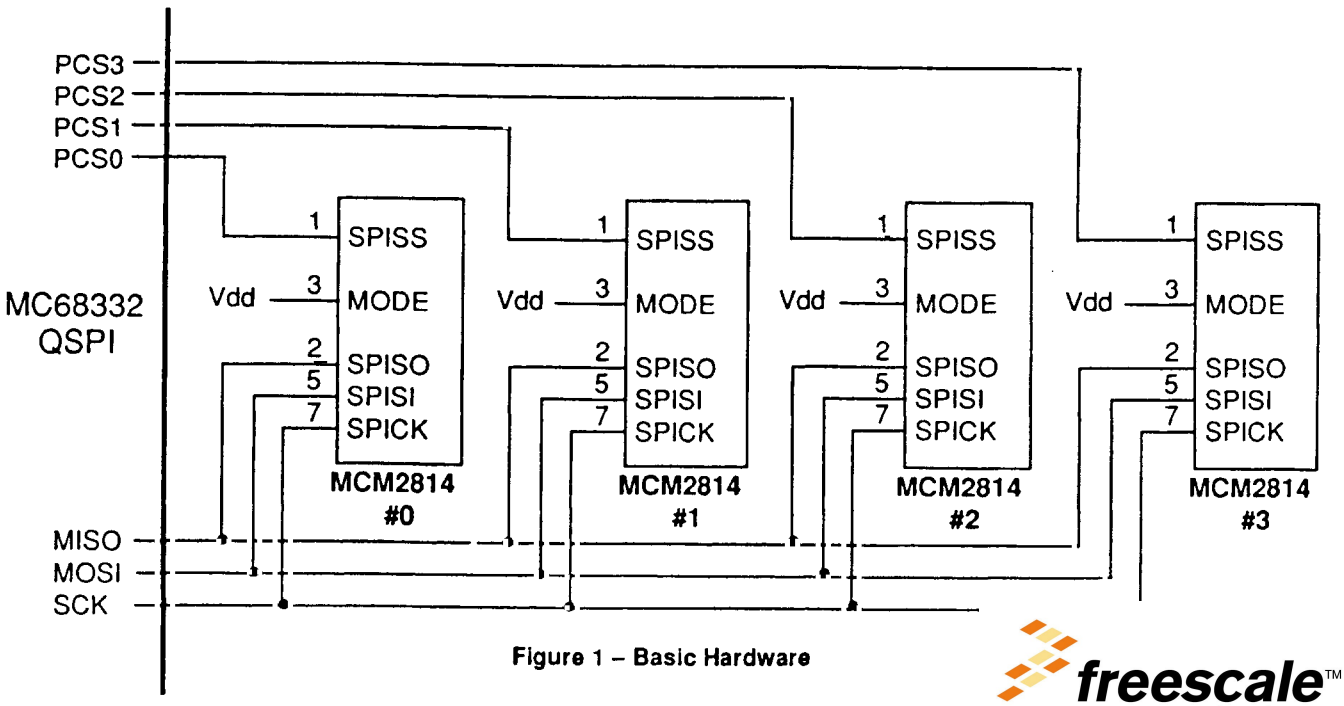


Figure 1 – Basic Hardware

SPI AND QSPI OVERVIEW

The SPI bus consists of two serial data lines, a clock line and one or more slave select lines. The two data lines, MOSI and MISO, are used by the master device on the bus to transmit and receive data respectively. The clock and slave select lines, designated SCK and PCS0-3 on the QSPI, are generated by the master, which in this case is the QSPI.

When the SPI master accesses a particular SPI device, the slave select line for that particular device is driven low before the required number of bits are transferred during SCK transitions.

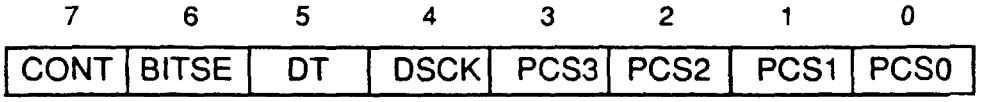
The SPI protocol defines that data transmission and reception always occur simultaneously, as while data is transmitted from the master on the MOSI line, data is being received along the MISO line. If the master only has to read data from a device, it may transmit uninitialised or 'don't care' values along MOSI, and conversely if the master only has to write data to a device it may ignore the data received along MISO.

The enhancements of the QSPI over the SPI consist mainly of the queued architecture of the QSPI. Rather than having to program each individual transfer before it is transmitted, as on the SPI, the QSPI can be configured to automatically carry out a number of transfers without intervention from the processor. Each transfer can be individually configured to access a specific device by using the PCS0-3 lines. Individual control of the number of bits per transfer, bus timing delays and the state of the PCS lines between transfers is also possible.

The transfer data and control information is contained in three queues:- the receive data queue REC.RAM, the transmit data queue TRAN.RAM and the command data queue COMD.RAM. The transmit and receive queues are 16 bits wide, although each SPI transfer can be from 8 to 16 bits long, defined by BITS of register SPCR0 and BITSE of the command entries. The command data queue is byte wide, with each byte configuring various aspects of one transfer, such as the number of bits in the transfer, bus delays and device selection. Figure 2 shows the bit usage of the command bytes.

All three queues are 16 entries deep, resulting in 32 bytes each for the transmit and receive data queues and 16 bytes for the command queue bits.

Command Queue Entry



BIT	STATE	ACTION
CONT	0	Slave Select Lines PCS 3-0 return to default states, as defined in QPDR, between transfers
	1	PCS 3-0 do not change between transfers
BITSE	0	Transfer length defaults to 8 bits
	1	Bits field of SPCR0 defines transfer length (8 - 16 bits)
DT	0	Default delay of 17 clocks after each transfer
	1	Delay specified by DTL field of SPCR1
DSCK	0	Default delay of 1/2 clock between device selection and transfer
	1	Delay specified by DSCKL field of SPCR1
PCS3-0	–	Defines the state of the slave select lines PCS3-0 during transfer

Figure 2 – Command entry format

NVMRWC SOFTWARE – OVERVIEW

The MC68332 assembly language software, NVMRWC, contains routines to both read and write EEPROM data via the QSPI, and is configured to be called as a function from a C language program.

The read routine is entered at <ee_read>, and is used to read up to 29 bytes of data starting from any MCM2814 location. Figures 3 and 4 show the main program flow for the read action, and an example read operation.

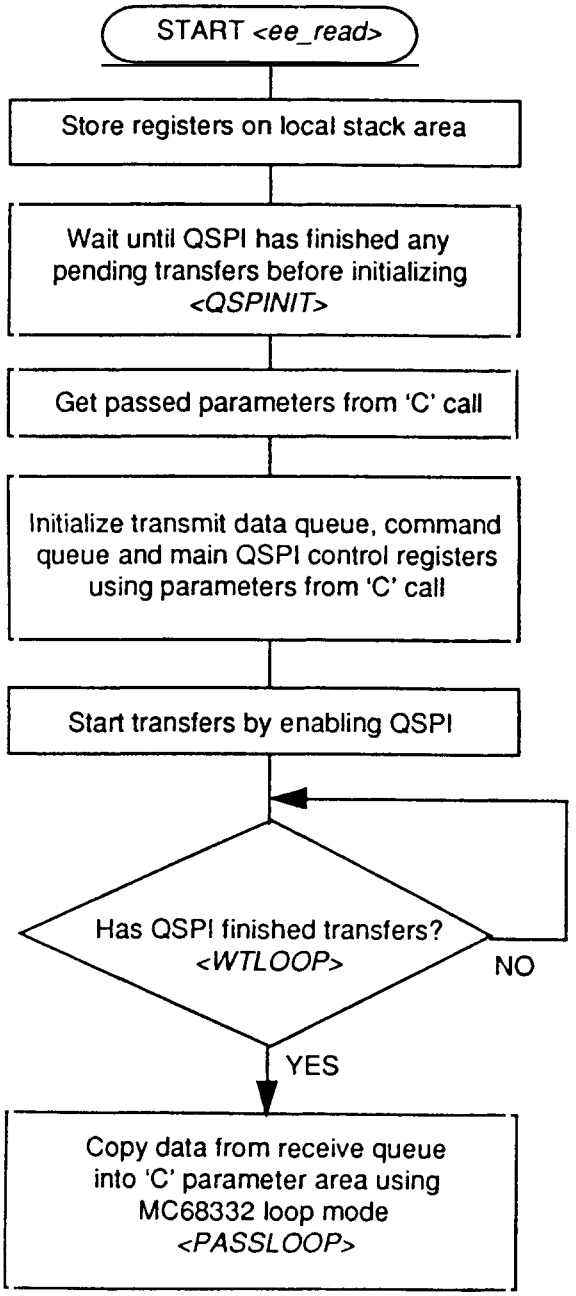


Figure 3 – EEPROM read flowchart

The write routine starts at <ee_write>, and is able to write up to 4 bytes of data starting at any location, as long as all of the bytes are within a 4 byte boundary. This limitation is due to the operation of the MCM2814, which is detailed in the data sheet for the device. Program flow for the write action and an example transfer are shown in figures 5 and 6.

Transmit Data Queue TRAN.RAM

15 – 8	7 – 0	
READ ($A7)	Address ($20)	Entry 0
XX	XX	Entry 1
XX	XX	Entry 2
XX	XX	Entry 3

Receive Data Queue REC.RAM

15 – 8	7 – 0	
XX	XX	Entry 0
Address ($20)	Byte $20 data	Entry 1
Byte $21 data	Byte $22 data	Entry 2
Byte $23 data	Byte $24 data	Entry 3

Command Queue CMD.RAM

CONT/BITSE/DT/DSCK				PCS3-0	
7	6	5	4	3210	
1	1	1	0	0010	Entry 0
1	1	1	0	0010	Entry 1
1	1	1	0	0010	Entry 2
0	1	1	0	0010	Entry 3

Example queue set-up for a read of 5 bytes of data starting from MCM2814 address $20, with the device select code, (PCS3-0), set to 0010. Note the use of 16 bit transfers for each two bytes of data. XX is un-initialised or unused data, underlined data is received from MCM2814.

Figure 4 – EEPROM read example

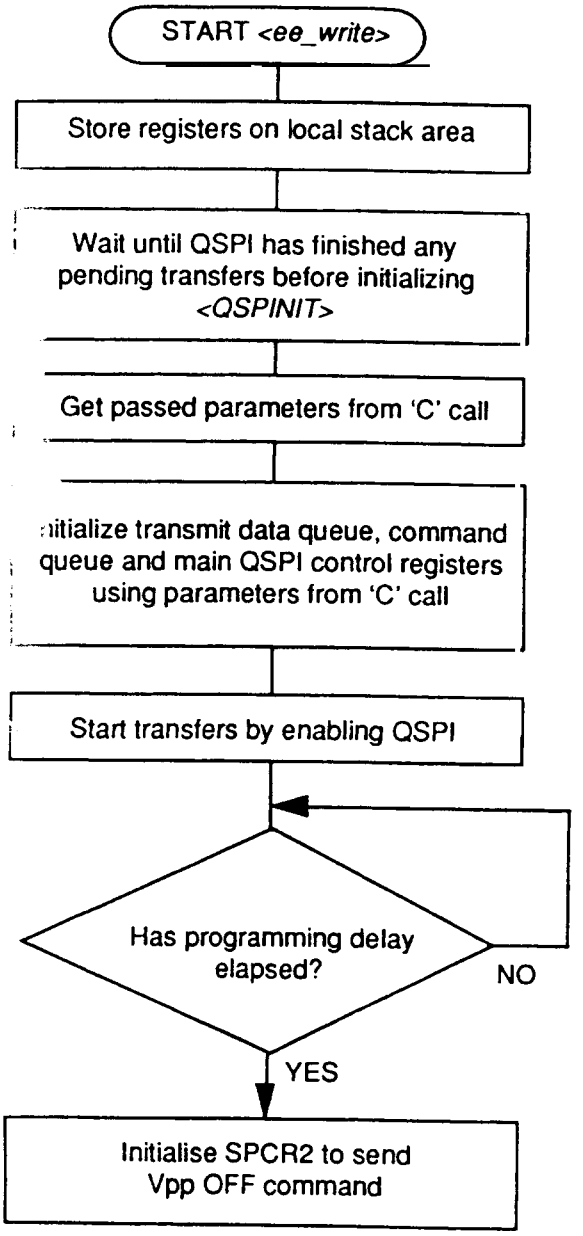


Figure 5 – EEPROM write flowchart

Transmit Data Queue TXD.RAM

15 8	7 0		
XX	Vpp ON	($A6)	Entry 0
XX	WRITE	($A2)	Entry 1
XX	Address	($10)	Entry 2
XX	Data	($AA)	Entry 3
XX	Data	($55)	Entry 4
XX	Vpp OFF	($A4)	Entry 5

Receive Data Queue REC.RAM

15 8	7 0	
XX	XX	Entry 0
XX	XX	Entry 1
XX	XX	Entry 2
XX	XX	Entry 3
XX	XX	Entry 4
XX	XX	Entry 5

Command Queue CMD.RAM

CONT/BITSE/DT/DSCK				PCS3-0	
7	6	5	4	3210	
0	0	1	0	0001	Entry 0
1	0	1	0	0001	Entry 1
1	0	1	0	0001	Entry 2
1	0	1	0	0001	Entry 3
0	0	1	0	0001	Entry 4
0	0	1	0	0001	Entry 5

Example queue set-up for a write of 3 bytes of data starting from MCM2814 address $10, with the device select code, (PCS3-0), set to 0010. XX is un-initialised or unused data, underlined data is received from MCM2814.

Figure 6 – EEPROM write example

WORKSPACE ALLOCATION – LINK, UNLK AND MOVEM

The routines <ee_read> and <ee_write> return with all processor registers restored to their original state. To accomplish this, the registers are written to the stack on entry, and recovered before returning. The instruction, LINK A6,#&-28, creates a 28 byte stack frame for this purpose by moving the stack pointer, A7, past the reserved area, and loading A6 to act as the frame pointer.

Storing the processor registers is accomplished in a single instruction by the MOVEM (move multiple) command. MOVEM.L D0/D1/D2/D3/A0/A1/A2,(A7) stores all of the listed registers as long words starting at the address pointed to by the stack pointer A7. As the LINK instruction sets A7 to point to the lowest address of the reserved 28 bytes, all of the registers are stored in this area. Figure 7 shows the stack organisation in detail.

The MOVEM instruction is used again at the end of the program, with the order of the operands reversed, to recover the register values, and the reserved stack area is deallocated by the UNLK (unlink) instruction. This recovers the original value of the specified local stack pointer, A6, and resets the main stack pointer, A7, to its previous value.

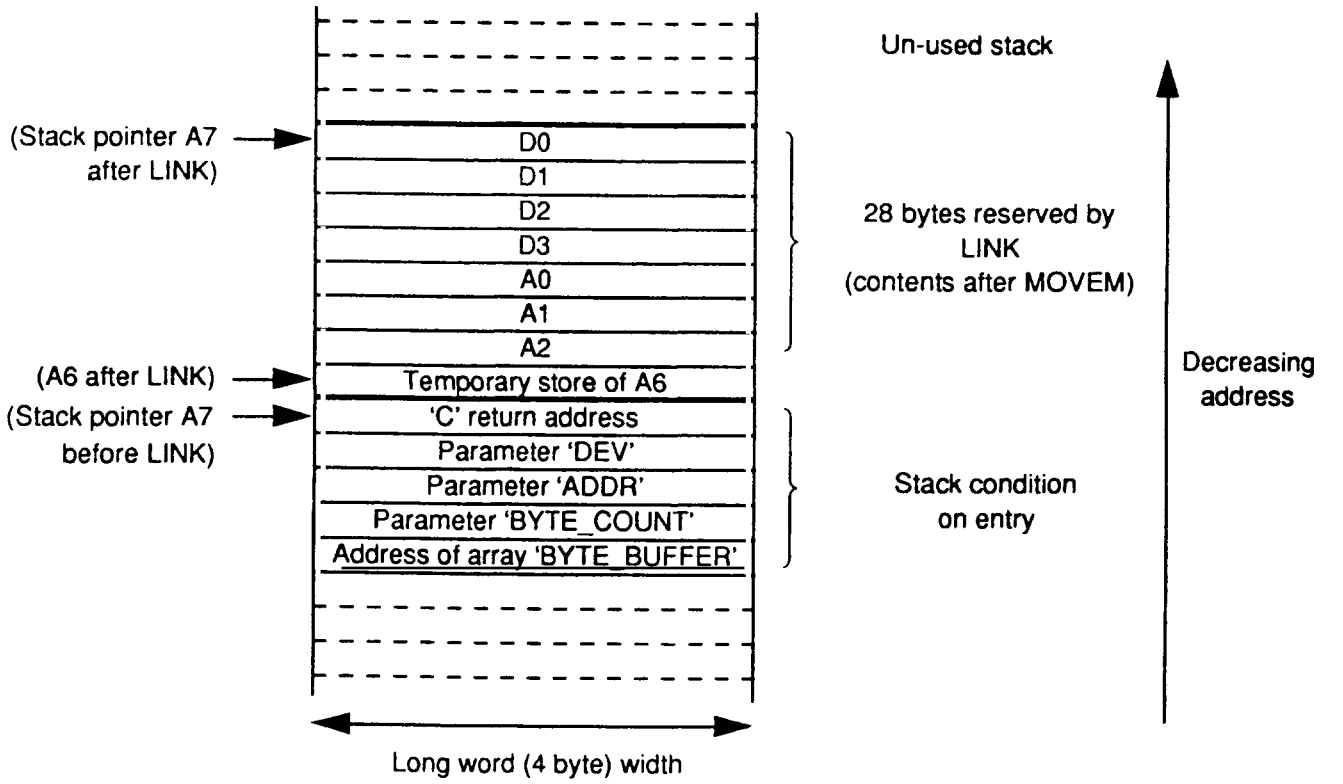


Figure 7 – Stack allocation

INITIALISING THE QSPI

Before the QSPI is used for an EEPROM read or write, it is initialised to the desired startup state. The subroutine <QSPINIT> allows the QSPI to complete any pending transfers before it is stopped and initialised. At this stage initialisation consists mainly of configuring the QSPI hardware by assigning the lines MISO,MOSI,SCK and PCS0-3 to the QSPI, with MOSI and PCS0-3 set as outputs which default high. The QSPI is also set to non wired-or outputs and SPI master with a clock rate of 100KHz at this point.

Note that if it is known the QSPI will be idle and in a suitable state when an EEPROM read or write is requested, or if it is acceptable to halt the QSPI with transfers pending, the initialisation subroutine could be simplified.

FUNCTION PARAMETER PASSING – MC68332 LOOP MODE

When either reading or writing EEPROM data, the software has to retrieve the parameters that the C program has placed on the stack. NVMRWC is configured to work with a C compiler that passes the variables in the stack locations shown in figure 7. If the C compiler passes the variables in a different manner, the program must be altered accordingly. The parameters are initially stored in the MC68332 internal registers as follows:-

A1 – Address in MC68332 memory of the C data array, <BYTE_BUFFER>

D0 – Number of bytes to transfer, <BYTE_COUNT>

D1 – MCM2814 starting byte address, <ADDR>

D2 – Device selection code for PCS0-3 pins, <DEV>

Note that A1 is the memory address of the main C array, rather than the actual data. This allows NVMRWC to read or modify the data in the array when necessary by using A1 as a pointer. An example of this is when the EEPROM data is passed back to the array after a read in the routine <PASSLOOP>. This program section uses a MOVE instruction to copy a byte of data from the QSPI receive data queue to the array, and a decrement if false, or DBF instruction to cause the program to loop around until a count register reaches 0. As the MOVE instruction occupies one word, the MC68332 automatically enters 'Loop Mode' when this program section is encountered. When in this mode, no instruction fetches are made to memory, thus greatly speeding execution.

TRANSMIT DATA QUEUE INITIALISATION

The transmit data queue has to be initialised with the correct sequence of MCM2814 commands and data before an EEPROM read or write can be carried out. Reading data from the EEPROM uses the simplest sequence, consisting of the following:-

MCM2814 READ command ($A7)

Starting byte address to be read

The remainder of the transmit queue does not need to be initialised, as after this sequence has been sent the MCM2814 transmits data, and no longer monitors incoming data. Because the MCM2814 remains selected between all of the individual byte transfers, the full 16 bit width of the data queues can be used to increase the maximum possible transfer size. The example in figure 4 shows that the read code and the byte address are sent as one 16 bit transfer, with the receive queue holding up to 29 bytes of received EEPROM data.

When writing data to the EEPROM, the following sequence is transmitted:-

Vpp ON command ($A6)

WRITE DATA command ($A2)

Starting byte address to be programmed

up to 4 bytes of data

Vpp OFF command ($A4)

Because the EEPROM has to be deselected at various points during the sequence, all transfers are 8 bit only. An example transfer is shown in figure 6.

COMMAND QUEUE INITIALISATION

The command queue is configured with the device selection state during and between transfers, timing information and bit size information.

For an EEPROM read or write sequence the device select code remains constant throughout, as all of the transfers are intended for a single device. This device select code, which is determined from the parameter <DEV>, is written into the PCS0-3 field of all of the COMD entries used.

An MCM2814 read sequence is treated as one command, and the device must remain selected for its full duration, even between the individual transfers. To accomplish this the CONT bits are set for all of the COMD entries except the one which controls the last transfer.

Because a MCM2814 write sequence consists of three command blocks (Vpp ON, WRITE and Vpp OFF) and the device has to be deselected between the blocks, the CONT bits have to be cleared for the last transfer of each block. This can be seen in the example, figure 6, where the CONT bits are clear in the COMD entries corresponding to the Vpp ON command, the last data byte and the Vpp OFF command.

As all EEPROM writes are byte size, BITSE is clear for all of the command queue entries. This forces the QSPI to use the default transfer size, which is 8 bits. For the EEPROM read operation all transfers are 16 bit, so BITSE is set for all of the COMD entries. This causes the QSPI to use the transfer size programmed into the BITS field of register SPCR0, which has previously been set to 16. If necessary, the BITS field can be used to select alternative transfer sizes from 8 to 16 bits.

To conform with the MCM2814 timing spec. a delay is generated after each transfer by setting bit DT in the command queue entries, causing the QSPI to use the 5µS delay specified in control register SPCR1.

No extra delay is needed between selection of the MCM2814 and data transfer, so the DSCK bits, which control this delay, are cleared in the command queue entries.

MAIN QSPI CONTROL REGISTERS

Before the QSPI transfers can be started, the main configuration register SPCR2 is configured. This register holds the first and last QSPI entry numbers that are to be sent which are dependent on the type of transfer and parameters, eg. the number of bytes to read/write. The WRAP control bits, which control the QSPI wrapping operation in which the QSPI constantly loops around a group of queue entries, are set to disable this function.

QSPI transfers are started by setting the QSPI enable bit, SPE, of SPCR1. The program then loops, testing for completion of transfers by polling the SPI finished flag, SPIF, of the status register SPSR.

DEMONSTRATION C PROGRAM – EECALL

A small demonstration C program, EECALL, shows the way that a C program can utilise the assembly language program NVMRWC as well as demonstrating its functions. Several functions can be invoked by a single keypress. These functions are:-

P – (Set Parameters) – This option allows the user to define the parameters Start Address, No. of Bytes and Device Code.

W – (Write Data) – This option makes one call to the assembly language routine <EE_WRITE>, using the parameters which have been set previously. The user is prompted to enter the data to be programmed, byte by byte. This data is entered as single ASCII characters, with no carriage return necessary.

R – (Read Data) – One call is made to the assembly language routine <EE_READ> using the previously set parameters. The returned data is printed in ASCII format, with each byte separated by a slash (/) character.

D – (EEPROM Dump) – The entire contents of one EEPROM (as selected by the previously set device code) are printed on screen in an ASCII table format. This option calls <EE_READ> 16 times, with each call reading 16 bytes of EEPROM.

T – (Text Entry) – When this option is selected, the user can enter a text message of undefined length to be programmed into EEPROM. To terminate the message a hash (#) must be entered. Programming is carried out character by character, by using a call to <EE_WRITE> to program each byte.

X – (Exit) – Exits the program EECALL by executing an RTS.

```
/*
 *      EECALL.C Basic front-end program to test and show use of
 *      assember calls EE_READ and EE_WRITE to read and write data
 *      on MCM2814 EEPROM connected to QSPI.
 */

        #include <terminal.h>

        typedef unsigned char byte;

        byte i,tbc,dev,addr,tadd,byte_count,byte_buffer[29];
        int iovar;

    char c;

        extern byte ee_write(byte d,byte a,byte b_count,byte *b_buffer);
        extern byte ee_read(byte d,byte a,byte b_count,byte *b_buffer);

void
pstring(s)
    char *s;
    {
    while (*s)
            _putchar(*s++);
    }

main()
    {
        byte error;

        dev=14;                 /* Initialise pass parameters */
        addr=0x10;
        byte_count=4;
        for (i=0;i<4;i++)
        byte_buffer[i]=i+40;

    open();                             /* initialize I/O device          */

    pstring("Enter an \"X\" to terminate the program\n\r");
        pstring("  ,,  ,, \"W\" to Write to EEPROM \n\r");
        pstring("  ,,  ,, \"R\" to Read the EEPROM\n\r");
        pstring("  ,,  ,, \"P\" to set Parameters\n\r");
        pstring("  ,,  ,, \"D\" to Dump entire EEPROM\n\r");
        pstring("  ,,  ,, \"T\" to enter Text message\n\r");


    while ((c = toupper(_getchar())) != 'X')
                {
                if (c == 'W')
                    {
                    pstring("Write Data\n\r");
                    for (i=0;i<byte_count;i++)
                        {
                        pstring("\n\rEnter char - ");
                        c=_getchar();
                        _putchar(c);
                        byte_buffer[i]=c;
                        }
                    pstring("\n\rWriting Data to EEPROM...\n\r");
                    error = ee_write(dev,addr,byte_count,byte_buffer);
                    error = error+1;
                    }
```

```
        if (c == 'T')
            {
            printf("Program text from address %d.\n\r",addr);
            pstring("Enter Text - # to terminate\n\r");
            tadd = addr;
            tbc = 1;
            while ((c =_getchar()) != '#')
                {
                _putchar(c);
                byte_buffer[0]=c;
                error = ee_write(dev,tadd,tbc,byte_buffer);
                ++tadd;
                }
            }

        if (c == 'R')
            {
            pstring("Read Data\n\r");
            ee_read(dev,addr,byte_count,byte_buffer);
            for (i=0;i<byte_count;i++)
                {
                putchar(byte_buffer[i]);
                putchar('/');
                }
            pstring("\n\r");
            }

        if (c == 'D')
            {
            pstring("Block EEPROM dump\n\r");
            tbc = 16;
            tadd = 0;
            do
                {
                printf("\n %4d - ",tadd);
                ee_read(dev,tadd,tbc,byte_buffer);
                for (i=0;i<16;i++)
                   {
                   if (byte_buffer[i] < 32 )
                      byte_buffer[i] = '.';
                   putchar(byte_buffer[i]);
                   }
                tadd = tadd + 16;
                putchar(' ');
                } while (tadd>0);
            pstring("\n\r");
            }

        if (c == 'P')
            {
            pstring("Parameters");
            pstring("\n\rEnter no. of bytes 0-29 :");
            do
               scanf("%d",&iovar);
              while ( iovar > 29 );
            byte_count = (byte)iovar;
            pstring("Enter start address 0-255 :");
            do
               scanf("%d",&iovar);
               while (iovar > 255);
            addr = (byte)iovar;
            printf("Bytes = %d Address = %d.\n",byte_count,addr);
            pstring("\n\r");
            }
        pstring("\n\rX:eXit W:Write R:Read P:Parameters D:Dump T:Text\n\r");
        pstring ("?");
        }

pstring("EXIT PROGRAM\n\r");
}
```

(This page intentionally left blank)

```
*     MCM2814 EEPROM Read and Write Subroutines for the MC68332 QSPI
*           Configured as a 'C' language external call
*
* Copyright Motorola 1990

* Call format: EE_READ (DEV,ADDR,BYTE_COUNT,*BYTE_BUFFER)

          LIB 68332.REG            Include MC68332 register equates

          section .data

*************************************************
*         EEPROM read
*************************************************
*     *** CREATE LOCAL STACK FRAME AND STORE REGISTERS
ee_read   LINK     A6,#&-28         Allocate local stack area of 28 bytes
*                                   and use A6 as local stack pointer
          MOVEM.L  D0/D1/D2/D3/A0/A1/A2,(A7)
*                                   Store registers in local stack frame

*     *** INITIALISE QSPI
          BSR QSPINIT              Disable QSPI and initialise I/O


*     *** FETCH PARAMETERS FROM STACK USING A0 AS POINTER
          LEA      (&52,A7),A0     Point to element above first parameter
          MOVE.L   -(A0),A1        Put BYTE_BUFFER address in A1
          MOVE.L   -(A0),D0        Put BYTE_COUNT in D0
          MOVE.L   -(A0),D1        Put ADDR in D1
          MOVE.L   -(A0),D2        Put DEV in D2

*     *** LOAD TXD QUEUE WITH MC2184 COMMANDS
*         0: Read byte command
*         1: MCM2814 byte address
          MOVE.B   #$A7,TXD        store READ command
          MOVE.B   D1,TXD+1        store byte address into TXD queue
*                                  (D1 can be used now)
*     *** ENSURE BYTE_COUNT IS IN RANGE 1 TO 29
          ANDI.W   #$00FF,D0       Clear MSB (word will be used in DBcc)
          CMP.B    #$1E,D0         Should be 0 < BYTE_COUNT < &30
          BCS      B_COK1          BYTE_COUNT < &30?
          MOVE.B   #$1D,D0         No, so force to &29
B_COK1    TST      D0              BYTE_COUNT <> 0?
          BNE      B_COK2
          MOVE.B   #$01,D0         No, so force to 1
*         BYTE_COUNT should be O.K. now
B_COK2    MOVE.W   D0,D1           Working copy of BYTE_COUNT in D1


*     *** SET UP COMD QUEUE

*         Use DEV code to calculate COMD queue entries
          ORI.B    #$E0,D2         Calculate entry with CONT bit set
*                                  16 bit transfer
          MOVEA.L  #COMD,A0        Use A0 as COMD queue pointer

*         No. of 32 bit transfers = ((BYTE_COUNT+2)/2) + 1
*         Calculate value in D2 for DBcc loop to set up COMD queue
          ADD.B    #$02,D1         BYTE_COUNT +2
          ASR.B    #1,D1           (BYTE_COUNT+2)/2
          SUB.B    #$01,D1         (BYTE_COUNT+2)/2-1
*         Set up all COMD entries except last with CONT bit set
COMDLOOP  MOVE.B   D2,(A0)+
          DBF      D1,COMDLOOP
*         Set up last COMD entry with CONT clear (deselect EEPROM at end)
          ANDI.B   #$6F,D2         Calculate entry with CONT bit clear
          MOVE.B   D2,(A0)         Install in COMD queue
```

```
*       *** CALCULATE LAST QSPI ENTRY NO. AND INCORPORATE IN SPCR2
            MOVE.B    D0,D1             Make working copy of BYTE_COUNT
            ADD.B     #$02,D1           BYTE_COUNT +2
            ASR.B     #1,D1             (BYTE_COUNT+2)/2
            MOVE.B    D1,SPCR2          Put into SPCR2 MSB
            MOVE.B    #$00,SPCR2+1      and 00 as start entry in LSB

*       *** READ DATA FROM MCM2814
            MOVE.W    #$8003,SPCR1      Enable QSPI, DTL delay of approx 5uS

*       *** WAIT FOR END OF TRANSMISSION
WTLOOP      TST.B     SPSR              Test SPIF bit
            BPL       WTLOOP            and wait till set

*       *** PASS DATA BACK TO C PROGRAM ARRAY
            SUB.B     #$01,D0           Use BYTE_COUNT-1 as loop counter
            MOVEA.L   #REC+3,A0         Use A0 as REC queue pointer

*           Use MC68332 LOOP MODE to fill array
PASSLOOP    MOVE.B    (A0)+,(A1)+       Copy one data entry from REC queue
            DBF       D0,PASSLOOP       (loop until D0<0 )

*       *** RESTORE REGISTERS AND DE-ALLOCATE STACK
            MOVEM.L   (A7),D0/D1/D2/D3/A0/A1/A2
*                                       Retrieve registers from local stack frame
            UNLK      A6                And de-allocate local stack
            RTS

************************************************
*           EEPROM write
************************************************

*       *** CREATE LOCAL STACK FRAME AND STORE REGISTERS
ee_write    LINK      A6,#&-28          Allocate local stack area of 16 bytes for
*                                       reg. store, using A6 as stack pointer
            MOVEM.L   D0/D1/D2/D3/A0/A1/A2,(A7)
*                                       Store registers in local stack frame


*       *** INITIALISE QSPI
            BSR QSPINIT                 Disable and initialise QSPI

*       *** FETCH PARAMETERS FROM STACK USING A0 AS POINTER
            LEA       (52,A7),A0        Point to element above first parameter
            MOVE.L    -(A0),A1          Put BYTE_BUFFER address in A1
            MOVE.L    -(A0),D0          Put BYTE_COUNT in D0
            MOVE.L    -(A0),D1          Put ADDR in D1
            MOVE.L    -(A0),D2          Put DEV in D2
*           Stack pointer unchanged



*       *** LOAD TXD QUEUE WITH MCM2814 COMMANDS
*           0: Vpp on command
*           1: Write data command
*           2: reserved for byte address
*           3:    ,,      ,, data
*           4: Vpp off command
            MOVE.B    #$A6,TXD+1        Vpp ON command
            MOVE.B    #$A2,TXD+3        WRITE command
            MOVE.B    D1,TXD+5          Use ADDR as byte address
*                                       (D1 can be used now)

*       *** Adjust Byte_Count for use in DBcc loop
            SUB.B     #$01,D0           Need BYTE_COUNT loop counter in range 0-3
            ANDI.W    #$03,D0           for use in DBcc loop, not 1-4
            MOVE.W    D0,D1             Working copy in D1
```

```
*       *** PROGRAM COMD AND TXD QUEUES
*           Program start of TXD and COMD queues
            MOVEA.L  #COMD,A0          Use A0 as COMD queue pointer
            MOVEA.L  #TXD+7,A2         Use A2 as TXD queue pointer
            ORI.B    #$20,D2           Calculate COMD entry with CONT bit clear
            MOVE.B   D2,(A0)+          Use as entry for Vpp ON (CONT clear)
            ORI.B    #$A0,D2           Calculate COMD entry with CONT set
            MOVE.B   D2,(A0)+          Use as entry for WRITE DATA and
            MOVE.B   D2,(A0)+          ADDR, (CONT set)

*           PROGRAM TXD AND CMD QUEUES
DATALOOP    MOVE.B   (A1)+,(A2)+       Put a byte of write data in TXD queue
            ADDA.L   #$01,A2           TXD buffer is WORD wide, so increment
            MOVE.B   D2,(A0)+          Put an entry into COMD queue (CONT set)
            DBF      D1,DATALOOP       Loop until finished

*           Finish off COMD and TXD queue setup
            MOVE.B   #$A4,(A2)         Vpp OFF command in TXD queue
            ANDI.B   #$2F,D2           Calculate COMD entry with CONT clear
            MOVE.B   D2,(-1,A0)        Change last data COMD entry to CONT clear
            MOVE.B   D2,(A0)           Last COMD entry (Vpp OFF), CONT clear


*       *** CALCULATE WHAT QUEUE ENTRIES TO SEND FOR PROGRAMMING
            ADD.B    #$03,D0           Last data entry is no. 2+BYTE_BUFFER
            MOVE.B   D0,SPCR2          Put into SPCR2 MSB
            MOVE.B   #$00,SPCR2+1      and 00 as start entry in LSB


*       *** START TRANSMISSION AND PROGRAMMING DELAY
            MOVE.W   #$8003,SPCR1      ** Start programming **
            MOVE.L   #$00008000,D1     Programming delay (approx 20ms)
LOOP        DBF      D1,LOOP

*       *** CALCULATE WHAT QUEUE ENTRIES TO SEND FOR Vpp OFF
            ADD.B    #$01,D0           Vpp OFF command is immediately after
*                                      programming sequence
            MOVE.B   D0,SPCR2          Put into SPCR2 MSB and LSB
            MOVE.B   D0,SPCR2+1        so that only this command is sent

*       *** RESTORE REGISTERS AND DE-ALLOCATE STACK
            MOVEM.L  (A7),D0/D1/D2/D3/A0/A1/A2
*                                      Retrieve registers from local stack frame
            UNLK     A6                And de-allocate local stack
            RTS
```

```
*****************************************************
*         QSPINIT
*         Orderly stop and initialise QSPI hardware
*****************************************************
QSPINIT   ORI.B    #$80,SPCR0       Ensure QSPI is master
          ANDI.B   #$BF,SPCR2       Clear WREN (stop wrapping)
          ANDI.B   #$7F,SPSR        Clear SPIF to enable sensing of when
*                                   transmission has finished

*         Is QSPI active
          TST.B    SPCR1            Test SPE bit to see if QSPI enabled
          BPL      MISS             Goto MISS if disabled

*         Wait till QSPI reached end of current queue
          ANDI.B   #$06,SPCR3       Ensure system not HALTed
NOSPIF    TST.B    SPSR             Wait until SPIFinished
          BPL      NOSPIF

*         Disable QSPI
          ANDI.B  #$7F,SPCR1        Clear SPE bit

*         Initialise QSPI for accessing MCM2814
MISS      MOVE.W   #$8054,SPCR0     Set MASTER, no WIRED OR, 16 bits, 100KHz
          MOVE.W   #$7B7E,QPAR      Configure MOSI,MISO+PCS0-3 as QSPI lines
          MOVE.W   #$00FA,QPDR      MOSI+PCS0 to default high
          ANDI.B   #$7F,SPSR        Clear SPIF to enable sensing of when
*                                   next transmission has finished
          RTS

*         Allow other programs (eg C program) to access routine labels
*         ee_write and ee_read

          export ee_write
          export ee_read
```

(This page intentionally left blank)

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com