TEXAS INSTRUMENTS

TMS34010 Math/Graphics Function Library

User's Guide

TMS34010 Math/Graphics Function Library User's Guide

1987

1987

Graphics Products

TMS34010 Math/Graphics Function Library User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications 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. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that 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.

Contents

Illustrations

Tables

Section 1

Introduction

The TMS34010 Graphics System Processor (**GSP**) is an advanced 32-bit microprocessor optimized for graphics systems. The GSP is a member of the TMS340 family of computer graphics products from Texas Instruments. The TMS34010 is well supported by a full set of hardware and software development tools, including a C compiler, a full-speed emulator, a software simulator, an IBM/TI-PC development board, and a *math/graphics function library*.

The TMS34010 math/graphics function library is a collection of mathematics and graphics functions that can be called from C programs. The math functions include standard C double-precision floating-point routines for performing algebraic, trigonometric, and transcendental operations. The graphics functions include routines for viewport management, bit-mapped text, graphics output, color-palette control, three-dimensional transformations, and graphics initialization.

The library can be installed on the following systems:

- **PCs:**
  - IBM-PC with PC-DOS
  - TI-PC with MS-DOS
- **VAX:**
  - VMS
  - DEC Ultrix
  - Unix System V

> **Note:**
>
> In order to use the math/graphics function library, you must have the TMS34010 assembly language tools package and the TMS34010 C compiler.

The TMS34010 can execute all the functions in the library. Most of the functions are system-independent. Some of the graphics functions, however, must manage system-dependent features; such functions are compatible with the TMS34010 software development board. For more information about system-dependent features, see Section 4.17 on page 4-36.

Topics covered in this introductory section include:

1.1 Development Tools Overview

Figure 1-1 shows the TMS34010 assembly language development flow. The center section of the illustration highlights the most common path; the other portions are optional.

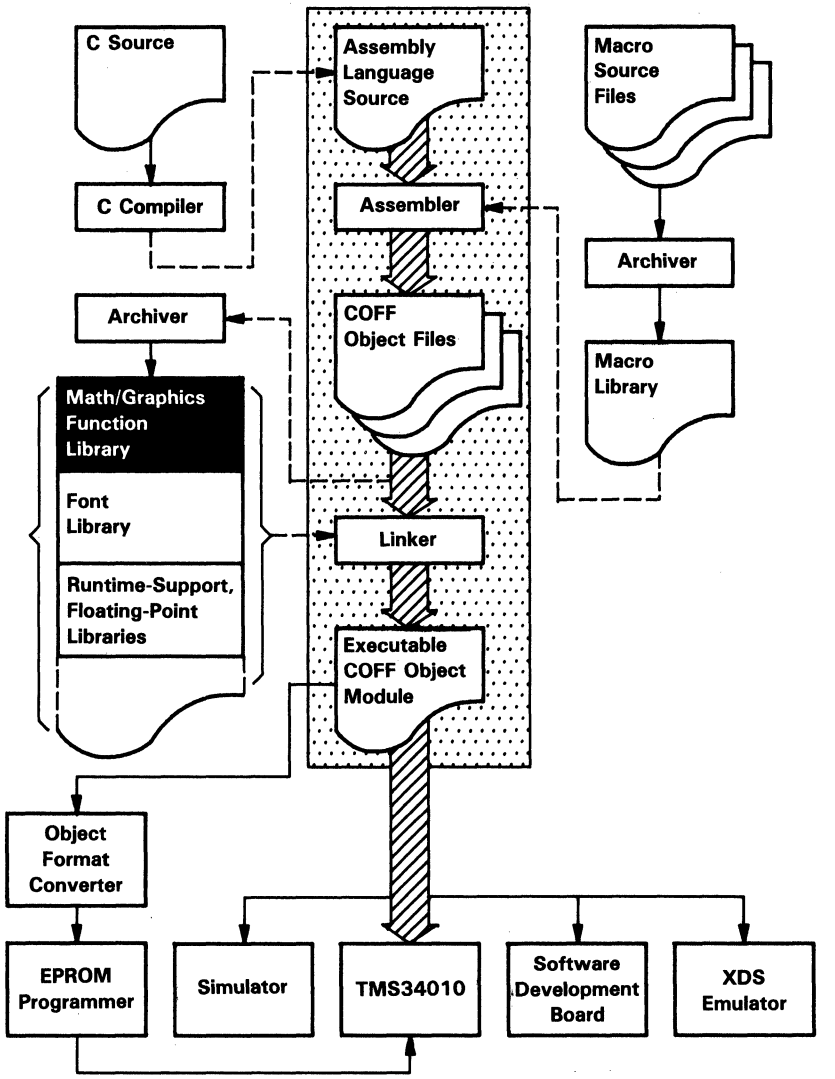


Figure 1-1. TMS34010 Assembly Language Development Flow

- The **C compiler** translates C source code into TMS34010 assembly language source code.

- The **assembler** translates assembly language source files into machine language object files.

- The **archiver** allows you to collect a group of files into a single archive library. It also allows you to modify the library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is to build a library of object modules. Several object libraries are available as TMS34010 products:

  - The **math/graphics function library** is discussed in this manual.

  - The **font library** is a separate product that contains a variety of proportionally spaced and monospaced fonts.

  - The **runtime-support** and **floating-point libraries** are shipped with the C compiler.

  - The **CCITT group3/group4 compression/decompression library**.

  - The **8514 IBM graphics display library**.

  These libraries contain functions that can be called from a C program. You can also create your own object libraries. To use an object library, you must specify it as linker input; the linker will include the members in the library that resolve external references during the link.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input.

- The main purpose of this development process is to produce a module that can be executed in a system that contains a **TMS34010**. You can use one of several debugging tools to refine and correct your code; available products include:

  - A **simulator**,

  - A **software development board** (SDB), and

  - An **emulator** (XDS).

  An **object format converter** is also available; it converts a COFF object file into a Tektronix-hex, Intel-hex, or TI-tagged object format file that can be downloaded to an EPROM programmer.

1.2 Manual Organization

Section 1 Introduction

Provides an overview of the function library, describes the TMS34010 development support tools, lists related documentation, and explains the style and symbol conventions used throughout this document.

Section 2 Installation and Operation

Contains instructions for installing the function library on PC and VAX systems, describes the files that are shipped with the product, provides examples of compiling, assembling, and linking C code that calls the functions, and provides examples of using the archiver with the library.

Section 3 Math Routines

Describes the functional categories of math routines, including double-precision floating-point functions, single-precision functions, type-conversion functions, and math routine error reporting.

Section 4 Graphics and Text Functions

Summarizes the graphics and text functions by category and provides general information about each category. Contains supplemental information about fonts and font management.

Section 5 Alphabetical Reference of Functions

Provides a page-by-page alphabetical reference of the functions that are in the library.

1.3 Related Documentation

The following TMS34010 documents are available from Texas Instruments:

- The ***TMS34010 C Compiler User's Guide*** (literature number SPVU005) tells you how to use the TMS34010 C compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS34010 assembly language source code. We suggest that you use *The C Programming Language* (Kernighan and Ritchie) as a companion to the *TMS34010 C Compiler User's Guide*.

- The ***TMS34010 Assembly Language Tools User's Guide*** (literature number SPVU004) describes common object file format, assembler directives, macro language, and assembler, linker, archiver, simulator, and object format converter operation.

- The ***TMS34010 Font Library User's Guide*** (literature number SPVU007) describes a set of fonts that are available for use in a TMS34010-based graphics system.

- The ***TMS34010 Data Sheet*** (literature number SPVS002) contains the recommended operating conditions, electrical specifications, and timing characteristics of the TMS34010.

- The ***TMS34010 User's Guide*** (literature number SPVU001) discusses hardware aspects of the TMS34010, such as pin functions, architecture, stack operation, and interfaces, and contains the TMS34010 instruction set.

- The ***TMS34010 Software Development Board User's Guide*** (literature number SPVU002) describes using the TMS34010 software development board (a high-performance, PC-based graphics card) for testing and developing TMS34010-based graphics systems.

- The ***TMS34070 User's Guide*** (literature number SPPU016) describes the TMS34070 color palette chip.

- The ***TMS34010 Software Development Board Schematics*** (literature number SPVU003) is a companion to the *TMS34010 Software Development Board User's Guide*.

You may also find the following documents useful:

Cody, William J. Jr. and William Waite. *A Software Manual for the Elementary Functions*. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

Newman, W.M., and R.F. Sproull. *Principles of Interactive Computer Graphics*. 2nd ed. New York: McGraw-Hill, 1979.

1.4 Style and Symbol Conventions

- In this document, program listings or examples, interactive displays, filenames, and symbol names are shown in a `special font`. Examples use **`a bold version of the special font`** for emphasis. Here is a sample program listing:

```
rvalu   = 0.0;
rvalv   = 1.0;
radians = atan(rvalu, rvalv)
```

- ***CR*** means *press the carriage return key.*

Section 2

Installation and Operation

This section contains step-by-step instructions for installing and operating the math/graphics function library. The library can be installed on five operating systems:

IBM and TI PCs

- PC-DOS[1] (IBM PC)
- MS-DOS[2] (TI PC)

Digital Equipment Corporation VAX-11[3]

- VMS operating system
- DEC Ultrix operating system
- Unix System V operating system

To use the math/graphics function library, you need the following software tools and object libraries:

- TMS34010 C compiler
- TMS34010 assembly language tools package (assembler, linker, and archiver)
- Runtime-support and floating-point libraries (`rts.lib` and `flib.lib`)[4]

The *TMS34010 C Compiler User's Guide* and *TMS34010 Assembly Language Tools User's Guide* describes these tools.

You will find instructions for installing and using the library in the following sections:

---

1 PC-DOS is a trademark of International Business Machines.

2 MS is a trademark of Microsoft Corporation.

3 VAX-11 and VMS are trademarks of Digital Equipment Corporation.

4 These libraries are shipped with the TMS34010 C compiler.

2.1 Installation for IBM/TI PCs with PC/MS-DOS

The math/graphics function library is shipped on several double-sided, dual-density diskettes.

The installation instructions use these symbols for drive names:

A: Floppy disk drive.
C: Winchester or hard disk (**E:** on TI PCs).

1) Make backups of the product diskettes.

   a) First format several blank diskettes. Insert a blank (destination) diskette in drive A. Enter:

      ```
      FORMAT A: CR
      ```

   b) Now copy the disks; enter:

      ```
      DISKCOPY A:  A: CR
      ```

      Follow the system prompts, removing and inserting the product and blank diskettes as directed.

2) Create a directory to contain the TMS34010 software. Enter:

   ```
   MD C:\GSPLIB CR
   ```

3) Copy the files onto the hard disk. Insert a product diskette in drive A. Enter:

   ```
   COPY A:\*.*  C:\GSPLIB\*.* CR
   ```

2.2 Installation for VAX/VMS

The tape was created with the VMS BACKUP utility at 1600 BPI.

1) Mount the tape on your tape drive.

2) Execute the following commands. Note that you must create a destination directory for the tools; in this example, `DEST:directory` represents that directory. Replace `TAPE:` with the name of the tape drive you are using.

```
$ allocate             TAPE:
$ mount/for/den=1600   TAPE:
$ backup               TAPE:GSP.bck DEST:directory
$ dismount             TAPE:
$ dealloc              TAPE:
```

2.3 Installation for VAX/ULTRIX and VAX/System V

This tape was made at 1600 BPI using the TAR utility. Follow these instructions to install the software:

1) Mount the tape on your tape drive.

2) Make sure that the directory in which you will store the tools is the current directory.

3) Enter the TAR command for your system; for example,

   ```
   TAR x
   ```

   This copies the entire tape into the directory.

2.4 Using the Library

Several types of files are shipped with the math/graphics library product:

- Object libraries (***.lib*** extension). These libraries contain the compiled, assembled versions of the functions; libraries are in archive format. The object libraries that are shipped with this product include:

```
grafix.lib
vuport.lib
text.lib
math.lib
```

- Source libraries (***.src*** extension). These libraries contain the C source or assembly language source versions of the functions; libraries are in archive format. The source libraries that are shipped with this product include:

```
grafix1.src
grafix2.src
grafix3.src
vuport.src
text.src
math.src
```

- Batch and command files (***.bat*** and ***.cmd*** extensions):
  - `gspc.bat` invokes the compiler (preprocessor, parser, and code generator) and the assembler.
  - `lc.bat` invokes the linker and calls a linker command file named `lc.cmd`.

Notes:

1. `lc.bat` expects `lc.cmd` to be in a directory named `gsplib`.
2. `lc.cmd` expects the functions in the math/graphics library to be in a directory named `gsplib`; it expects the `rts.lib` and `flib.lib` libraries to be in a directory named `gsptools`.

Refer to the product release notes for a complete list of the files that are shipped with the product.

2.4.1 Creating an Object Module that Contains Called Functions

Most of the the math/graphics functions can be called from a C program. (The syntaxes of various function calls are defined in later sections). To use the functions, you must compile, assemble, and link the C source program that calls the functions. When you link the resulting object file, you must also link in the appropriate function library. Whenever you specify an object library as linker input, the linker automatically includes the library members that contain called functions or resolve symbol references.

Example 2-1 and Example 2-2 show methods for compiling, assembling, and linking a C source program that calls functions in the function library. *Note that the examples are for PC/MS-DOS systems.* In these examples, assume that a file named `test.c` calls the functions open—vuport, select—vuport, and close—vuport; these functions are in the library `\gsplib\vuport.libs`.

Example 2-1. Method 1 - Invoke Each Tool Individually

1) Compile `test.c`:

```
a)  gspcpp test
    C Pre-Processor,      Version x.x, 87.100
    (c) Copyright 1985, 1987 Texas Instruments Inc.

b)  gspcc test
    GSP C Compiler,       Version x.x, 87.100
    (c) Copyright 1985, 1987 Texas Instruments Inc.
       "test.c"  ==> main

c)  gspcg test
    GSP C Codegen,        Version x.x, 87.100
    (c) Copyright 1985, 1987 Texas Instruments Inc.
       "test.c"  ==> main
```

This creates an output file named `test.asm`.

2) Assemble `test.asm`:

```
gspa test
GSP COFF Assembler,  Version x.x, 87.100
(c) Copyright 1985, 1987 Texas Instruments Inc.
 PASS 1
 PASS 2

No Errors, No Warnings
```

This creates an output file named `test.obj`.

3) Set the environment variable so the linker can find the object libraries:

```
set C_DIR = \libs; \gsplib
```

4) Link `test.obj` with the appropriate object libraries:

```
gsplnk test vuport.libs rts.lib flib.lib -o test.out
GSP COFF Linker,      Version x.x, 87.260
(C) Copyright 1985, 1987, Texas Instruments Inc.
```

This creates an object module called `test.out`.

Example 2-2. Method 2 - Use the Batch Files to Invoke the Tools

1) Use the `gspc.bat` file to compile and assemble `test.c`:

```
gspc test
---[test]---
C Pre-Processor,      Version x.x, 87.100
(c) Copyright 1985, 1987 Texas Instruments Inc.
GSP C Compiler,       Version x.x, 87.100
(c) Copyright 1985, 1987 Texas Instruments Inc.
   "test.c"  ==> main
GSP C Codegen,        Version x.x, 87.100
(c) Copyright 1985, 1987 Texas Instruments Inc.
   "test.c"  ==> main
GSP COFF Assembler,  Version x.x, 87.100
(c) Copyright 1985, 1987 Texas Instruments Inc.
 PASS 1
 PASS 2

No Errors, No Warnings
       Successful Compile of Module test
```

2) Use the `lc.bat` file to link `test.obj`. `lc.bat` calls a linker command file named `lc.cmd` that automatically links in the object libraries.

```
lc test
gsplnk   -c   -m   test.map   -o   test.out   test.obj
\gsplib\lc.cmd
GSP COFF Linker,      Version x.x, 87.260
(C) Copyright 1985, 1987, Texas Instruments Inc.
```

This creates an object module called `test.out`.

2.4.2 Archive Files

An archive file (or library) is a partitioned file that contains complete files as members. The math/graphics function library contains two types of libraries:

- Source libraries, which contain the source versions of the functions as members.
- Object libraries, which contain the compiled, assembled versions of the functions as members.

The TMS34010 archiver is a software utility that allows you to manipulate the members of a library by adding members, extracting members, deleting members, and replacing members. The *TMS34010 Assembly Language Tools User's Guide* contains complete instructions for using the archiver. Example 2-3, Example 2-4, and Example 2-5 show how you might use the archiver to manipulate the math/graphics libraries.

Example 2-3. Listing the Contents of a Library

Since the math/graphics library contains several object archive files, you may want to list the contents of the individual libraries so that you can determine which functions reside in which libraries. To do this, invoke the archiver with the -t option:

```
gspar -t \gsplib\vuport.lib
GSP Archiver          Version x.x, 87.261
(c) Copyright 1985, 1987, Texas Instruments Inc.

        FILE NAME     SIZE   DATE
 ---------------     -----  ------------------------
     set_visr.asm     4961  Mon Mar  9 11:03:32 1987
       close_vu.c     2532  Tue Mar 31 08:14:56 1987
       copy_vup.c     6891  Tue Mar 31 08:29:40 1987
       initvupo.c     2716  Tue Mar  3 15:05:06 1987
           .             .            .
           .             .            .
           .             .            .
```

Example 2-4. Extracting and Replacing a Member

Assume that you want to modify a function called close_vuport. The source code for this function is in the library `\gsplib\vuport.src`; the object code for this function is in the library `\gsplib\vuport.lib`. Follow these steps to create a new object file and replace it in the library:

1) Extract the function from the source library (the -x option tells the archiver to extract the specified member):

```
gspar -xv \gsplib\vuport.lib close_vu.c
GSP Archiver          Version x.x, 87.261
(c) Copyright 1985, 1987, Texas Instruments Inc.
  ==>  extract 'close_vu.c'
```

2) Edit the source file.

3) Compile and assemble the modified function:

```
gspc close_vuport
```

4) Replace the version of `close_vu.obj` that is in the library with the new version (the -r option tells the archiver to replace the specified member):

```
gspar -rv \gsplib\vuport.lib close_vu.obj
GSP Archiver          Version x.x, 87.261
(c) Copyright 1985, 1987, Texas Instruments Inc.
  ==>  replace 'close_vu.c'
  ==>  building archive '\gsplib\vuport.lib'
```

Note that the v command tells the archiver to print supplementary status information.

Example 2-5. Extracting all the Members from a Library

You can extract *all* the members of a library by invoking the archiver with the -x option without specifying any member names:

```
gspar -x \gsplib\vuport.src
GSP Archiver          Version x.x, 87.261
(c) Copyright 1985, 1987, Texas Instruments Inc.
  ==>  extract 'close_vu.c'
  ==>  extract 'close_vu.c'
  ==>  extract 'copy_vup.c'
  ==>  extract 'initvupo.c'
                .
                .
                .
```

Section 3

Math Routines

The math routines are a collection of algebraic, trigonometric, and transcendental functions on real arguments. The library includes:

- Double-precision, floating-point functions that can be called from C.

  These functions conform to the ANSI C standard function names and definitions. The arguments that are input to these functions and the values that are returned from them are double-precision, floating-point numbers. The *TMS34010 C Compiler User's Guide* describes floating-point formats.

  Double-precision, floating-point math errors are reported by means of the customizable fp_error function (discussed in the *TMS34010 C Compiler User's Guide*). An error number is assigned to each type of error; refer to Section 3.2 for a list of errors and their associated error numbers.

- Single-precision functions that can be called from assembly language.

  These functions include addition, multiplication, sine, cosine, and conversion between single-precision floating-point format and 32-bit fixed-point format. The math routines are implemented in TMS34010 assembly language, and call functions contained in the floating-point library `flib.lib`. The *TMS34010 C Compiler User's Guide* describes floating-point formats. The functions use a 32-bit fixed-point format, which is a 2s complement representation with 16 bits of integer and 16 bits of fraction.

- Type-conversion functions that can be called from C.

  These functions convert arrays of numbers between floating-point, fixed-point, short-integer, and long-integer representations. The functions accept an input array of one type and produce an output array of another type.

The algorithms used in the trigonometric, logarithmic, hyperbolic, and exponential functions are adapted from *A Software Manual for the Elementary Functions* (Cody and Waite).

Topics in this section include:

3.1 Double-Precision Functions

These math routines can be called from a C program.

Function Name	Description
`double acos(x)` `double x;`	Returns a double-precision number that represents the arc cosine of x.
`double asin(x)` `double x;`	Returns a double-precision number that represents the arc sine of x.
`double atan(x)` `double x;`	Returns a double-precision number that represents the arc tan of x.
`double atan2(u,v)` `double u,v;`	Returns a double-precision number that represents the arc tangent of u divided by v.
`double ceil(x)` `double x;`	Returns a double-precision number that represents the smallest integer greater than or equal to x.
`double cos(x)` `double x;`	Returns a double-precision number that represents the cosine of x.
`double cosh(x)` `double x;`	Returns a double-precision number that represents the hyperbolic cosine of x.
`double cotan(x)` `double x;`	Returns a double-precision number that represents the cotangent of x.
`double exp(x)` `double x;`	Returns a double-precision number that represents the natural number *e* raised to the power of x (e^x).
`double fabs(x)` `double x;`	Returns a double-precision number that represents the the absolute value of x.
`double floor(x)` `double x;`	Returns a double-precision number that represents the largest integer less than or equal to the value of x.
`double fmod(x,y)` `double x,y;`	Returns a double-precision number that represents the remainder of x divided by y, according to the formula x-y x N, where N is the quotient of x/y truncated to an integer.
`double frexp(value, exp)` `double value;`	Breaks a double-precision number into a normalized fraction and an exponent.

`double ldexp(value, exp)`
`double value;`
`int exp;`
	Returns value x 2^{exp}; commonly used to rebuild a double-precision number.

`double log(x)` `double x;`	Returns a double-precision number that represents the natural logarithm (base e) of x; that is, ln(x).
`double log10(x)` `double x;`	Returns a double-precision number that represents the common logarithm of x; that is, $\log_{10}(x)$.

`double modf(value, exp)`
`double value;`
`int *exp;`
	Breaks a double-precision number into a signed fraction and a signed integer.

`double pow(x,y)` `double x,y;`	Returns a double-precision number that represents x raised to the power of y; that is, x^y.
`double sin(x)` `double x;`	Returns a double-precision number that represents the sine of x.
`double sinh(x)` `double x;`	Returns a double-precision number that represents the hyperbolic sine of x.
`double sqrt(x)` `double x;`	Returns a double-precision number that represents the square root of x.
`double tan(x)` `double x;`	Returns a double-precision number that represents the tangent of x.
`double tanh(x)` `double x;`	Returns a double-precision number that represents the hyperbolic tangent of x.

3.2 Math Routine Error Reporting

Error Code (decimal)	Error Code (hexadecimal)	Error Description	Functions Generating the Error
17	11	**Argument > 1.0 E+8** default result is zero	sin cos
18	12	**abs(argument) > 1.0** default result is $\pm\infty$	asin acos
19	13	**abs(argument) < 1.0 E-300** default result is $\pm\infty$	cotan
20	14	**abs(argument) > 1.0 E+8** default result is zero	cotan tan
21	15	**argument > 500** default result is $\pm\infty$	exp
22	16	**argument < -500** default result is zero	exp
23	17	**Both arguments = 0.0** default result is $+\infty$	atan2
24	18	**X^Y, where X < 0** default result is $(-X)^Y$	pow
25	19	**X^Y, where X=0 and Y≤0** default result is $-\infty$	pow
26	1A	**Argument ≤ 0** default result is $-\infty$	log log10
27	1B	**X^Y results in overflow** default result is $+\infty$	pow
28	1C	**X^Y results in underflow** default result is zero	pow

3.3 Single-Precision Routines

These math routines can be called from assembly language programs by using the EXGPC instruction.

Function Name	Description
FL_ADD	Adds two single-precision floating-point values.
FL_COS	Calculates the cosine of a real number that represents an angle expressed in radians.
FL_MULT	Multiplies two single-precision floating-point values.
FL_SIN	Calculates the sine of a real number that represents an angle expressed in radians.
FIX2FL	Converts a fixed-point number to a single-precision floating-point number.
FL2FIX	Converts a single-precision floating-point number to a fixed-point number.

3.4 Array Conversion Functions

These conversion routines can be called from C programs.

Function Name	Description
fix_to_float	Convert an array of fixed-point numbers to an array of single-precision floating-point numbers.
fix_to_long	Convert an array of fixed-point numbers to an array of long integers.
fix_to_short	Convert an array of fixed-point numbers to an array of short integers.
float_to_fix	Convert an array of single-precision floating-point numbers to an array of fixed-point numbers.
long_to_fix	Convert an array of long integers to an array of fixed-point numbers.
short_to_fix	Convert an array of short integers to an array of fixed-point numbers.

Section 4

Graphics and Text Functions

The Math/Graphics Function Library contains a variety of graphics and text functions. The graphics functions:

- Perform viewport management.
- Produce graphics output, including:
  - Lines,
  - Ellipses,
  - Arcs, **and**
  - Polygons.
- Provide color palette control.

The text functions:

- Provide the capability for drawing bit-mapped text to the screen.
- Allow you to select among a variety of fonts.
- Supply information about text attributes.

The library supports both proportionally spaced and monospaced fonts, and includes several fonts. Additional fonts are available with the TMS34010 Font Library (see the *TMS34010 Font Library User's Guide* for more information).

This section describes the graphics and text functions:

4.1 Summary Table (Functional Grouping)

Graphics System Initialization Functions	
Function	**Description**
clear_screen	Clears the entire screen to a specified pixel value.
init_grafix	Initializes the graphics environment.
init_palet	Sets the color lookup table to default palette values.
init_screen	Clears the screen and sets the palette to default colors.
init_text	Initializes text data structures and installs the system font.
init_video	Initializes video timing and screen refresh registers.
init_vuport	Initializes viewport data structures and opens viewport 0.
new_screen	Clears the entire screen and initializes the color palette.
3D Transformation Matrix Functions	
Function	**Description**
copy_matrix	Copies an input matrix to an output matrix.
copy_vertex	Copies an input vertex list to an output vertex list.
init_matrix	Initializes an array to a 4-by-4 identity matrix.
perspec	Performs perspective transformation on a list of vertices.
rotate	Rotates a 3D matrix in the XY, YZ, and ZX planes.
scale	Scales a matrix in the X, Y, and Z dimensions.
transform	Uses a matrix to transform a vertex list.
translate	Translates a matrix by displacements in X, Y, and Z.
vertex_to_point	Converts a list of 3D vertices to a list of 2D points.
Text Output Functions	
Function	**Description**
draw_char	Draws a single bit-mapped character to the screen.
draw_string	Draws a string of bit-mapped characters to the screen.
Text Attribute Inquiry and Control Functions	
Function	**Description**
add_text_space	Incrementally adjusts horizontal spacing between characters.
char_high	Returns the character height for the selected font.
char_wide_max	Returns the maximum character width in the selected font.
get_ascent	Returns the ascent value for the selected font.
get_descent	Returns the descent value for the selected font.
get_first_ch	Returns the first character represented in the selected font.
get_last_ch	Returns the last character represented in the selected font.
get_leading	Returns the leading value for the selected font.
get_width	Returns the pixel width of the specified character string.
Font Management Functions	
Function	**Description**
get_font_max	Returns the maximum number of installed fonts.
install_font	Installs a font and assigns the designated index.
select_font	Selects a previously installed font.
Graphics Output Functions	
Function	**Description**
bound_fill	Fills to a specified boundary color.
bound_patnfill	Fills to a specified boundary color with pattern.
draw_line	Draws a pixel-thick line between two points.
draw_oval	Draws the outline of an ellipse one pixel thick.
draw_ovalarc	Draws an elliptical arc one pixel in thickness.
draw_piearc	Draws a one-pixel-thick outline of a pie slice of an ellipse.
draw_point	Draws a pixel at designated coordinates.

Graphics Output Functions (continued)	
Function	**Description**
draw_polyline	Draws a list of one-pixel-thick lines.
draw_rect	Draws an outline of a rectangle one pixel thick.
fill_convex	Draws a solid-filled convex polygon.
fill_oval	Draws a solid-filled ellipse.
fill_piearc	Draws a solid-filled pie slice of an ellipse.
fill_polygon	Draws a solid-filled polygon.
fill_rect	Draws a solid-filled rectangle.
frame_oval	Draws a solid elliptical border of specified thickness.
frame_rect	Draws a solid rectangular border of specified thickness.
patnfill_oval	Draws a pattern-filled ellipse.
patnfill_polygon	Draws a pattern-filled polygon.
patnfill_rect	Draws a pattern-filled rectangle.
patnframe_oval	Draws a pattern-filled elliptical border of specified thickness.
patnframe_rect	Draws a pattern-filled rectangular border of specified thickness.
patnfill_convex	Draws a pattern-filled convex polygon.
patnfill_piearc	Draws a pattern-filled pie slice of an ellipse.
patnpen_line	Uses a pen and pattern to draw a line between two points.
patnpen_ovalarc	Uses a pen and pattern to draw an elliptical arc.
patnpen_piearc	Uses a pen and pattern to draw a pie slice of ellipse.
patnpen_point	Draws a pattern-filled pen at designated coordinates.
patnpen_polyline	Uses a pen and pattern to draw a list of lines.
pen_line	Uses a solid pen to draw a line between two points.
pen_ovalarc	Uses a solid pen to draw an elliptical arc.
pen_piearc	Uses a solid pen to draw a pie slice of an ellipse.
pen_point	Draws a solid-filled pen at designated coordinates.
pen_polyline	Uses a pen to draw a list of lines.
seed_fill	Seed (or flood) fills a connected region with a solid color.
seed_patnfill	Seed fills a connected region with a pattern.
styled_line	Draws a styled line between two points.

Pixel and Pixel Array Manipulation Functions	
Function	**Description**
bit_expand	Expands a two-dimensional bit array to colors 0 and 1.
get_pixel	Reads a pixel from the specified coordinates.
get_rect	Copies a rectangular area of the screen into a pixel array.
move_pixel	Moves a pixel from source to destination.
move_rect	Moves pixels from source rectangle to destination.
put_pixel	Writes a pixel to the specified coordinates.
put_rect	Copies a pixel array to a rectangular area of screen.
run_decode	Decompresses a run-length encoded image onto the screen.
run_encode	Stores the on-screen image in run-length encoded form.
zoom_rect	Zooms a source rectangle to fit a destination rectangle.

Graphics Attribute Control Functions	
Function	**Description**
get_patn_max	Returns the maximum number of installed patterns.
get_pmask	Returns the current color plane mask.
get_ppop	Returns the current pixel processing option.
get_psize	Returns the current pixel size.
get_transp	Returns the current transparency flag.
install_patn	Installs a 16-by-16 bit map in the pattern table.
select_patn	Selects a pattern from the pattern table.
set_color0	Sets the background color.
set_color1	Sets the foreground color.
set_pensize	Sets the pen width and height.
set_pmask	Sets the color plane mask.
set_ppop	Sets the pixel processing operation.
transp_off	Disables the pixel attribute of transparency.
transp_on	Enables the pixel attribute of transparency.

Color Palette Functions	
Function	**Description**
color_blend	Blends from starting color to ending color over a specified block of scan lines.
getall_palet	Reads multiple registers in the color lookup table.
set_palet	Loads specified register in the color lookup table.
setall_palet	Loads multiple registers in the color lookup table.
Viewport Management Functions	
Function	**Description**
close_vuport	Closes a viewport.
copy_vuport	Copies attributes of one viewport to another.
cpw	Compares a point to a window (visibility rectangle).
get_vuport_max	Returns the maximum number of open viewports.
move_vuport	Moves a viewport to a new position on screen.
open_vuport	Opens a new viewport and returns its index.
select_vuport	Selects a viewport that is already open.
set_cliprect	Sets the size and position of a clipping rectangle.
set_origin	Sets the position of a viewport-relative origin.
size_vuport	Changes the size of a viewport.
Miscellaneous Graphics Functions	
Function	**Description**
delay	Waits for the specified number of ticks (tick = 1/60 second).
lib_id	Returns a string identifier for the library revision.
lmo	Returns the bit number of the leftmost one in an argument.
peek	Returns the specified word in memory.
peek_breg	Returns the specified B-file register.
poke	Writes a value to the specified word in memory.
poke_breg	Writes a value to the specified B-file register.
rep_pixel	Replicates a pixel value throughout a 32-bit integer.
rmo	Returns the bit number of the rightmost one in an argument.
wait_scan	Waits for the designated horizontal line to be scanned.
xytoaddr	Converts XY coordinates to the memory address of pixel.

4.2 Graphics System Initialization Functions

These functions initialize the software graphics environment by:

- Setting selected I/O registers and B-file registers to appropriate initial values,
- Assigning default values to key variables,
- Clearing the screen, **and**
- Setting the color palette to default values.

Table 4-1 summarizes the graphics system initialization functions.

Table 4-1. Summary of Graphics System Initialization Functions

Function Name	Description
clear_screen	Clears the frame buffer to the specified pixel value.
init_grafix	Initializes the graphics environment. This function must be called before using any of the graphics output functions.
init_palet	Loads the default color palette into the color lookup table.
init_screen	Clears the frame buffer to the specified pixel value and loads the default color palette values into the color lookup table.
init_text	Initializes the text data structures, and opens the system font as font number 0. This function must be called before using any of the text functions.
init_video	Initializes CRT control timing, enables screen refresh, and defines screen dimensions and pixel size. This function must be called before using the initialization functions init_grafix and init_vuport.
init_vuport	initializes viewport data structures, and opens system viewport 0 as the system viewport. This function must be called before using any of the viewport functions.
new_screen	Clears the frame buffer to the specified pixel value, and loads the specified color palette values into the color lookup table.

4.3 3D Transformation Functions

These functions allow you to rotate, scale, translate, and apply perspective transformations to points in three-dimensional space. You can use these functions to construct a a 4×4 homogeneous transform matrix, and transform the object that the matrix represents through a series of rotations, scalings, and translations. You can then use the resulting matrix to transform an object represented as a list of 3D points.

The Z coordinate axis used for 3D operations is assumed to be perpendicular to the face of the screen. The positive Z direction moves away from a person who is looking at the screen (into the screen). The face of the screen is at Z=0. The X axis is horizontal along the face of the screen; the positive X direction is left to right. The Y axis is vertical along the face of the screen; the positive Y direction is top to bottom. You can use the viewport functions included in the library to move the XY origin used for graphics output. The default location of the origin is at the top left corner of the screen; this is the location of the origin following initialization.

Points in three-dimensional space are represented in a data structure called a **vertex list**. Each point in the vertex list is represented by three coordinates, (X,Y,Z). The vertex list can be converted to a list of two-dimensional points, each specified by two coordinates, (X,Y), to facilitate their use by graphics output functions such as draw_polyline and fill_polygon.

Vertex list elements and transformation elements are represented as 32-bit fixed-point values. The fixed-point format is not a standard C data type, but is an internal data format used by several library functions. A fixed-point value is a 32-bit, 2s complement value whose 16 LSBs lie to the right of the binary point.

Table 4-2 summarizes the matrix functions. Refer to *Principles of Interactive Computer Graphics* (Newman and Sproull) for a description of homogeneous coordinate transformations.

Table 4-2. Summary of 3D Transformation Functions

Function Name	Description
copy_matrix	Copies a 4×4 matrix.
copy_vertex	Copies a vertex list (list of three-dimensional vertices).
init_matrix	Initializes a 16-element array to a 4×4 identity matrix.
perspec	Performs perspective transformation on a vertex list (list of three-dimensional vertices).
rotate	Rotates a matrix by specified angles in the XY, YZ, and ZX planes.
scale	Scales a matrix by specified scaling factors in the X, Y, and Z dimensions.
transform	Uses a 4×4 transformation matrix to transform a list of three-dimensional vertices.
translate	Translates a matrix by specified displacements in the X, Y, and Z directions.
vertex_to_point	Converts a vertex list (list of three-dimensional vertices) to a list of two-dimensional points.

4.4 Text Output Functions

The text output functions draw bitmapped text to the screen. The library supports both proportionally spaced and monospaced fonts. Text can be placed at arbitrary positions on the screen. Table 4-3 summarizes the text output functions.

Table 4-3. Summary of Text Output Functions

Function Name	Description
draw_char	Draws a single character to the screen using the current font.
draw_string	Draws a string of characters to the screen using the current font.

4.5 Text Attribute Inquiry and Control Functions

These functions provide you with values that are associated with text attributes. Several attributes, including ascent, descent, leading, character height, and character width, are associated with each font in the font library. Figure 4-1 illustrates these attributes.

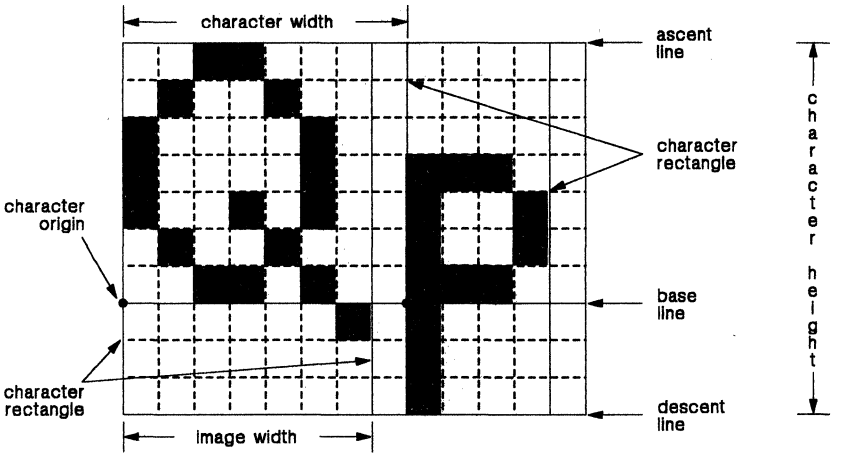


Figure 4-1. Text Attributes

- The **character origin** is a point on the baseline that is positioned at the left side of the character pattern. This point is used as a reference for locating the character to be drawn.

- The **character rectangle** is a rectangle that encloses the character image. The sides of the rectangle are defined by the image width and the character height. For example, in Figure 4-1, the character rectangle for ***Q*** is seven pixels wide and ten pixels high.

- The **character width** is the image width plus the space separating this character from the next character. The character width can vary within a font and between fonts.

- The **image width** is the width in pixels of the portion of the character pattern bitmap that contains the character image, excluding the space to the right of the character. The image width can vary within a font and between fonts.

- The **ascent line** is a horizontal line that coincides with the top of the highest-reaching character in the font. The ascent value is the difference between the Y coordinates of the ascent line and the baseline.

- The **baseline** is an imaginary horizontal line that coincides with the bottom of each character, excluding descenders. The starting Y coordi-

nate given to the draw_string or draw_char function specifies the Y value of the baseline. The starting X coordinate specifies the left edge of the first character. The starting X and Y coordinates together define the character origin of the first character in the string.

- The **descent line** is a horizontal line that coincides with the bottom of the character having the lowest-reaching descender in the font. The descent value is the difference between the Y coordinates of the descent line and the baseline.

- The **character height** is the vertical separation between successive rows of text, and is the sum of the ascent, descent, and leading values.

- The **leading** is the vertical spacing between the bottom of one row of characters and the top of the row of characters below it. The leading value is calculated as difference in Y coordinates of the descent line of one row of characters and the ascent line of the row of characters below that row.

Table 4-4 summarizes the text attribute functions. Refer to the *TMS34010 Font Library User's Guide* for additional information on the font data structure.

Table 4-4. Summary of Text Attribute Inquiry Functions

Function Name	Description
add_text_space	Specifies a value to be added to the default horizontal spacing between characters.
char_high	Returns the character height for the current font.
get_first_ch	Returns the first character represented in the current font.
get_last_ch	Returns the last character represented in the current font.
char_wide_max	Returns the maximum character width in the current font.
get_ascent	Returns the ascent value for the current font.
get_descent	Returns the descent value for the current font.
get_leading	Returns the leading value for the current font.
get_width	Returns the width (in pixels) of a specified character string drawn in the current font.

4.6 Font Management Functions

The font management functions allow you to install a text font and select one of the installed fonts for use. The function library includes several bit-mapped text fonts; additional fonts are available with the TMS34010 Font Library. Table 4-5 summarizes the font management functions.

Table 4-5. Summary of Font Management Functions

Function Name	Description
get_font_max	Returns the maximum number of fonts that can be installed.
install_font	Installs a new font, and assigns an index to it.
select_font	Selects a previously installed font.

Each font is fully described by a font data structure, which contains the character pattern bitmap and other information necessary to extract individual character patterns from the bitmap. Figure 4-2 shows the C definition for this structure.

```
/*-------------------------------------------------------------
 * TMS34010 Graphics Function Library
 *-------------------------------------------------------------
 * font_struct data structure
 *
 * Data structure for storage of text font bit map and attributes.
 * The charpatn[], loctable[], and owtable[] arrays vary in size
 * according to the font. This is the reason they are given dummy
 * declarations below. The routines that manipulate these arrays
 * are written in assembly code.
 *-------------------------------------------------------------
 */
struct font_struct {
    short fonttype;    /* font type code                          */
    short firstchar;   /* ASCII code of first character           */
    short lastchar;    /* ASCII code of last character            */
    short widemax;     /* maximum character width                 */
    short kernmax;     /* maximum character kerning amount        */
    short ndescent;    /* negative of the descent value           */
    short frectwide;   /* width of font rectangle                 */
    short charhigh;    /* character height                        */
    short owtloc;      /* offset to offset/width table            */
    short ascent;      /* ascent (how far above baseline)         */
    short descent;     /* descent (how far below baseline)        */
    short leading;     /* leading (row bottom to next row top)    */
    short rowwords;    /* no. of words per row of char patterns   */
/*  short charpatn[n];    character pattern bitmap                 */
/*  short loctable[n];    character offsets in charpatn            */
/*  short owtable[n];     offset/width table                       */
};
typedef struct font_struct FONT;
```

Figure 4-2. Font Data Structure

and thus can be explicitly manipulated in a C program. The last three members (arrays `charpatn`, `loctable`, and `owtable`) have variable lengths; they are given dummy declarations in the structure definition above, and are manipulated by assembly language routines. The font structure members are defined as follows:

`fonttype` *Font type code.* This code designates the type of font. Proportionally spaced fonts are identified by a fonttype value of 9000h. (The software treats even monospaced fonts such as Corpus—Christi as proportionally spaced fonts.)

`firstchar` *ASCII code of first character.* The `firstchar` value identifies the lowest ASCII code for which a character pattern (other than the missing character pattern) is provided.

`lastchar` *ASCII code of last character.* The `lastchar` value identifies the highest ASCII code for which a character pattern (other than the missing character pattern) is provided.

`widemax` *Maximum character width.* The `widemax` value is the character width of the widest character in the font. It equals the sum of the image width and the space to the right of the character image.

`kernmax` *Maximum character kerning amount.* The font structure permits character kerning; that is, a character's pattern may overlap the space occupied by the character to its left. The kerning amount is the number of horizontal pixels of overlap, and is equal to the portion of the image width of the character that falls to the left of the character origin. It should always be 0 or negative. The `kernmax` is the maximum kerning amount for all characters in the font.

`ndescent` *Negative of the descent value.* The `ndescent` value is the negative (2s complement) of the number of pixels between the baseline and the descent line for the font.

`frectwide` *Width of font rectangle.* The `frectwide` is the width of the widest character image in the font. Whereas `widemax` represents the sum of the image width and the space to the right of the character, `frectwide` represents only the image width.

`charhigh` *Character height.* The `charhigh` value represents the vertical distance (in pixels) between the baseline of one row of text and the baseline of the following row. It equals the sum of the ascent and the descent. The `charhigh` value is also the number of rows in the bitmap representing the character patterns for the font.

`owtloc` *Offset to offset/width table.* The `owtloc` value is the word offset from itself to the start of the `owtable` array. It is equivalent to 4+(`rowwords` x `charhigh`) + (`lastchar-firstchar`+3)+1.

`ascent` *Ascent.* The `ascent` value is the vertical distance (in pixels) from the baseline to the ascent line for the font.

`descent` *Descent.* The `descent` value is the vertical distance (in pixels) from the descent line one row of text to the ascent line of the next row of text beneath it.

`leading` *Leading.* The `leading` is the vertical distance (in pixels) from the descent line of one row of text to the ascent line of the next row of text beneath it.

`rowwords` *Number of words per row of character pattern bitmap.* The `rowwords` value is the width (in words) of the bitmap containing the patterns for the characters in the font. The pitch (width in bits) of the bitmap is obtained by multiplying `rowwords` by 16.

`charpatn[n]` *Character pattern bitmap.* Figure 4-1 shows the format of the character pattern bitmap contained in the `charpatn` array. The last character image in the bitmap is that of the missing symbol. The top row of the bitmap contains the top row of the image for each character in the font; the second row contains the second row of the image for each character; and so on. The storage space in words required by the bitmap is calculated as (`rowwords` × `charhigh`).

`loctable[n]` *Table for locating individual character images in the character pattern bitmap.* The `loctable` member corresponding to a particular ASCII character code gives the offset in bits from the start of the `charpatn` array to the start of the image for that character. The `loctable` array contains entries only for ASCII characters `firstchar` through `lastchar`, plus the missing character. The total number of members in the `loctable` array is equal to *n* = (`lastchar-firstchar+3`). The location of the first character is contained in `loctable[0]`; the location of the last character is contained in `loctable[lastchar-firstchar]`. The location of the missing character is contained in `loctable[lastchar-firstchar+1]`. The final array member, `loctable[lastchar-firstchar+2]`, points to one bit beyond the end of the top row of the missing character image. The offset of the image for an ASCII code *i* from the base address of the `charpatn` array is contained in `loctable[`*i*`-firstchar]`. The width of the image is calculated as *w* = (`loctable[`*i*`-firstchar+1] - (loctable[`*i*`-firstchar]`. If the character represented by ASCII code *i* is missing from the table, the `loctable` member representing that character has the same value as the member for character *i*-1: `loctable[`*i*`-firstchar] = loctable[`*i*`-firstchar+1]`.

`owtable[n]` *Offset/width table.* The `owtable` array is a table of values giving the character offset and character width for each character in the font; the 8 MSBs of each member give the character offset, and the 8 LSBs give the character width. However, if an ASCII code *i* represents a character whose pattern is missing from the font, then `owtable[`*i*`-firstchar]`, the offset/width table entry for that character, is set to -1 (all 1s). The `owtable` array contains entries only for ASCII characters `firstchar` through `lastchar`, and also for the missing character. The `owtable` array has the same number of members as the `loctable` array. The offset/width value for the first character is contained in `owtable[0]`. The offset/width value for the last (nonmissing) character is contained in `owtable[lastchar-firstchar]`. The offset/width value for the missing symbol is contained in `owtable[lastchar-firstchar+1]`. The last member of the array, `owtable[lastchar-firstchar+2]`, contains a value of -1.

4.7 Available Fonts

The math/graphics function library includes several fonts that can be used with the library's text functions. The TMS34010 Font Library provides additional fonts (see the *TMS34010 Font Library User's Guide*).

Figure 4-3 shows the five fonts that are included in the math/graphics function library.

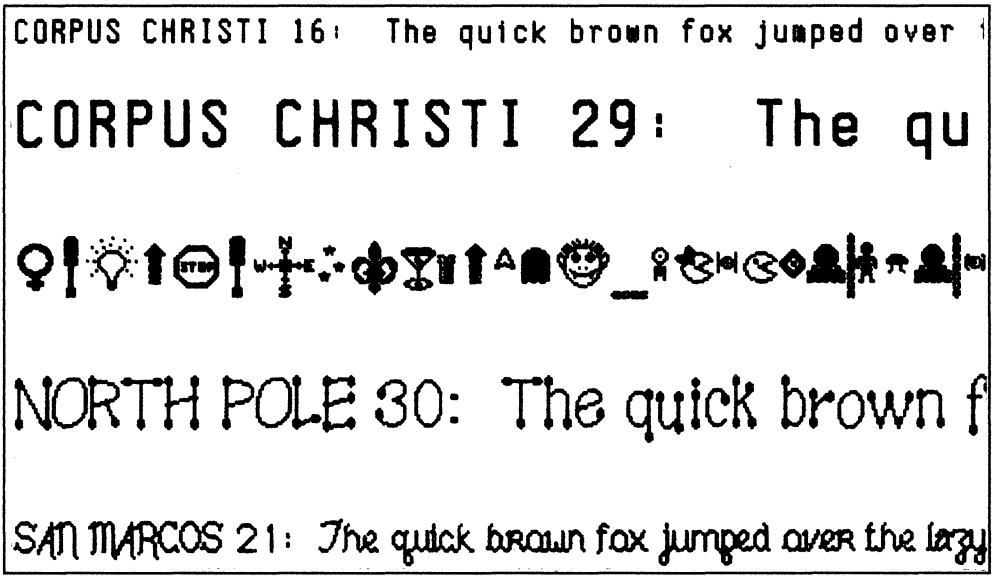


Figure 4-3. Five Fonts

The font names are listed in Table 4-6 along with the font structure names that are used within a C program to install the font in the font table.

Table 4-6. Installable Font Symbols

Font Name	Symbol Name
CORPUS CHRISTI 16	corpus_christi16
CORPUS CHRISTI 29	corpus_christi29
MONTROSE 28	montrose28
NORTH POLE 30	north_pole30
SAN MARCOS 21	san_marcos21

The program in Figure 4-4 draws the text display shown in Figure 4-3.

```
/*-----------------------------------------------------------------
 *                  Show available fonts.
 *-----------------------------------------------------------------
 */
#include "fntstruc.h"       /* Define FONT structure as data type. */

 extern FONT corpus_christi29,  /* Corpus Christi font, size = 29 */
             montrose28,        /* Montrose font,       size = 28 */
             north_pole30,      /* North Pole font,     size = 30 */
             san_marcos21;      /* San Marcos font,     size = 21 */

 static char *cap[] = {
     "CORPUS CHRISTI 16:  ", "CORPUS CHRISTI 29:  ",
     "MONTROSE 29:  ", "NORTH POLE 30:  ", "SAN MARCOS 21:  "
 };

 main()
 {
     char *s;   /* text              */
     int x, y;  /* text coordinates */
     int i;     /* loop counter      */

     init_video(1);
     init_grafix();
     init_text();  /* Corpus Christi 16 selected as default font */
     init_screen();
                   /* Remember the '&' symbol!                   */
     install_font(1, &corpus_christi29);
     install_font(2, &montrose28);
     install_font(3, &north_pole30);
     install_font(4, &san_marcos21);

     s = "The quick brown fox jumped over the lazy sleeping dog.";
     for (i = 0, y = 50; i <= 4; ++i) {
         y += char_high();                  /* character height */
         select_font(i);
         y += char_high();                  /* character height */
         x = draw_string(0, y, cap[i]);
         draw_string(x, y, s);
     }
 }
```

Figure 4-4. Drawing the Text Display for Figure 4-3

4.8 Graphics Output Functions

The graphics functions draw several shapes in a variety of styles. Table 4-7 describes the figure types and drawing styles. Table 4-8 shows which shapes can be drawn in a particular style. The column headers list the available styles and the row headers list the available shapes; a check mark indicates that a shape can be drawn with a particular style. Table 4-9 (page 4-17) describes the individual graphics output functions.

Table 4-7. List of Figure Types and Drawing Styles

Figure Types	
Function Name	**Description**
bound	Fill bounded set of pixels beginning at specified start point.
line	A straight line.
oval	Ellipse in standard position (major and minor axes parallel with coordinate axes).
ovalarc	An arc of an ellipse in standard position, specified in terms of beginning and ending angles.
point	A single pixel or pen image drawn at the indicated XY coordinate pair.
polygon	A filled region defined by a collection of straight edges. Both convex polygons and arbitrarily complex polygons are supported.
polyline	A collection of straight lines. Figures made up of many lines can be drawn more efficiently by using the polyline commands than by repeated calls to the line functions.
piearc	Pie arc or wedge. Similar to ovalarc, but with addition of sides drawn from center of ellipse to arc endpoints to produce closed figure.
rect	Rectangle with vertical and horizontal sides.
seed	Fill connected set of pixels beginning at specified seed point.
Drawing Styles	
Function Name	**Description**
draw	Draws figure outline one pixel wide using COLOR1.
fill	Draws figure interior filled in solid COLOR1.
frame	Draws frame in solid COLOR1. Horizontal and vertical thicknesses of frame border are both specified.
patnframe	Draws frame using pattern in COLOR0 and COLOR1. Horizontal and vertical thicknesses of frame border are both specified. The 16-by-16 pattern is programmable.
patnpen	Draws figure outline using pen and pattern in COLOR0 and COLOR1. Pen size and 16-by-16 pattern are programmable.
pen	Draws figure outline using pen in solid COLOR1. Pen is rectangular with programmable height and width.
patnfill	Draws figure interior filled with pattern in COLOR0 and COLOR1. The 16-by-16 pattern is programmable.

Table 4-8. Checklist of Available Figure Types and Drawing Styles

	Drawing Style						
Figure Type	**draw**	**pen**	**patnpen**	**fill**	**patnfill**	**frame**	**patnframe**
bound				√	√		
line	√	√	√				
oval	√			√	√	√	√
ovalarc	√	√	√				
piearc	√	√	√	√	√		
point	√	√	√				
polygon				√	√		
polyline	√	√	√				
rect	√			√	√	√	√
seed				√	√		

Table 4-9. Summary of Graphics Output Functions

Function Name	Description
bound_fill	Fills a bounded set of pixels given a starting point and a boundary color.
bound_patnfill	Fills a bounded set of pixels with the current pattern given a starting point and a boundary color.
draw_line	Draws a straight line one pixel thick.
draw_oval	Draws the outline of an ellipse. The outline is one pixel in thickness.
draw_ovalarc	Draws an elliptical arc one pixel thick.
draw_piearc	Draws the outline of a pie slice of an ellipse. The outline is one pixel thick.
draw_point	Draws a pixel at the specified coordinates.
draw_polyline	Draws a list of lines one pixel thick.
draw_rect	Draws the outline of a rectangle. The sides are one pixel in thickness.
fill_convex	Fills a convex polygon.
fill_oval	Draws a solid-filled ovalangle.
fill_piearc	Draws a solid-filled pie slice of an ellipse.
fill_polygon	Draws a solid-filled polygon given a list of edges.
fill_rect	Draws a solid-filled rectangle.

Table 4-9. Summary of Graphics Output Functions (Concluded)

Function Name	Description
frame_oval	Draws an elliptical frame of specified thickness. The frame border is solid-filled.
frame_rect	Draws a rectangular frame of specified thickness. The frame border is solid-filled.
patnfill_convex	Fills a convex polygon with the current pattern.
patnfill_oval	Draws an ellipse filled with the current pattern.
patnfill_piearc	Draws a pattern-filled pie slice of an ellipse.
patnfill_polygon	Draws a pattern-filled polygon given a list of edges.
patnfill_rect	Draws a rectangle filled with the current pattern.
patnframe_oval	Draws an elliptical frame of specified thickness. The frame border is filled with the current pattern.
patnframe_rect	Draws a rectangular frame of specified thickness. The frame border is filled with the current pattern.
patnpen_line	Draws a straight line using the drawing pen and the current pattern.
patnpen_ovalarc	Draws an elliptical arc using the drawing pen with the current pattern.
patnpen_piearc	Draws the outline of a pie slice of an ellipse using the drawing pen and the current pattern.
patnpen_point	Draws the pen with the current pattern at the specified coordinates.
patnpen_polyline	Draws a list of lines using the drawing pen and the current pattern.
pen_line	Draws a straight line using the drawing pen.
pen_ovalarc	Draws an elliptical arc using the drawing pen.
pen_piearc	Draws the outline of a pie slice of an ellipse using the drawing pen.
pen_point	Draws the pen at the specified coordinates.
pen_polyline	Draws a list of lines using the drawing pen.
seed_fill	Fills a connected set of pixels given a seed point.
seed_patnfill	Fills a connected set of pixels with a pattern given a seed point.
styled_line	Draws a styled line using the specified 32-bit line-style pattern.

4.9 Graphics Attribute Control Functions

These functions allow you to select and enable a variety of graphics attributes, including:

- Foreground (COLOR1) and background (COLOR0) colors,
- Pixel transparency,
- Pixel processing operation code,
- Plane mask,
- Pen width and height, **and**
- 16 x 16 fill pattern.

Table 4-10 summarizes these functions. Refer to the *TMS34010 User's Guide* for descriptions of COLOR0, COLOR1, transparency, pixel processing, and the plane mask.

Table 4-10. Summary of Graphics Attribute Control Functions

Function Name	Description
get_patn_max	Gets the maximum number of patterns that can be installed in pattern table at one time.
get_pmask	Gets the current plane mask.
get_ppop	Gets the current pixel processing option.
get_psize	Gets the current pixel size.
get_transp	Gets the current transparency flag.
install_patn	Installs a new pattern in pattern table.
select_patn	Selects a pattern that is already installed in pattern table.
set_color0	Sets the background color used for text, bitmap expansion, and patterns.
set_color1	Sets the foreground drawing color used for vectors, fills, text, bitmap expansion and patterns.
set_pensize	Sets the width and height of rectangular drawing pen.
set_ppop	Sets the pixel processing operation code.
set_pmask	Sets the plane mask, enabling or disabling individual color planes as specified.
transp_off	Disables the pixel attribute of transparency.
transp_on	Enables the pixel attribute of transparency.

4.10 Fill Patterns

Graphics functions which include ***patn*** as part of their names draw with a pattern instead of a solid color. The pattern is specified as a 16×16 bitmap, and is represented in memory as an array of 256 contiguous bits. The bits in a pattern are listed in left-to-right order within a row, and rows listed in top-to-bottom order. You must install a pattern in the library's pattern table before you call a function that draws with the pattern.

Figure 4-5 shows an example of a pattern as it appears on the screen. The small squares represent individual bits in the pattern; shaded squares represent 1s and white squares represent 0s. The bit at the top left corner is the first bit (bit 0) in the pattern array. The bit at the lower right hand is the last bit (bit 255) in the array.

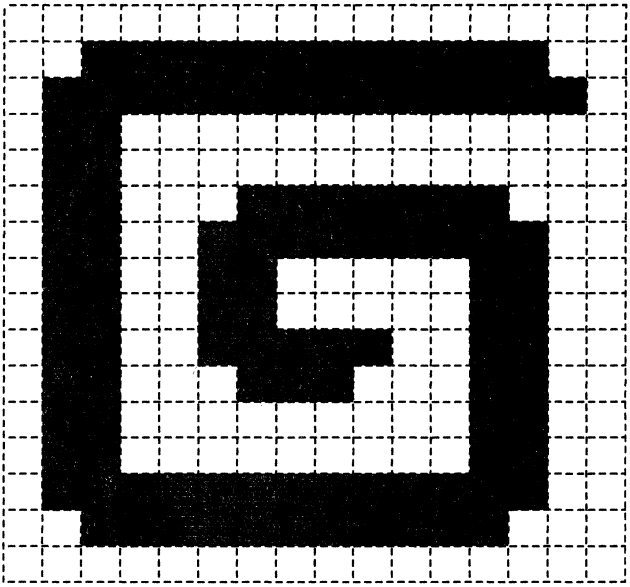


Figure 4-5. A 16×16 Pattern

When a pattern is drawn to the screen, the 0s in the bit map are replaced with COLOR0, and the 1s in the bit map are replaced with COLOR1. The pattern is mapped into 16×16 cells on the screen. The X and Y coordinates at the top left corner of each cell are both multiples of 16.

The library supports several patterns that are installed as the default patterns when the init_grafix function is called. You can view the default patterns by executing the program shown in Figure 4-6.

```
/*-----------------------------------------------------------------
 *       Show available patterns in deck-of-cards display.
 *-----------------------------------------------------------------
 */
main()
{
    int x, y, dx, dy, i, hue0, hue1;

    init_video(1);
    init_grafix();          /* Initialize default patterns. */
    init_screen();
    i = get_patn_max();
    dx = 480 / i;
    dy = 320 / i;
    x = y = 0;
    hue0 = hue1 = 0;
    for (--i; i >= 0; --i, x += dx, y += dy) {
        select_patn(i);
        set_color0(rep_pixel(hue0++));
        set_color1(rep_pixel(--hue1));
        patnfill_rect(160, 160, x, y);
    }
}
```

Figure 4-6. Program for Displaying the Default Patterns

You can use the install_patn function to install your own pattern in any position in the pattern table. The program in Figure 4-7 installs and displays the pattern shown in Figure 4-5.

```
/*-----------------------------------------------------------------
 *           Install a pattern in the pattern table.
/*-----------------------------------------------------------------
 */
main()
{
    typedef enum { FIELDWIDTH = 1 } BIT;
    static BIT mypatn[] = {
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
        0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
        0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,
        0,1,1,0,0,1,1,1,1,1,1,1,1,1,0,0,
        0,1,1,0,0,1,1,0,0,0,0,0,1,1,0,0,
        0,1,1,0,0,1,1,0,0,0,0,0,1,1,0,0,
        0,1,1,0,0,1,1,1,1,1,1,0,0,1,1,0,0,
        0,1,1,0,0,0,1,1,1,0,0,0,1,1,0,0,
        0,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,
        0,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,
        0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
        0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    };

    init_video(1);
    init_grafix();              /* Installs default patterns. */
    init_screen();
    install_patn(5, mypatn);    /* Assign index = 5.          */
    set_color0(rep_pixel(4));   /* Expand 0s to this color.   */
    set_color1(rep_pixel(7));   /* Expand 1s to this color.   */
    patnfill_oval(448, 288, 96, 96);
}
```

Figure 4-7. Program for Installing Fonts

4.11 Color Palette Functions

These functions allow you to read and modify the color lookup table that is used to translate pixel values into colors on the screen. In systems that support color indexing, the color associated with each possible pixel value is specified in the corresponding entry in a color lookup table. The colors seen on the display are generated by using the pixel values from the display memory as indices into the lookup table. Each table entry specifies the red, green, and blue intensities that make up the color.

The color lookup table is loaded into the internal registers of a color-palette device such as a TMS34070 color-palette chip. The driver routines that implement the palette functions are necessarily device dependent. The implementation that runs on the TMS34010 Software Development Board assumes a pixel size of four bits and a TMS34070 color palette that is configured in line-load mode. The set_palet function changes the color associated with a specified pixel value, and is relatively easy to emulate in systems containing other types of palette devices. More difficult to emulate are the functions which make use of the ability of the TMS34070 color-palette chip to load its registers with a new lookup table at the beginning of each scan line in the frame. With this ability, a different color palette can be assigned to each scan line or group of scan lines. Three functions use this feature of the TMS34070 color palette: setall_palet, getall_palet and color_blend.

Refer to the *TMS34070 User's Guide* for information about the TMS34070 color palette. Refer to the *TMS34010 Software Development Board User's Guide* for information on configuring the palette on the SDB to operate in line-load mode.

Table 4-11 summarizes the color palette functions.

Table 4-11. Summary of Color Palette Functions

Function Name	Description
color_blend	Creates gradual changes in shading, highlights, and color blending effects by gradually varying the red, green and blue intensities of the color associated with a specified pixel value on a line-by-line basis.
getall_palet	Reads multiple registers of the TMS34070 color palette. The palette for the specified scan line is returned.
set_palet	Changes the color that is associated with a specified pixel value.
setall_palet	Loads multiple registers of the TMS34070 color palette. The palette is affected only over a specified group of scan lines.

4.12 Pixel and Pixel Array Manipulation Functions

These functions copy and process individual pixels and two-dimensional arrays of pixels. (2D pixel arrays correspond to rectangular areas of the screen.) Table 4-12 summarizes the pixel array functions.

Table 4-12. Summary of Pixel Array Functions

Function Name	Description
bit_expand	Expands a bitmap to the specified rectangular area of the screen. The expansion process replaces the 1s in the bitmap with COLOR1 and replaces the 0s with COLOR0.
get_pixel	Returns the value of the specified pixel on the screen.
get_rect	Captures a rectangular area of the screen into the specified pixel array.
move_pixel	Copies a pixel from one screen location to another.
move_rect	Copies the pixels in a rectangular area of the screen to another rectangular area of the same size.
put_pixel	Copies the specified value to the specified pixel on the screen.
put_rect	Copies an array of pixels to a rectangular area of the screen.
run_decode	Decompresses a previously run-length encoded image and copies the image to a specified location on the screen.
run_encode	Uses run-length encoding to compress an image contained in a specified rectangular area of the screen.
zoom_rect	Zooms the pixels in the specified source rectangle on the screen to fit the specified destination rectangle on the screen.

4.13 Viewport Management Functions

These functions allow you to change the position and size of a viewport. They also allow you to change the relative origin and clipping rectangle associated with the viewport.

A viewport is a rectangular area of the screen; drawing can occur within the boundaries of a viewport. Multiple viewports can be open simultaneously, but only one viewport is active at a time. Drawing operations can take place only within the active viewport. All graphics output is automatically clipped so that only pixels lying inside the boundaries of the viewport are drawn.

When a viewport is moved or resized, anything previously drawn to the screen is not affected; changes in the viewport *do* affect subsequent drawing operations. Similarly, when the relative origin is moved, or the clipping rectangle is changed, only subsequent drawing operations are affected.

Figure 4-8 shows an example in which two viewports, 0 and 6, are open. A relative XY origin and a clipping rectangle are associated with each viewport in Figure 4-8. All graphics output is drawn in the coordinate system defined by the relative origin associated with the active viewport. The clipping rectangle allows you to restrict drawing to a rectangular region within the viewport. Drawing can occur only in the visibility rectangle represented by the intersection of the active viewport, the clipping rectangle associated with that viewport, and the screen.

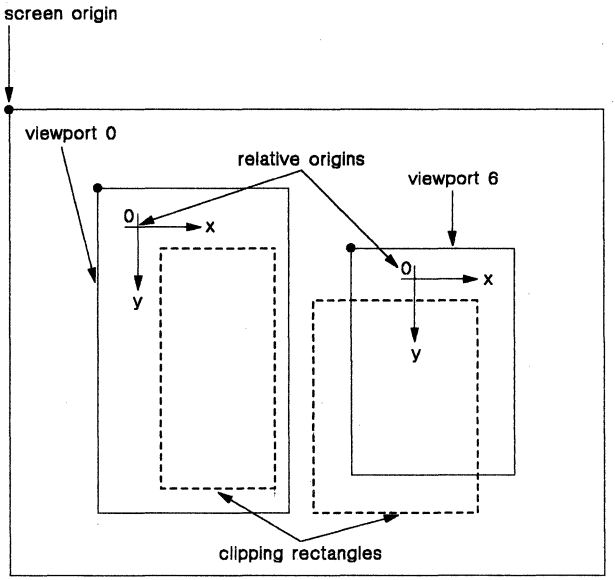


Figure 4-8. Two Viewports

Applications that do not require viewports, clipping rectangles, or relative origins can ignore them. The init_grafix function establishes a default viewport

for which the visibility rectangle is the entire screen. It also positions the XY origin at the top left corner of the screen. Applications that use viewports, clipping rectangles, or relative origins must call the init_vuport function before calling any of the viewport functions.

A set of graphics and text attributes are associated with each open viewport. When a viewport is activated, the state of these attributes at the end of the previous activation (of the same viewport) is automatically restored. The following is a list of attributes associated with each viewport:

- The **viewport** width and height and the **XY coordinates** of its top left corner (specified as displacements from the top left corner of the screen).
- A **relative origin** that is specified in terms of its X and Y displacements from the top left corner of the viewport.
- A **clipping rectangle** that is specified in terms of its width and height, and the XY coordinates at its top left corner, relative to the viewport's relative origin.
- A **pattern index** that indicates the current pattern.
- A **font index** that indicates the current font.
- A text horizontal **spacing increment** that indicates the amount by which the default spacing between characters is modified. The increment can be positive, negative, or zero. (See the add_text_space function.)
- The current width and height of the **drawing pen**.
- The current **COLOR0** that defines the background color specified for text and pattern drawing.
- The current **COLOR1** that defines the foreground color specified for text and pattern drawing, and the drawing color used for lines, solid fills, and so on.
- The current **pixel processing operation** code that identifies one of the 22 pixel processing operations performed by the TMS34010.
- A **transparency** flag that indicates whether the pixel transparency attribute is currently enabled or disabled.
- The current **plane mask** that indicates which color planes are enabled and disabled during graphics output operations.

When you move a viewport, you must specify the viewport's new position (as measured from its top left corner) relative to the screen origin, which lies at the top left corner of the screen. When you move the viewport's relative XY origin, you must specify the new origin in terms of its X and Y displacements from the top left corner of the viewport. When the viewport position is changed, the relative origin moves with it. The position of the clipping rectangle is defined in terms of the relative origin. When the position of either the viewport or the relative origin is changed, the position of the clipping rectangle moves accordingly.

Table 4-13 summarizes the viewport management functions.

Table 4-13. Summary of Viewport Management Functions

Function Name	Description
close_vuport	Closes a viewport that was previously opened.
copy_vuport	Copies the attributes from one viewport to another. Both viewports must be open.
cpw	Compares point to window and returns 4-bit outcode.
get_vuport_max	Gets maximum number of viewports that can be open at once.
init_vuport	Initializes the viewport data structures, and opens the system viewport as viewport 0.
move_vuport	Moves the viewport to a new position on the screen. The position is specified in terms of X and Y displacements from the top left corner of the screen.
open_vuport	Opens a new viewport.
select_vuport	Activates a viewport that is already open.
set_cliprect	Sets the clipping rectangle to the specified width and height, and moves it to the specified position. The position is specified in terms of X and Y displacements from the viewport-relative origin.
set_origin	Sets the XY origin for the viewport to the new position. The position is specified in terms of X and Y displacements from the top left corner of the viewport.
size_vuport	Changes the width and height of a viewport as specified. The position of the top left corner of the viewport remains fixed while the bottom right corner is adjusted to accommodate the new dimensions.

4.14 Miscellaneous Functions

Table 4-14 summarizes the functions that are not described in the previous categories.

Table 4-14. Summary of Miscellaneous Functions

Function Name	Description
delay	Delays the specified number of ticks (tick = 1/60 second).
lib_id	Gets character string specifying current revision of function library.
lmo	Returns the bit number of the leftmost one in the 32-bit argument.
peek	Fetches a 16-bit word from the specified address in memory.
peek_breg	Reads the contents of a specified 32-bit B-file register.
poke	Pokes a 16-bit word into the specified address in memory.
poke_breg	Loads a specified B-file register with a 32-bit value.
rep_pixel	Replicates a pixel value throughout a 32-bit integer.
rmo	Returns the bit number of the rightmost one in the 32-bit argument.
wait_scan	Waits until the electron beam has finished scanning the specified horizontal line of the CRT.
xytoaddr	Converts viewport-relative XY coordinates to 32-bit memory address of a pixel.

4.15 Special Data Formats

The library's graphics functions use the following four data formats:

- Transformation matrix,
- Vertex list,
- Point list, **and**
- Line list.

These four data formats specify the organization of information that is passed between the function library routines and an application program. They differ from the text font storage structure described earlier, which is managed automatically by the text functions. The transformation matrix, vertex list, point list, and line list are described in the following paragraphs.

4.15.1 Transformation Matrix

The transformation matrix is a 4 x 4 matrix that is stored in a 16-element array of 32-bit fixed-point values. The 32-bit fixed-point format places the 16 LSBs to the right of the binary point, partitioning the value into a 16-bit 2s-complement integer Matrix elements are mapped into the array in row major order, as illustrated in Figure 4-9.

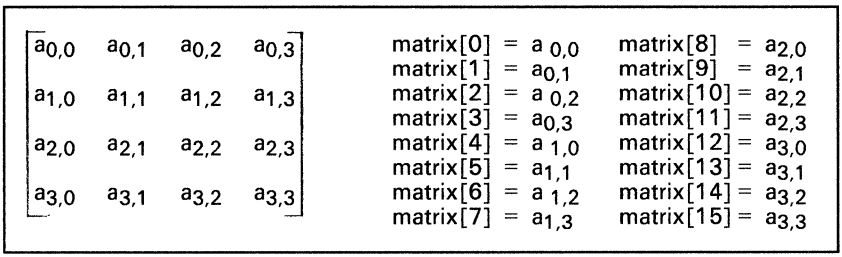


Figure 4-9. Transformation Matrix Format

4.15.2 Vertex List

The vertex list is an array of values that represent a collection of points in three-dimensional space. Each point is specified in terms of its X, Y, and Z coordinates. Each coordinate is represented as a 32-bit fixed-point value. Figure 4-10 illustrates the vertex list format. The X, Y, and Z coordinate values for a point *k* are stored in `vertex_list` array elements 3*k*, 3*k*+1 and 3*k*+2, respectively. An array that specifies *N* vertices must contain 3N 32-bit fixed-point elements, each of which is a coordinate value.

```
typedef long FIX;              /* 32-bit fixed-point type    */
                               /* x,y,z values for vertices */
FIX x[N], y[N], z[N];          /* 0, 1, 2, ... , k           */
FIX vertex_list[3*N];          /* vertex list                */
        .
        .
        .
vertex_list[0] = x[0];         /* x,y,z coords for point 0 */
vertex_list[1] = y[0];
vertex_list[2] = z[0];
vertex_list[3] = x[1];         /* x,y,z coords for point 1 */
vertex_list[4] = y[1];
vertex_list[5] = z[1];
vertex_list[6] = x[2];         /* x,y,z coords for point 2 */
vertex_list[7] = y[2];
vertex_list[8] = z[2];
        .
        .
        .
vertex_list[3k] = x[k];        /* x,y,z coords for point k */
vertex_list[3k+1] = y[k];
vertex_list[3k+2] = z[k];
```

Figure 4-10. Vertex List Format

4.15.3 Point List

The point list is an array of values representing a collection of points in two-dimensional space. Each point is specified in terms of its X and Y coordinates. Each coordinate is represented as a 16-bit integer (C type short). Figure 4-11 illustrates the point list format. The X and Y coordinate values for a point *k* are stored in `point_list` array elements 2*k* and 2*k*+1, respectively. An array specifying *N* points must contain 2*N* 16-bit integer elements, each of which is a coordinate value.

```
/* x,y values for points 0,1,2,...,k */
short x[N], y[N];
short point_list[2*N];  /* point list */
      .
      .
      .
point_list[0] = x[0];    /* x,y coord's for point 0 */
point_list[1] = y[0];
point_list[2] = x[1];    /* x,y coord's for point 1 */
point_list[3] = y[1];
point_list[4] = x[2];    /* x,y coord's for point 2 */
point_list[5] = y[2];
      .
      .
      .
point_list[2k] = x[k];   /* x,y coord's for point k */
point_list[2k+1] = y[k];
```

Figure 4-11. Point List Format

4.15.4 Line List

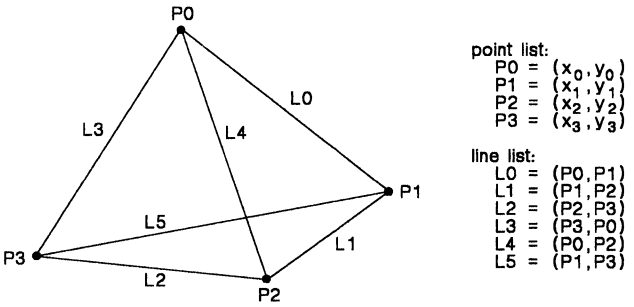
The line list is an array of values that represent a collection of straight lines. Each element of the line list array is an index into a point list (described in Section 4.15.3). The line list contains the topology information for a graphics object, and specifies which pairs of points are connected by the lines (if wireframe) or edges (if solid) of the object. Each line in the line list is represented by two adjacent elements. The first element is an index specifying the starting point for the line, and the second element is an index specifying the ending point. Each index is a 16-bit integer (C type `short`).

Figure 4-12 illustrates the relationship between the line list and point list formats. The example contains a wireframe figure that is made up of four lines. The indices for the starting and ending points of a line k are stored in line_list array elements $2k$ and $2k+1$, respectively. An array that specifies n lines must contain $2n$ 16-bit integer elements, each of which is an index into a point list. The figure shown in Figure 4-12(a) can be drawn using the following function call:

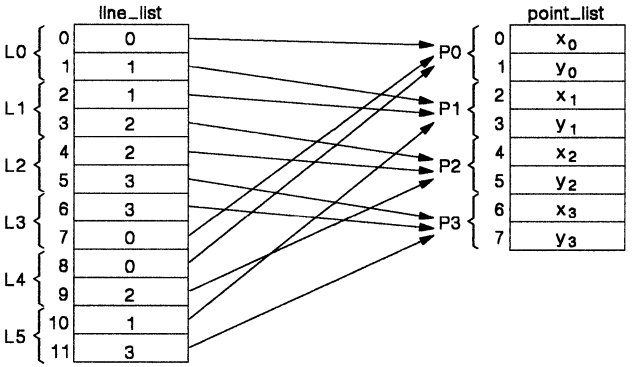
```
draw_polyline(6, line_list, point_list)
```

Figure 4-12 (b) shows the mapping of the data for the figure in Figure 4-12(a) to the `line_list` and `point_list` arrays. For example, line L0 is defined in the line list as having end points P0 and P1. Points P0 and P1 are defined in the point list as having coordinates (x0,y0) and (x1,y1).

If `line_list` array elements $2k$ and $2k+1$ contain index values n and m, respectively, the line is drawn from point n to point m of the point list array. Point list elements $2n$ and $2n+1$ contain the two coordinates of point n, (xn,yn). Similarly, the two coordinates of point m, (xm,ym), are stored in point list elements $2m$ and $2m+1$.



(a) A Wireframe Figure



(b) Mapping Data from the Wireframe into the line_list and pt_list Arrays

Figure 4-12. Line List Format

4.16 Mapping Pixels to XY Coordinates

Figure 4-13 illustrates the conventions that are used to map XY coordinates to pixels on the screen. The filled area is a rectangle of width w=5 and height h=3 whose top left corner is located at XY coordinates (4,2). The fill is performed by the following function call:

```
fill_rect(5, 3, 4, 2)
```

Pixels lying within the perimeter of the specified rectangle are "turned on" to represent the fill area. By convention, X increases from left to right, and Y increases from top to bottom. The default origin is at the upper left corner of the screen. (The origin may be relocated at an arbitrary position on or off screen by means of a call to the set_origin function.) The XY coordinates passed to graphics routines are constrained to be integer values. The coordinate grid is overlayed on the screen such that integer XY coordinate pairs coincide with pixel corners (not with pixel centers). The conventions used for determining which pixels are selected to represent filled areas and infinitely thin vectors are explained in the following discussion.

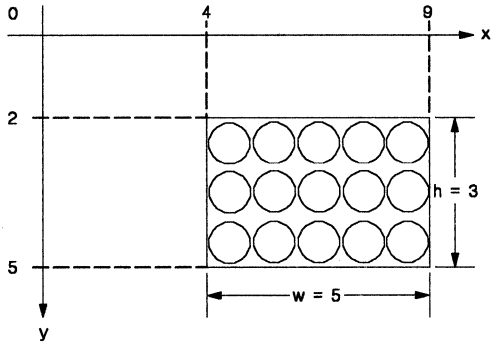


Figure 4-13. Rectangle Fill

4.16.1 Area Filling Conventions

Figure 4-14 shows a complex filled area. In this case, a fill_polygon command defines the fill area indicated by the straight edges in the figure. The rule for determining whether a pixel is selected as part of the fill area is as follows. If the center of the pixel falls within the mathematical boundary of the area, it is "turned on" to indicate that it is part of the fill area. (If a pixel's center falls precisely on the boundary between two areas, by convention the pixel is considered to be part of the area immediately below and to the right of the pixel). Pixels whose centers lie outside the boundary are not considered part of the fill region. The same principles are applied to the filling of other shapes (ellipses and thick lines drawn with a rectangular drawing pen, for example).

Graphics functions that follow the above conventions for filled areas include all functions whose names include the modifiers *fill*, *pen*, or *frame*.

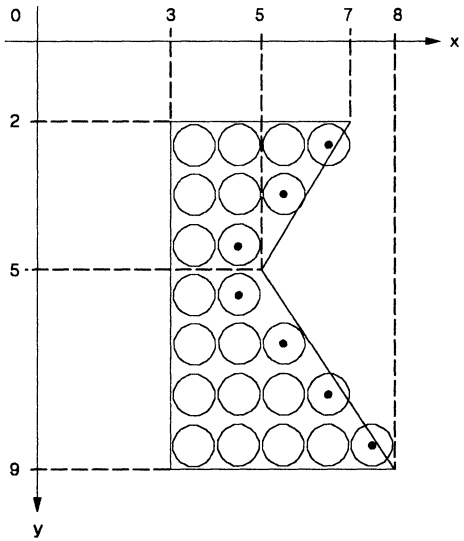


Figure 4-14. Polygon Fill

4.16.2 Vector Drawing Conventions

Points, lines, and arcs are defined mathematically to be infinitely thin. Since these figures contain no area, they are invisible if drawn using the conventions described above for filled areas. A different set of conventions must be used to make points, lines, and arcs visible. These are referred to as vector drawing conventions (to distinguish them from the area filling conventions discussed previously). Vector drawing conventions apply to all functions whose names include the modifier *draw*.

The vector drawing conventions associate the point identified by the integer coordinate pair (X,Y) with the pixel located to its lower right; that is, the pixel whose center is located at coordinates (X+1/2,Y+1/2). For example, the draw_point(7,10) command turns on the pixel at (7.5,10.5). As a second example, the polygon from Figure 4-14 is shown again in Figure 4-15, but is outlined rather than filled. (The draw_polyline function is used.) The points selected to represent the right side of the polygon are indicated as small black dots. The pixel to the lower right of each point is turned on to represent the edge of the polygon.

A line or arc drawn using the vector drawing conventions consists of a connected set of pixels. This means that the line or arc is drawn as a continuous set of pixels that connect (or touch) horizontally, vertically or diagonally, without gaps or holes in between.

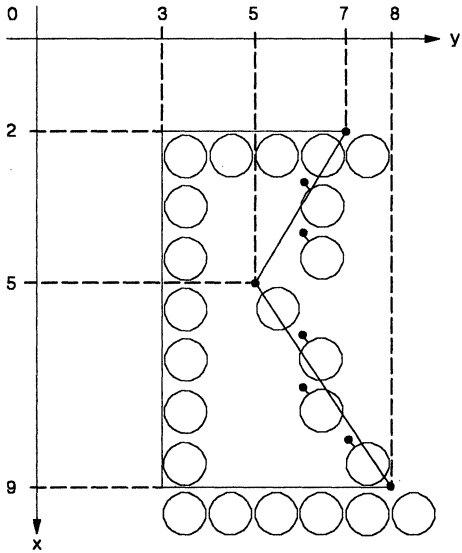


Figure 4-15. Polygon Outline

4.16.3 The Drawing Pen

The drawing commands that use vector drawing conventions can only draw lines and arcs that are a single pixel thick. To draw lines and curves of arbitrary thickness, a rectangular **pen** (or *brush* or *logical pel*) is used. Graphics functions that use the drawing pen have names containing the modifier *pen*.

You can use graphics commands to set the drawing pen's height and width to arbitrary positive, nonzero values. The pen is rectangular; its position is identified by its top left corner. For example, when a pen of width w and height h draws a point at (X,Y), the resulting rectangle's top left corner lies at (X,Y), and its bottom right corner lies at (X+w,Y+h). The rectangular area covered by the pen is filled with either a solid color or with the current pattern, depending on the function used.

The area under the drawing pen is filled according to the area filling conventions described previously. When the width and height of the drawing pen both equal 1, a line or arc drawn by the pen is similar in appearance to that drawn by a function following the vector drawing conventions. However, the pen functions conform to the area filling conventions, so a pen function can more faithfully track the perimeter of a filled area than a corresponding draw function. For example, consider an ellipse defined by some width w, height h, and coordinates (x,y). If a draw_oval(w,h,x,y) function call outlines a filled ellipse drawn by a fill_oval(w,h,x,y) function, the draw_oval function may not in all instances select the same perimeter pixels as the filled ellipse. This can leave gaps between the filled area and the outline. In contrast, a pen_oval(w,h,x,y) function call follows the filled ellipse precisely, remaining flush to the ellipse at all points along the perimeter.

4.17 System Implementation Issues

Most of the functions in the library are independent of system-dependent features such as pixel size and frame buffer dimensions. However, implementations of hardware functions such as the color palette, video timing, and bulk clearing of VRAMs necessarily differs from system to system. The function library provides several system-dependent functions to control such features.

4.17.1 Register Usage Conventions

Assembly language functions that are used in conjunction with the graphics functions should follow certain guidelines for register use. The following registers must be restored to their original states (the state before the function was called) before control is returned to the calling routine:

- Status register fields FE1 and FS1 must be restored. Fields FE0 and FS0 need not be restored.
- All A-file registers except A8 must be restored.
- In general, all B-file registers must be restored. However, certain B-file registers may be altered by attribute control functions that update parameters such as COLOR0 and COLOR1.
- In general, I/O registers CONTROL, DPYCTL, CONVSP, and CONVDP should be restored before returning to the calling routine. However, some I/O register bits may be altered by attribute control functions that update parameters such as the plane mask, pixel processing operation, or transparency flag. These register bits typically are not changed by graphics output functions.

Upon entry to a function, certain registers are in a known state and contain well-defined parameters. Assume that the following registers are in these states:

- **Status register.** The C environment always leaves the FE1 and FS1 fields defined as follows:
  - FE1 = 0
  - FS1 = 32

  FE0 and FS0 are undefined.
- **B-file registers.** Seven of the B-file registers are in a known state when a function is entered:
  - SPTCH - pitch of selected font.
  - DPTCH - screen pitch (difference in starting memory addresses of any two successive scan lines in display memory).
  - OFFSET - memory address of pixel at top left of screen.
  - WSTART - top left corner of current visibility region.

- WEND - bottom right corner of current visibility region.
- COLOR0 - source background color for PixBlts (text and pattern fills).
- COLOR1 - source foreground color for PixBlts, fills and vectors.

- **I/O registers.** Certain I/O registers contain defined parameters at entry to a function:
  - CONTROL - contains current pixel processing operation code and transparency control bit. These are set by the application program and may vary from one call to the next. In contrast, the window mode, PBH and PBV bits are set to specific values. The window mode is set to enable clipping without interrupts (W=3). The PBH and PBV bits are both zero.
  - CONVSP - is set up for the pitch of the selected font.
  - CONVDP - is set up for the screen pitch.
  - PSIZE - the number of bits per pixel on the screen.
  - PMASK - contains the current plane mask.

4.17.2 Functions with System Dependencies

The current implementation of system-dependent library functions supports the TMS34010 Software Development Board. System-dependent aspects of the SDB are chiefly due to the special capabilities of the VRAM and color-palette device that are used on the SDB. The TMS4161 or TMS4461 video RAMs used on the SDB are capable of bulk clearing the frame buffer, but video RAMs from other manufacturers may not support the register-to-memory cycles necessary to implement this feature. (The bulk clear capability is described in the *TMS34010 User's Guide*.) Also, the TMS34070 color palette provides color indexing for displays using four bits per pixel. The TMS34070 is capable of loading a new lookup table prior to each scan line of the display, and this permits the color palette to be changed on a line-by-line basis. Comparable devices from other manufacturers may not provide this capability.

Table 4-15 is a list of library routines that perform system dependent functions. If a routine is not listed, assume that it is not system dependent.

Table 4-15. Functions with System Dependencies

Function	System Dependency
clear—screen	Uses the TMS34070 color palette; uses the bulk clear capability of the TMS4161 video RAM.
color—blend	Uses the TMS34070 color palette.
getall—palet	Uses the TMS34070 color palette.
init—palet	Uses the TMS34070 color palette.
init—screen	Uses the TMS34070 color palette; uses the bulk clear capability of the TMS4161 video RAM.
init—video	Initializes video timing and screen refresh registers.
new—screen	Uses the TMS34070 color palette; uses the bulk clear capability of the TMS4161 video RAM.
set—palet	Uses the TMS34070 color palette.
setall—palet	Uses the TMS34070 color palette.

4.17.3 Uninitialized System Parameters

The function library assumes that certain system parameters are under control of an operating system or control program, and avoids initializing or modifying these parameters. Specifically, library functions do not alter the following hardware registers:

- The master interrupt enable bit (IE) in the status register
- The INTENB (interrupt enable) register
- The cache disable bit (CD) in the CONTROL register
- The DRAM-refresh control bits (RR and RM) in the CONTROL register
- The four host interface registers (HSTADRL, HSTADRH, HSTDATA, and HSTCTL)

4.17.4 Interrupts

The assembly language routines within the library use the TMS34010's A14 register as a general-purpose register. Interrupt service routines should make no assumptions regarding the state of A14 at the time an interrupt occurs. In particular, they should not assume that A14 points to the top of the C parameter stack.

The library does not use interrupts. A number of graphics functions in the library make use of the window violation detection capabilities of the TMS34010, but they assume that the WV interrupt is disabled.

Similarly, the library's wait—scan and delay functions poll the display interrupt request, but assume that the display interrupt is disabled. An operating environment or application program that includes a display interrupt service routine may have difficulty using these two functions as currently implemented.

Section 5

Alphabetical Reference of Functions

This section contains a reference of the math/graphics function library. The discussions are organized into alphabetical order; each discussion begins on a new page so you easily can find each function. Each discussion:

- Shows the syntax of the function declaration and arguments that the function uses.
- Contains a description of the function operation, which explains any input arguments and return values.
- Provides an example that uses the function.

Note:

All of the functions can be called from a C program *except* for these functions:

FIX2FL
FL2FIX
FL_ADD
FL_COS
FL_MULT
FL_SIN

Syntax

```
double acos(x)
   double x;
```

Description

The acos function calculates the inverse cosine of a double-precision floating-point number. Both the argument x and the return value are double-precision floating-point numbers.

The return value is an angle expressed in radians:

- If x is in the range [-1,1], the return value is in the range [0,π].
- If x is outside the range [-1,1], fp—error is called with error code = 18 (see the description of the floating-point facility in *TMS34010 C Compiler User's Guide*).
- If x > 1, a value of +∞ is returned.
- If x < -1, a value of -∞ is returned.

Example

```
/*****************************************/
/* acos returns value expressed in radians */
/*****************************************/
  extern double acos();
  double realval, radians;

  realval = 1.0;
  radians = acos(realval);
  return (radians);     /* acos returns π/2 */
```

Syntax

```
void add_text_space(n)
   int n;     /* Add n to default spacing */
              /* between characters       */
```

Description

The add_text_space function changes the horizontal spacing between characters by an amount `n`. Associated with each text font is a default spacing between a character and the character to its right. When a string of characters is drawn to the screen, `n` is added to the default spacing between characters defined in the data structure for the current font. Argument `n` is specified in multiples of the pixel width; it can be positive or negative, depending on whether you wish to increase or decrease the spacing.

Once text spacing is modified by the add_text_space function, the spacing increment remains in effect (within the viewport) until this function is called again. The init_text function sets the spacing increment to its default value, 0.

> **Note:**
>
> Before you call the add_text_space function, call the init_text function to initialize the text data structures.

Example

```
short x, y, i;
char *s;

init_video(1);
init_grafix();
init_screen();
init_text();
s = "Note increasing space between characters.";
for (i = -8, y = 0; i < 20; ++i) {
    add_text_space(i);
    x = 320 - get_width(s)/2;
    y += char_high();
    draw_string(x, y, s);
}
```

Syntax

```
double asin(x)
   double x;
```

Description

The asin function calculates the inverse sine of a double-precision floating-point number. Both the argument x and the return value are double-precision floating-point numbers.

The return value is an angle expressed in radians.

- If x is in the range [-1,1], the return value is in the range $[-\pi/2,+\pi/2]$.
- If x is outside the range [-1,1], fp_error is called with error code = 18 (see the description of the floating-point facility in *TMS34010 C Compiler User's Guide*).
- If $x > 1$, a value of $+\infty$ is returned.
- If $x < -1$, a value of $-\infty$ is returned.

Example

```
/************************************************/
/*   asin returns value expressed in radians    */
/************************************************/
   extern double asin();
   double realval, radians;

   realval = 1.0;
   radians = asin(realval);  /* asin returns π/2 */
```

Syntax

```
double atan(x)
   double x;
```

Description

The atan function calculates the inverse tangent of a real number. Both argument x and the return value are double-precision floating-point values.

Given an input argument x, *atan(x)* returns a number *y* such that *tan(y)* = *x*. The return value is an angle expressed in radians, and is restricted to the range [-π/2,+π/2].

Example

```
/*************************************************/
/*    atan returns a value expressed in radians  */
/*************************************************/
   extern double atan();
   double realval, radians;

   realval = 0.0;
   radians = atan(realval);      /* return value = 0 */
```

Syntax

```
double atan2(u,v)
   double u,v;
```

Description

The atan2 function calculates the inverse tangent of the quotient of real number u divided by real number v. The two arguments u and v and the return value are all double-precision floating-point values.

The return value is an angle expressed in radians, and is in the range $[-\pi,+\pi]$.

- Given input arguments u and v, *atan(u,v)* returns a number *y* such that $tan(y) = u/v$.
- When both arguments are 0, the return value is $+\infty$, and fp_error is called with error code = 23 (see the description of the floating-point facility in *TMS34010 C Compiler User's Guide*).

Example

```
extern double atan2();
double rvalu, rvalv;
double radians;

rvalu   = 0.0;
rvalv   = 1.0;
radians = atan2(rvalr, rvalu); /* return value = 0 */
```

Syntax

```
void bit_expand(srcbits, srcpitch, w, h, xleft, ytop)
   short srcbits[];      /* source bit map          */
   long  srcpitch;       /* source pitch            */
   int   w, h;           /* dest width and height   */
   int   xleft, ytop;    /* dest left side and top */
```

Description

The bit—expand function expands a bit map onto the screen by replacing each bit in the source with one of two pixel values. 0s in the bit map are expanded to pixel value COLOR0, and 1s are expanded to COLOR1. (See references to set—color0 and set—color1 functions.)

- The source bitmap is specified in terms of its base address and pitch:
  - `srcbits` specifies the base address.
  - `srcpitch` specifies the memory pitch of the bit map.
- The last four arguments specify the rectangular area of the screen that is modified:
  - The width `w`,
  - The height `h`, **and**
  - The coordinates of the top left corner (`xleft,ytop`).

  `w` and `h` must be nonnegative.

The source pitch is the difference in starting addresses of two adjacent rows in the source bitmap. (TMS34010 addresses are bit addresses. The pitch is the number of bits in memory between the start of one row of the bit map and the next.) The pitch can be any number greater than or equal to the number of pixels per row of the destination array (the `w` argument). The source array `srcbits` containing the bit map must be large enough to contain one bit for every pixel in the destination array.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
/*********************************/
/*   Expand bitmap onto screen   */
/*********************************/
typedef enum { FIELDWIDTH = 1 } BIT;
static BIT bitmap[] = {
     0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,
     0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,
     0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,
     0,0,1,1,0,0,0,1,1,1,1,1,0,0,0,0,
     0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,
     0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,
     0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,
     0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,
     0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,
     0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
     1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
     1,1,1,1,1,1,1,1,1,0,1,1,1,0,0,0,
     1,1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,
     1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,
     1,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,
     1,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,1,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
     0,1,1,1,1,1,0,1,1,1,1,1,0,0,0,0,
     1,1,1,1,1,0,0,0,1,1,1,1,1,0,0,0
};
short buf[640/4];          /* Assume 4 bits/pixel */
int   x=20, y=10, p=16, w=16, h=41;

init_video(1);
init_grafix();
init_screen();
bit_expand(bitmap, p, w, h, x, y);
zoom_rect(w, h, x, y, 10*w, 10*h, 100, 10, buf);
```

Syntax

```
void bound_fill(x, y, buffer, size, b_color)
   int  x, y;                /* starting point for */
                             /* seed fill          */
   char buffer[];            /* temporary buffer   */
   int  size;                /* size of buffer in  */
                             /* bytes              */
   unsigned long b_color;    /* boundary color     */
```

Description

The bound_fill function fills a bounded set of pixels. Starting at pixel coordinates (x,y), the function flood fills in all directions until the boundary color b_color is encountered. Pixels in the filled region are set to the current COLOR1.

Given a pixel size of *n* bits, the function uses only the *n* LSBs of b_color. The function ignores higher order bits.

A pixel is considered part of the bounded region if it is not equal to the boundary color, and has a horizontally or vertically adjacent neighbor pixel that is part of the region. (A diagonally adjacent neighbor is not sufficient.)

Argument buffer is an array that the function uses as as temporary working storage. The function destroys the original contents of the buffer. Argument size is the size of the buffer in bytes.

The bound_fill function differs from the seed_fill function, which fills a connected set of pixels the same color as the starting pixel.

The bound_fill function aborts (returns immediately) if any of these conditions are detected:

- The pixel at starting coordinates (x,y) is equal to the boundary color b_color.
- Starting coordinates (x,y) lie outside the current visibility rectangle (window).
- If at any point the buffer size is insufficient to continue.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long u, v, a, b, c, dc;
char buffer[100];

init_video(1);
init_grafix();
init_screen();
dc = 0x11111111;              /* Assume 4 bits/pixel */
u = v = 36 << 16;
for (c = -1; c != 0; c -= dc) {
     set_color1(c);
     a = u >> 16;
     b = v >> 16;
     draw_line(320+a, 240-b, 320+b, 240+a);
     draw_line(320+b, 240+a, 320-a, 240+b);
     draw_line(320-a, 240+b, 320-b, 240-a);
     draw_line(320-b, 240-a, 320+a, 240-b);
     u += u >> 3;
     v += v >> 3;
     u += v >> 3;
     v -= u >> 3;
}
set_color1(0x11111111);   /* fill color */
bound_fill(320, 240, buffer, 100, 7);
```

Syntax

```
void bound_patnfill(x, y, buffer, size, b_color)
   int  x, y;                /* starting point for */
                             /* seed fill          */
   char buffer[];            /* temporary buffer   */
   int  size;                /* size of buffer in  */
                             /* bytes              */
   unsigned long b_color;    /* boundary color     */
```

Description

The bound_patnfill function fills a bounded set of pixels. Starting at pixel coordinates (x,y), the function flood fills in all directions until the boundary color b_color is encountered. Pixels in the filled region are drawn with the current pattern, which is drawn in COLOR0 and COLOR1.

Given a pixel size of *n* bits, the function uses only the *n* LSBs of b_color. The function ignores higher order bits.

A pixel is considered part of the bounded region if it is not equal to the boundary color, and has a horizontally or vertically adjacent neighbor pixel that is part of the region. (A diagonally adjacent neighbor is not sufficient.)

Argument buffer is an array that the function uses as temporary working storage. The function destroys the original contents of the buffer. Argument size is the size of the buffer in bytes.

The bound_patnfill function differs from the seed_patnfill function, which fills a connected set of pixels the same color as the starting pixel.

The bound_patnfill function aborts (returns immediately) if any of these conditions are detected:

- The pixel at starting coordinates (x,y) is equal to the boundary color b_color.
- Starting coordinates (x,y) lie outside the current visibility rectangle (window).
- If at any point the buffer size is insufficient to continue.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long u, v, a, b, c, dc;
char buffer[100];

init_video(1);
init_grafix();
init_screen();
dc = 0x11111111;          /* Assume 4 bits/pixel */
u = v = 36 << 16;
for (c = -1; c != 0; c -= dc) {
     set_color1(c);
     a = u >> 16;
     b = v >> 16;
     draw_line(320+a, 240-b, 320+b, 240+a);
     draw_line(320+b, 240+a, 320-a, 240+b);
     draw_line(320-a, 240+b, 320-b, 240-a);
     draw_line(320-b, 240-a, 320+a, 240-b);
     u += u >> 3;
     v += v >> 3;
     u += v >> 3;
     v -= u >> 3;
}
select_patn(10);                /* fill pattern */
set_color0(0x11111111);         /* fill color0  */
set_color1(0x33333333);         /* fill color1  */
bound_patnfill(320, 240, buffer, 100, 7);
```

Syntax

```
double ceil(x)
    double x;
```

Description

The ceil function returns a double floating point number representing the smallest integer greater than or equal to the input argument `x`.

Example

```
extern double ceil();

double answer;

answer = ceil(3.1415,&exp);

/* after execution, answer will be 4.0 */
```

Syntax `int char_high()`

Description The char_high function returns the character height (in pixels) for the current font. The character height is defined as the vertical distance between two adjacent rows of text, as measured from the two baselines. The character height is calculated as the sum of three quantities:

- Ascent,
- Descent, **and**
- Leading.

> **Note:**
>
> Before you call the char_high function, call the init_text function to initialize the text data structures.

Example

```
extern int char_high();
static char *s[] = {
    "1st line",
    "2nd line",
    "3rd line"
 };
int i, x, y;

init_video(1);
init_grafix();
init_text();
x = 8;
y = char_high();
for (i = 0; i <= 2; ++i) {
    draw_string(x, y, s[i]);
    y += char_high();
}
```

Syntax

```
int char_wide_max()
```

Description

The char_wide_max function returns the width (in pixels) of the widest character in the current font. The returned value is the sum of the character image width and the space preceding the next character to the right. (The character image is the bit map containing the character pattern.)

> **Note:**
>
> Before you call the char_wide_max function, call the init_text function to initialize the text data structures.

Example

```
extern int char_high(), char_wide_max();
static char c, *s;
int i, w, h, x, y;

init_video(1);
init_grafix();
init_text();
x = w = char_wide_max();
y = h = char_high();
s = "TMS34010";
while ((c = *s++) != '\0') {
    draw_char(x, y, c);
    y += h;
    x += w;
}
```

Syntax

```
void clear_screen(pixval)
  long pixval;    /* pixel value to which */
                  /* screen is set        */
```

Description

The clear—screen function clears the screen to the specified pixel value. The entire display memory is affected. For example, clear—screen(0) clears the entire frame buffer to 0s, including the color palette areas along the left edge of the screen. Video RAM register-to-memory cycles are used to make this function execute rapidly.

The pixel value must be replicated to fill the entire 32 bits of `pixval`. For example, if the pixel size is 4 bits, and the pixel value is 5, `pixval` is specified as 0x55555555.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short mypalet[16] = {
  0x0000, 0x00F0, 0x0F00, 0x0FF0, 0xF000, 0xF0F0,
  0xFF00, 0xFFF0, 0x0000, 0x0090, 0x0900, 0x0990,
  0x9000, 0x9090, 0x9900, 0x9990
};

init_video(1);
init_grafix();
/* Assume 4 bits per pixel */
clear_screen(0x55555555);
/* Restore palette */
setall_palet(mypalet,0xFFFF,480,0);
/* Draw border */
frame_rect(640, 480, 0, 0, 25, 20);
```

Syntax

```
int close_vuport(index)
  int index;  /* Identifies viewport to be closed */
```

Description

The close_vuport function closes a viewport that was previously opened and deletes all reference to the viewport structure from the graphics environment. The viewport is specified by argument `index`, which is the index value returned when the viewport was opened.

If the active viewport is designated by the argument, viewport 0 automatically becomes the active viewport when the previous viewport is closed. Viewport 0 cannot be closed; only viewports in the range 1 to *n*-1 can be closed, where *n* is the value returned by the get_vuport_max function. When the function is called with a valid index, the value 0 is returned to confirm that the viewport was closed as requested. When the function is called with an invalid index, a value of -1 is returned to indicate that no action was taken.

> **Note:**
>
> Before you call the close_vuport function, call the init_vuport function to initialize the viewport data structures.

Example

```
int index;
    .
    .
    .
index = open_vuport();
    .
    .
    .
close_vuport(index);
```

Syntax

```
void color_blend(pxlval, y1, y2, red1, grn1, blu1,
                 red2, grn2, blu2)
   int pxlval;  /* pixel value affected  */
   int y1;      /* starting scan line    */
   int y2;      /* ending scan line      */
   int red1;    /* red intensity at y1   */
   int grn1;    /* green intensity at y1 */
   int blu1;    /* blue intensity at y1  */
   int red2;    /* red intensity at y2   */
   int grn2;    /* green intensity at y2 */
   int blu2;    /* blue intensity at y2  */
```

Description

The color—blend function creates gradual changes in shading, highlights, and color blending effects by gradually varying the red, green, and blue intensities of the color associated with a specified pixel value on a line-by-line basis. Gradual vertical shading over takes place over a group of contiguous scan lines. The starting scan line is designated by `y1`, and the ending scan line is designated by `y2`.

Argument `pxlval` is the pixel value whose color is affected over the specified scan lines. It is also the number of the color palette register loaded with the specified red, green, and blue intensities. The range of `pxlval` is 0 to 15.

The y coordinates `y1` and `y2` are relative to the origin of the active viewport. Note that it is not necessary for `y1` to be less than `y2`, and vice versa. Changes to the palette are automatically restricted to the y limits of the visibility rectangle (intersection of screen with active viewport and clipping rectangle); scan lines corresponding to y values outside this range are unaffected. Intensities `red1`, `grn1`, `blu1`, `red2`, `grn2`, and `blu2` are 8-bit values in the range 0 to 255. Linear interpolation is used to adjust the 4-bit red, green, and blue values output by the TMS34070 DACs to approximate the 8-bit resolutions specified for the intensities.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
/************************************************/
/* Draw solid_filled rectangle that gradually   */
/* changes colors from red at the top to blue-  */
/* gray at the bottom.                          */
/************************************************/
set_color1(0x33333333);   /* 4 bits per pixel */
fill_rect(200, 100, 125, 65);
color_blend(3, 65, 165, 255, 0, 0,  70, 70, 150);
```

Syntax

```
typedef long FIX
void copy_matrix(matrixin, matrixout)
   FIX  matrixin[16];
   FIX  matrixout[16];
```

Description

The copy—matrix function copies a 4×4 input matrix to a 4×4 output matrix. Both the input matrix `matrixin` and output matrix `matrixout` are stored in 16-element arrays of type FIX.

Example

```
typedef long FIX;
{
 FIX matrixin[16];
 FIX matrixout[16];
      .
      .
      .
 copy_matrix(matrixin, matrixout);
}
```

Syntax

```
void copy_vertex(n, vertexin, vertexout)
   typedef long FIX;  /* fixed-point format          */
   int n;             /* number of vertices in list */
   FIX vertexin[];    /* input vertex list          */
   FIX vertexout[];   /* output vertex list         */
```

Description

The copy—vertex function copies an input vertex list to an output vertex list. Input argument `n` is the number of vertices that are copied from the input vertex list to the output vertex list. Both the input vertex list `vertexin` and output vertex list `vertexout` are arrays of 32-bit fixed-point values. A vertex is stored as three consecutive 32-bit coordinate values, X, Y, and Z. Each array contains 3n 32-bit elements. See Figure 4-10 (page 4-30) for the vertex list format.

Example

```
typedef long FIX;
/*****************************************************/
/* Copy 5 vertices from vertexin[] to vertexout[]. */
/* Each vertex consumes 3 storage elements in an   */
/* array, and the minimum array size is 3*5 = 15.  */
/*****************************************************/
FIX vertexin[3*5], vertexout[3*5];
      .
      .
      .
copy_vertex(5, vertexin, vertexout);
```

Syntax

```
int copy_vuport(index1, index2)
   int index1, index2;
```

Description

The copy—vuport function copies all attributes of the source viewport to the destination viewport. The source viewport is designated by argument `index1`, which is the value returned by the open—vuport function when the viewport was opened. The destination viewport is designated by argument `index2`. Both the source and the destination viewports must be opened before calling the copy—vuport function.

The destination viewport automatically becomes the active viewport. If either viewport was not previously opened, a value of -1 is returned to indicate that an error was detected and that no viewport was copied. Otherwise, a value of 0 is returned to indicate that the viewport was successfully copied. See the description of the open—vuport function for a list of viewport attributes copied by the function.

> **Note:**
>
> Before you call the copy—vuport function, call the init—vuport function to initialize the viewport data structures.

Example

```
/*************************************************/
/* Create 2 new viewports identical to the first, */
/* but located in different areas of the screen.  */
/*************************************************/
int index[4];                   /* viewport indices */

init_video(1);
init_grafix();
init_vuport();
init_screen();
/***   Open viewport 1    ***/
index[1] = open_vuport();
size_vuport(150,300);
move_vuport(10,50);
      .
      .
      .
/***   Make two new viewports similar to first    ***/
index[1] = open_vuport();       /* create viewport 2 */
index[2] = open_vuport();       /* create viewport 3 */
copy_vuport(index[1],index[2]);
copy_vuport(index[1],index[3]);
/***   Move viewport 3 to right of viewport 1    ***/
move_vuport(340,50);
/***   Move viewport 2 between other two viewports    ***/
select_vuport(index[2]);
move_vuport(170,50);
```

Syntax

```
double cos(x)
   double x;
```

Description

The cos function calculates the cosine of real number x, where x is an that is angle expressed in radians. Both the argument and return value are double-precision floating-point values.

An argument x with a magnitude greater than or equal to 1.0E+8 causes cos(x) to return a value of 0, and fp_error is called with error code = 17 (see the *TMS34010 C Compiler User's Guide* for a description of the fp_error function).

Example

```
extern double cos();
double radians, cval;    /* cval is returned by cos */

radians = 3.1415927;
cval = cos(radians);     /* return value = -1 */
return(cval);
```

Syntax

```
double cosh(x)
   double x;
```

Description

The cosh function returns the hyperbolic cosine of a real number `x`. Both the argument `x` and return value are double-precision floating-point values.

Example

```
extern double cosh();
double x, y;

x = 0.0;
y = cosh(x);        /* return value = 1.0 */
```

Syntax

```
double cotan(x)
   double x;
```

Description

The cotan function calculates the cotangent of a real number x, where x is an angle that is expressed in radians. The sign of the result is the same as the sign of the argument. Argument x and the return value are both double-precision floating-point values.

If the absolute value of argument x is greater than or equal to 1.0E+8, then a value of 0 is returned, and fp_error is called with error code = 20. If the absolute value of x is less than or equal to 1.0E-300, then cotan(x) returns a value of $+\infty$ or $-\infty$, and fp_error is called with error code = 19 (see the *TMS34010 C Compiler User's Guide* for a description of the fp_error function).

Example

```
extern double cotan();
double radians, cotval;

radians = 3.1415927/2.0;      /* 90 degrees */

cotval = cotan(radians);      /* return value = 0  */
```

Syntax

```
int cpw(x, y)
   int x, y;     /* pixel coordinates */
```

Description

The cpw function generates 4-bit outcode based on a pixel's position relative to the current window. Arguments `x` and `y` are the coordinates of the pixel.

The window is the visibility rectangle defined as the intersection of:

- The screen,
- The viewport, **and**
- The clipping rectangle.

The outcode value is contained in the 4 LSBs of the return value. Outcode values include:

0000_2 if the point lies *within* the window.

$01xx_2$ if the point lies *above* the window.

$10xx_2$ if the point lies *below* the window.

$xx01_2$ if the point lies *left* of the window.

$xx10_2$ if the point lies *right* of the window.

Refer to the *TMS34010 User's Guide* for a detailed description of the outcodes.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
/*********************************************/
/*       Bounce dot off walls of window      */
/*********************************************/
#define XMIN    100     /* Define window limits */
#define YMIN    100
#define XMAX    300
#define YMAX    300
extern int cpw();
int i, outcode, x=XMIN, y=YMIN, dx=5, dy=3;

init_video(1);
init_grafix();
init_vuport();
init_screen();
set_cliprect(XMAX-XMIN, YMAX-YMIN, XMIN, YMIN);
for (i = 1; i < 200; ++i) {
    if ((outcode = cpw(x += dx, y += dy)) != 0) {
        if (outcode & 1) {          /* Bounce off  */
            x += 2*(XMIN-x);        /* left wall   */
            dx = -dx;
        } else if (outcode & 2) {   /* Bounce off  */
            x -= 2*(x-XMAX);        /* right wall  */
            dx = -dx;
        }
        if (outcode & 4) {          /* Bounce off  */
            y += 2*(YMIN-y);        /* top wall    */
            dy = -dy;
        } else if (outcode & 8) {   /* Bounce off  */
            y -= 2*(y-YMAX);        /* bottom wall */
            dy = -dy;
        }
    }
    draw_point(x, y);
}
```

Syntax

```
void delay(n)
   int n;       /* number of ticks */
```

Description

The delay function waits for a number of ticks, n, to elapse before returning control to the calling program. One tick equals 1/60th of a second. The function is synchronized to the frame rate, and a tick is registered at the end of each frame.

Given a positive argument n, the function counts *n+1* end-of-frames before returning control to the calling program. Control is returned just after the bottom line of the display is output. If n = 0, the function delays only until the end of the current frame is encountered.

If the display interrupt is enabled, the function aborts immediately upon being called. If argument n is negative, the function aborts immediately. No error code is generated in either case.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
/*********************************/
/*     Draw ticking second hand    */
/*********************************/
typedef long FIX;
static FIX rotation[3] = {0, 0, 0};
static long xyz[] = {0,-200,0, 30,0,0, 0,30,0, -30,0,0};
static short connect[8] = {0,1, 1,2, 2,3, 3,0};
FIX   matrix[16];
FIX   verts[12];
short xy[8];
int   angle;

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
for (;;)
    for (angle = 0; angle < 360; angle += 6) {
        init_matrix(matrix);
        rotation[0] = angle << 16;
        rotate(matrix, rotation);
        long_to_fix(12, xyz, verts);
        transform(matrix, 4, verts);
        vertex_to_point(4, verts, xy);
        delay(60);                    /* Wait 1 second */
        init_screen();
        draw_oval(420, 420, -210, -210);
        draw_polyline(4, connect, xy);
    }
```

Syntax

```
int draw_char(x, y, c)
   int  x, y;          /* starting coordinates */
   char c;             /* ASCII character code */
```

Description

The draw—char function draw a single bit-mapped character. The character is drawn in the current font.

- Arguments x and y specify the position of the character:
  - Coordinate x is the horizontal position at the left edge of the character.
  - Coordinate y is the vertical position at the baseline of the string (*not* at the top of the string).
- Argument c is a pointer to a character.

The return value is the x coordinate of the next character position to the right of the specified character. The x value is expressed in viewport-relative coordinates. If the character lies entirely above or below the window, the unmodified starting x coordinate is returned.

> **Note:**
>
> Before you call the draw—char function, call the init—text function to initialize the text data structures and call the init—grafix function to initialize the graphics environment.

Example

```
int  x, y;
char c;

init_video(1);
init_grafix();
init_text();              /* Install default font */
init_screen();
x = 0;
y = -170 << 16;
/** Draw the letters 'A' through 'Z' **/
for (c = 'A'; c < 'Z'; ++c) {
    draw_char((x>>16)+304, (y>>16)+244, c);
    x += y >> 3;
    y -= x >> 3;
    draw_char((x>>16)+304, (y>>16)+244, c -'A'+'a');
    x += y >> 3;
    y -= x >> 3;
}
```

Syntax

```
void draw_line(x1, y1, x2, y2)
   int x1, y1;       /* Start coordinates */
   int x2, y2;       /* End coordinates   */
```

Description

The draw_line function uses Bresenham's algorithm to draw a line from the starting point to the ending point.

- `x1` and `y1` specify the starting coordinates
- `x2` and `y2` specify the ending coordinates.

The line is one pixel in thickness and is drawn in the current COLOR1.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int i, x1 = -40, y1 = 80, x2 = 80, y2 = 280;

init_video(1);
init_grafix();
init_screen();
/***  Draw 202 lines in different orientations  ***/
for (i = 202; i > 0; --i) {
    draw_line(x1+305, y1+222, x2+305, y2+222);
    x1 += y1 >> 5;
    y1 -= x1 >> 5;
    x2 += y2 >> 5;
    y2 -= x2 >> 5;
}
```

Syntax

```
void draw_oval(w, h, xleft, ytop)
   int w, h;          /* width and height of */
                      /* enclosing rectangle */
   int xleft, ytop;   /* XY coordinates at   */
                      /* top left corner     */
```

Description

The draw_oval function draws the outline of an ellipse given the minimum enclosing rectangle in which the ellipse is inscribed. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes. The enclosing rectangle is defined by four arguments:

- The width `w`
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The outline is one pixel thick, and is drawn in the current COLOR1.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y;

init_video(1);
init_grafix();
init_screen();
/***  Draw ellipses of various sizes  ***/
for (w = 0, x = 4; w < 33; ++w, x += w + 3)
     for (h = 0, y = 4; h < 28; ++h, y += h + 3)
         draw_oval(w, h, x, y);
```

Syntax

```
void draw_ovalarc(w, h, xleft, ytop, theta, arc)
   int w, h,          /* width and height          */
   int xleft, ytop;   /* top left corner           */
   int theta;         /* starting angle (degrees) */
   int arc;           /* angle extent (degrees)    */
```

Description

The draw_ovalarc function draws an arc taken from an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the co-ordinate axes. The arc is one pixel in thickness, and is drawn in the current COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.
- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
#define XC     320 /* Screen center coordinates      */
#define YC     240
#define WMAX   636 /* Limits of enclosing rectangle  */
#define HMAX   476
#define DX      16 /* Increment rectangle dimensions */
#define DY      12
int w, h;

/***  Draw spiral using draw_ovalarc function   ***/
init_video(1);
init_grafix();
init_screen();
for (w = WMAX, h = HMAX; w > DX; h -= DY) {
    draw_ovalarc(w, h, XC-w/2, YC-h/2, 0, 270);
    w -= DX;
    draw_ovalarc(w, h, XC-w/2, YC-h/2, 270, 90);
}
```

Syntax

```
void draw_piearc(w, h, xleft, ytop, theta, arc)
   int  w, h,          /* width and height          */
   int  xleft, ytop;   /* top left corner           */
   int  theta;         /* starting angle (degrees) */
   int  xend, yend     /* XY coordinates for end pt*/
```

Description

The draw—piearc function draws an arc taken from an ellipse. Two straight lines connect the two end points of the arc with the center of the ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes. The arc and two lines are all one pixel in thickness, and are drawn in the current COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.

- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, t, dt, dx;

init_video(1);
init_grafix();
t = dx = dt = 8;
w = h = 80;
/** Draw animated pieman **/
for (x = -w, y = 240-w/2; x < 650; x += dx) {
     if ((t += dt) > 80 || t <= 0)
         dt = -dt;
     delay(0);
     init_screen();
     draw_piearc(w, h, x, y, t/2, 360-t);
     draw_piearc(w, h, x+w/2, y, -15, 30);
}
```

Syntax

```
void draw_point(x, y)
   int  x, y;        /* pixel coordinates */
```

Description

The draw_point function draws a single pixel. Arguments x and y give the XY coordinates of the designated pixel. The pixel is drawn in the current COLOR1.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int i, x, y, xy, yx;

init_video(1);
init_grafix();
init_screen();
x = xy = 0;
y = yx = 200;
/** Draw lissajous pattern in dots **/
for (i = 1200; i > 0; --i) {
     draw_point(x+320, y+240);
     x += yx >> 4;
     yx -= x >> 4;
     y += xy >> 5;
     xy -= y >> 5;
}
```

Syntax

```
void draw_polyline(n, linelist, ptlist)
   int   n;                  /* number of lines */
   short linelist[];         /* list of lines   */
   short ptlist[];           /* list of points  */
```

Description

The draw—polyline function draws multiple lines.

- `n` specifies the number of lines that are drawn.
- `linelist` is an array of type short; it specifies the list of lines that are drawn. Each element in the `linelist` array is an index into the `ptlist` array.
- The third argument, the `ptlist` array, contains the XY coordinates of the starting and ending points for each line.

Each pair of adjacent 16-bit elements in the `ptlist` array is an X coordinate followed by a Y coordinate. Each pair of adjacent 16-bit elements in the `linelist` array is a pair of indices into the `ptlist` array, and designates the start and end points of a line.

For example, the first line drawn is specified in the first two elements, `linelist[0]` and `linelist[1]`. Assume that these contain index values 4 and 7, respectively. The starting coordinates for the line are contained in `ptlist[2*4]` and `ptlist[2*4+1]`. The ending coordinates are contained in `ptlist[2*7]` and `ptlist[2*7+1]`.

The individual elements of the `linelist` array are assigned as follows:

```
linelist[0]    = starting point of line 0
linelist[1]    = ending point of line 0
linelist[2]    = starting point of line 1
linelist[3]    = ending point of line 1
    .
    .
    .
linelist[2n]   = starting point of line n-1
linelist[2n+1] = ending point of line n-1
```

The individual elements of the `ptlist` array are assigned as follows:

```
ptlist[0]      = x coordinate value for point 0
ptlist[1]      = y coordinate value for point 0
ptlist[2]      = x coordinate value for point 1
ptlist[3]      = y coordinate value for point 1
    .
    .
    .
ptlist[2m]     = x coordinate value for point m-1
ptlist[2m+1]   = y coordinate value for point m-1
```

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short xy[] = {
    380,200, 480,200, 480,300, 380,300,
    340,270, 340,170, 440,170,
    230,180, 280,300, 160,300, 146,263
};
static short cube[] = {
    0,1, 1,2, 2,3, 3,4, 4,5, 5,6, 6,1, 3,0, 5,0,
};
static short pyramid[] = {
    7,8, 8,9, 9,10, 10,7, 7,9
};

/***  Draw a cube and a pyramid sitting side by side  ***/
init_video(1);
init_grafix();
init_screen();
draw_polyline(9, cube, xy);
set_color1(0x11111111);        /* Assume 4 bits/pixel */
draw_polyline(5, pyramid, xy);
```

Syntax

```
void draw_rect(w, h, xleft, ytop)
   int w, h;          /* width and height    */
                      /* of rectangle        */
   int xleft, ytop;   /* coordinates at top  */
                      /* left corner         */
```

Description

The draw—rect function draws the outline of a rectangle. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The outline is one pixel in thickness, and is drawn in the current COLOR1.

The draw—rect function is equivalent to the following four calls to the draw—line function:

```
draw_line(xleft, ytop, xleft+w, ytop);
draw_line(xleft, ytop+h, xleft+w, ytop+h);
draw_line(xleft, ytop+1, xleft, ytop+h-2);
draw_line(xleft+w, ytop+1, xleft+w, ytop+h-2);
```

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
init_video(1);
init_grafix();
init_screen();
/*****************************/
/**  Draw one big rectangle  **/
/**    and four little ones   **/
/*****************************/
draw_rect(440, 280, 100, 100);
draw_rect(420, 30, 110, 110);
draw_rect(220, 220, 110, 150);
draw_rect(190, 150, 340, 150);
draw_rect(190, 60, 340, 310);
```

Syntax

```
int draw_string(x, y, s)
   int  x, y;     /* starting coordinates              */
   char *s;       /* ASCII string terminated by NULL */
```

Description

The draw_string function draws a string of characters to a position on the screen. The string is drawn in the current font.

- The first two arguments define the starting position for the string:
  - Coordinate `x` is the horizontal position at the left edge of the string.
  - Coordinate `y` is the vertical position at the baseline of the string (*not* the top of the string).
- Argument `s` is a character string. The string is in standard C format: a sequence of 8-bit ASCII character codes terminated by a NULL character (ASCII code = 0).

The return value is the `x` coordinate of the next character position to the right of the string. The `x` value is expressed in viewport-relative coordinates. If the string lies entirely above or below the window, the unmodified starting `x` coordinate is returned.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int  i, x, y;
char *s;

s = "Hello world.";
init_video(1);
init_grafix();
init_text();
init_screen();
transp_on();
x = 0;
y = 200 << 16;
/** Write "Hello world" to the screen 50 times **/
for (i = 50; i > 0; --i) {
    draw_string((x>>16)+272, (y>>16)+244, s);
    x += y >> 3;
    y -= x >> 3;
}
```

Syntax

```
double exp(x)
   double x;
```

Description

The exp function calculates the exponential function of real number x. The value returned is natural number *e* raised to the power x. Both the argument and return value are double-precision floating-point values.

- If x > 500, a value of $+\infty$ is returned, and fp_error is called with error code = 21 (see the reference to the fp_error function in the *TMS34010 C Compiler User's Guide*).
- If x < -500, a value of 0 is returned, and fp_error is called with error code = 22.

Example

```
extern double exp();
double x, y;      /* y is returned by exp     */

 x = 2.0;
 y = exp(x);      /* y = 7.38, which is e**2.0 */
```

Syntax

```
double fabs(x)
   double x;
```

Description

The fabs function calculates the absolute value of a real number x. Both argument x and the return value are double-precision floating-point values.

Example

```
extern double fabs();
double x, y;

x = -57.5;
y = fabs(x);        /* return value = +57.5 */
```

Syntax

```
int fill_convex(n, edgelist, ptlist)
   int   n;              /* number of polygon vertices */
   short edgelist[];     /* list of edges              */
   short ptlist[];       /* list of vertices (points)  */
```

Description

The fill—convex function fills a convex polygon given a list of points representing the vertices. In order to be drawn correctly, the polygon must be convex; that is, it should contain no concavities. A polygon must have at least three vertices to be visible. An edge of the polygon is assumed between the first and last vertices specified. The polygon is solid-filled with the current COLOR1.

The function requires three input arguments:

- Argument `n` defines the number of vertices in the polygon.
- The second argument, `edgelist`, is an array of type short. The members of the array are indices that specify the order in which the vertices are traversed, moving in a clockwise direction around the edge of the polygon. (*Clockwise*, in this context, assumes X increasing from left to right and Y increasing from top to bottom.) Each element of the `edgelist` array is an index into the `ptlist` array.
- The third argument, `ptlist`, is an array of type short. Each pair of adjacent 16-bit elements contains the X and Y coordinates, respectively, of a vertex.

For example, `edgelist[k]` contains the index for vertex k, where k is in the range 0 to `n-1`. The XY coordinates for vertex k are contained in `ptlist[2*n]` and `ptlist[2*n+1]`.

The fill—convex function does automatic culling of back faces to support 3D applications. In other words, a polygon is drawn only if its front side is visible; that is, if it is facing toward the screen. If the vertices are specified in counterclockwise order, the polygon is assumed to be facing away from the screen and is therefore not drawn. In this case, a value of 0 is returned by the function. Otherwise, a value of 1 is returned to indicate that the polygon is visible.

The back face test is done by first comparing vertices `n-2`, `n-1` and 0 to determine whether the polygon vertices are specified in clockwise (front face) or counterclockwise (back face) order. This test relies on the polygon containing no concavities. If the three vertices are found to be colinear, the back face test is made again using the next three vertices, `n-1`, 0 and 1. The test repeats until three vertices are found that are not colinear. If all the vertices are colinear, the polygon is invisible and a value of 0 is returned.

This function is similar to the fill—polygon routine, but is specialized for rapid drawing of convex polygons. Note that the `edgelist` array format for the fill—convex function differs from the `linelist` array format for the fill—polygon function. While the fill—convex function is more specialized than the fill—polygon function, it also executes more rapidly and supports realtime applications such as animation.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long i, hue;
static short connect[] = { 0, 1, 2 };
static short xy[] = { 0,-170, 196,170, -196,170 };

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
init_screen();
/** Fill triangles in 15 different colors **/
for (i = 15, hue = 0; i > 0; --i) {
     set_color1(hue += 0x11111111);
     fill_convex(3, connect, xy);
     xy[0] += xy[1] >> 3;
     xy[1] -= xy[0] >> 3;
     xy[2] += xy[3] >> 3;
     xy[3] -= xy[2] >> 3;
     xy[4] += xy[5] >> 3;
     xy[5] -= xy[4] >> 3;
}
```

Syntax

```
void fill_oval(w, h, xleft, ytop)
   int  w, h;              /* width and height of */
                           /* enclosing rect      */
   int  xleft, ytop;       /* XY coordinates of   */
                           /* top left corner     */
```

Description

The fill_oval function draws an ellipse that is solid-filled with the current COLOR1. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes.

The ellipse is defined by the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y;

init_video(1);
init_grafix();
init_screen();
/***   Fill ellipses of various sizes   ***/
for (w = 0, x = 4; w < 33; ++w, x += w + 3)
     for (h = 0, y = 4; h < 28; ++h, y += h + 3)
          fill_oval(w, h, x, y);
```

Syntax

```
void fill_piearc(w, h, xleft, ytop, theta, arc)
   int  w, h,          /* width and height          */
   int  xleft, ytop;   /* top left corner           */
   int  theta;         /* starting angle (degrees)  */
   int  arc;           /* extent of angle (degrees) */
```

Description

The fill_piearc function draws a pie-shaped wedge that is solid-filled with the current COLOR1. The wedge is bounded by an arc and two straight edges. The arc is taken from an ellipse in standard position, with its major and minor axes parallel to the coordinate axes. The two straight edges are defined by lines connecting the end points of the arc with the center of the ellipse.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.

- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
/*******************************/
/*     Draw animated pieman    */
/*******************************/
int w, h, x, y, t, dt;

init_video(1);
init_grafix();
t = dt = 8;
w = h = 80;
for (x = -120, y = 350; x < 720; x += 8) {
     if ((t += dt) > 80 || t <= 0)
         dt = -dt;
     delay(0);
     init_screen();
     fill_piearc(w, h, x, y, t/2, 360-t);
     fill_piearc(w, h, x+40, y, -15, 30);
}
```

Syntax

```
void fill_polygon(n, linelist, ptlist)
   int   n;              /* number of edges              */
   short linelist[];     /* list of edges                */
   short ptlist[];       /* list of vertex coordinates *//
```

Description

The fill_polygon function fills a polygon given a list of lines representing the edges of the polygon. No restrictions are placed on the shape of the polygons filled by the function: edges can cross each other, filled areas can contain holes, and two or more filled regions can be disconnected from each other. The polygon is solid-filled with COLOR1.

The function requires three input arguments:

- Argument `n` defines the number of vertices in the polygon.
- The second argument, `linelist`, is an array of type short. Each pair of elements in the `linelist` array defines an edge: the first of the two elements defines the starting vertex of the edge, and the second defines the ending vertex. Each element of the `linelist` array is an index into the `ptlist` array.
- The third argument, `ptlist`, is an array of type short. Each pair of adjacent 16-bit elements contains the X and Y coordinates, respectively, of a vertex.

Each pair of adjacent 16-bit elements in the `ptlist` array is an X coordinate followed by a Y coordinate. Each pair of adjacent 16-bit elements in the `linelist` array is a pair of indices into the `ptlist` array.

For example, the first edge that is drawn is specified in array elements, `linelist[0]` and `linelist[1]`. Assume that these contain index values 4 and 7, respectively. The starting coordinates for the line defining the edge are contained in `ptlist[2*4]` and `ptlist[2*4+1]`. The ending coordinates are contained in `ptlist[2*7]` and `ptlist[2*7+1]`.

The individual elements of the `linelist` array are assigned as follows:

```
linelist[0]     = starting vertex for edge 0
linelist[1]     = ending vertex for edge 0
linelist[2]     = starting vertex for edge 1
linelist[3]     = ending vertex for edge 1
      .
      .
      .
linelist[2n]    = starting vertex for edge n-1
linelist[2n+1]  = ending vertex for edge n-1
```

The individual elements for the `ptlist` array are assigned as follows:

```
ptlist[0]       = x coordinate value for point 0
ptlist[1]       = y coordinate value for point 0
ptlist[2]       = x coordinate value for point 1
ptlist[3]       = y coordinate value for point 1
      .
      .
      .
ptlist[2m]      = x coordinate value for point m-1
ptlist[2m+1]    = y coordinate value for point m-1
```

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
static short xy[] = {
    440,80, 540,380, 500,380, 472,300, 368,300,
    340,380, 300,380, 400,80, 420,140, 459,260,
    381,260, 277,133, 248,346, 300,340, 222,138,
    319,147, 204,333, 180,311, 340,172, 360,200,
    160,280, 150,240, 368,240, 360,280, 160,200,
    180,168, 340,308
};
static short shape[] = {
    0,1, 1,2, 2,3, 3,4, 4,5, 5,6, 6,7, 7,0, 8,9,
    9,10, 10,8, 11,12, 12,13, 13,14, 14,11, 15,16,
    16,17, 17,18, 18,15, 19,20, 20,21, 21,22, 22,19,
    23,24, 24,25, 25,26, 26,23
};
init_video(1);
init_grafix();
init_screen();
/* Fill overlapping 'A' and '*' shapes */
fill_polygon(27, shape, xy);
```

Syntax

```
void fill_rect(w, h, xleft, ytop)
   int  w, h;              /* width and height */
                           /* of rectangle     */
   int  xleft, ytop;       /* XY coords at top */
                           /* left corner      */
```

Description

The fill_rect function draws a rectangle that is solid-filled with the current COLOR1. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

Example

```
init_video(1);
init_grafix();
init_screen();
/*****************************/
/**  Draw one big rectangle  **/
/**    and four little ones   **/
/*****************************/
fill_rect(440, 280, 100, 100);
set_color1(0x11111111);       /* Assume 4 bits/pixel */
fill_rect(420, 30, 110, 110);
fill_rect(220, 220, 110, 150);
fill_rect(190, 150, 340, 150);
fill_rect(190, 60, 340, 310);
```

Syntax

```
float *fix_to_float(n, in_array, out_array)
   typedef long FIX;    /* fixed-point format          */
   int   n;             /* number of elements to be    */
                        /* converted                   */
   FIX   in_array[];    /* array of fixed-point values */
   float out_array[];   /* array of float values       */
```

Description

The fix_to_float function converts an array of fixed-point values to single-precision floating-point values. Elements of input array are 32-bit, 2s complement, fixed-point numbers with the binary point located between the 16 LSBs and 16 MSBs. Elements of output array are of type float.

The input arguments include:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

A pointer to the first element of the output array is returned.

Example

```
typedef long FIX;
long  xyz1[9] = { 0,-58,0, 50,29,0, -50,29,0 };
FIX   xyz2[9];
float xyz3[9];

long_to_fix(9, xyz1, xyz2);
fix_to_float(9, xyz2, xyz3);
```

Syntax

```
long *fix_to_long(n, in_array, out_array)
   typedef long FIX;
   int  n;                  /* number of elements to */
                            /* be converted          */
   FIX  in_array[];         /* array of fixed-point
                            /* values                */
   long out_array[];        /* array of integers     */
```

Description

The fix_to_long function converts an array of fixed-point numbers to long integers. Elements of the input array are 32-bit, 2s complement, fixed-point numbers with the 16 LSBs to the right of the binary point. Elements of the output array are 32-bit, 2s complement integers (C type long). The conversion from fixed-point format is done by simply shifting the elements right by 16 (truncation with sign extension).

The input arguments include:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

A pointer to the first element of the output array is returned.

The value returned by the function is a pointer to the output array, `out_array`.

Example

```
typedef long FIX;
static  FIX xy1[2] = { 0,-150<<16 };
long xy2[2];

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
for (;;) {
    fix_to_long(2, xy1, xy2);
    delay(0);
    init_screen();
    draw_line(0, 0, xy2[0], xy2[1]);
    xy1[0] -= xy1[1] >> 6;
    xy1[1] += xy1[0] >> 6;
}
```

Syntax

```
short *fix_to_short(n, in_array, out_array)
   typedef long FIX;
   int   n;              /* number of elements to be */
                         /* converted                */
   FIX   in_array[];     /* array of fixed-point     */
                         /* values                   */
   short out_array[]; /* array of integers           */
```

Description

The fix_to_short function converts an array of fixed-point numbers to short integers. Elements of the input array are 32-bit, 2s complement, fixed-point numbers whose 16 LSBs are to the right of the binary point. Elements of the output array are 16-bit, 2s complement integers (C type short). The conversion from fixed-point format is done by simply shifting the elements right by 16 (truncation).

The input arguments include:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

The value returned by the function is a pointer to the output array, `out_array`.

Example

```
typedef long FIX;
static  short ptlist[] = { 0,-200, 30,15, -30,15 };
static  short connect[] = { 0, 1, 2 };
FIX xy[6];
int i;

init_video(1);
init_grafix();
init_vuport();
short_to_fix(6, ptlist, xy);
set_origin(320, 240);
for ( ; ; ) {
    for (i = 0; i <= 2; ++i) {
        xy[2*i] -= xy[2*i+1] >> 6;
        xy[2*i+1] += xy[2*i] >> 6;
    }
    fix_to_short(6, xy, ptlist);
    delay(0);
    init_screen();
    fill_convex(3, connect, ptlist);
}
```

Syntax

```
.global FIX2FL
```

Description

The FIX2FL function converts a fixed-point number to a single-precision floating-point number. The format used for a 32-bit, 2s complement, fixed-point number places the binary point between the 16 LSBs and 16 MSBs. Refer to discussion of single-precision floating-point format in Appendix D of the *TMS34010 C Compiler User's Guide*.

> **Note:**
>
> You cannot call the FIX2FL function from a C program. You must call it from an an assembly language program by using the EXGPC instruction. Arguments are passed through the TMS34010 register file.

The argument is passed to this function in register A10 and the result is returned in A10. The prior contents of A8, A11, and A10 are lost. The function is called with an EXGPC instruction using register A5.

Example

```
*********************************************************
***   Convert fixed-point array to floating point.    ***
*********************************************************

      MOVI  FIX2FL,A4   ; load address of FIX2FL routine
LOOP:
      MOVE  *A1+,A10,1  ; get next element from input
      MOVE  A4,A5       ; copy address of FIX2FL routine
      EXGPC A5          ; execute FIX2FL routine
      MOVE  A10,*A2+,1  ; copy converted element to output
      DSJS  A0,LOOP     ; more elements to convert?
```

Syntax

```
.global FL2FIX
```

Description

The FL2FIX function converts a single-precision floating-point value to a fixed-point value. The format used for a 32-bit, 2s complement, fixed-point number places the binary point between the 16 LSBs and 16 MSBs.

> **Note:**
>
> You cannot call the FL2FIX function from a C program. You must call it from an an assembly language program by using the EXGPC instruction. Arguments are passed through the TMS34010 register file.

The argument is passed to this function in register A10 and the result is returned in A10. The prior contents of A8, A10, and A11 are lost. The function is called with an EXGPC instruction using register A5.

Example

```
*********************************************************
***   Convert floating-point array to fixed point.  ***
*********************************************************
      MOVI  FL2FIX,A4   ; load address of FL2FIX routine
LOOP:
      MOVE  *A1+,A10,1 ; get next element from input
      MOVE  A4,A5       ; copy address of FL2FIX routine
      EXGPC A5          ; execute FL2FIX routine
      MOVE  A10,*A2+,1 ; copy converted element to output
      DSJS  A0,LOOP     ; more elements to convert?
```

Syntax

```
.global FL_ADD
```

Description

The FL_ADD function adds two single-precision floating-point values. Refer to discussion of single-precision floating-point format in Appendix D of the *TMS34010 C Compiler User's Guide*.

> **Note:**
>
> You cannot call this function from a C program. You must call it from an assembly language program by using the EXGPC instruction.

The single-precision arguments are passed via registers A9 and A10; the result is returned in A10. The prior contents of A7 through A12 are lost. The function is called with an EXGPC function using register A5.

Example

```
 ************************************************
 * Vector addition routine: add array x to array y. *
 ************************************************
      .global  FL_ADD       ; declare function external
      MOVE     *-A14,A0,1   ; get count
      MOVE     *-A14,A1,1   ; get pointer to x array
      MOVE     *-A14,A2,1   ; get pointer to y array
LOOP:
      MOVE     *A1+,A9,1    ; get x[i]
      MOVE     *A2,A10,1    ; get y[i]
      MOVI     FL_ADD,A5    ; put entry point in A5
      EXGPC    A5
      MOVE     A10,*A2+,1   ; store sum of x[i], y[i]
      DSJ      A0,LOOP      ; loop again if --count > 0
```

Syntax

```
.global FL_COS
```

Description

The FL_COS function calculates the cosine of a real number that represents an angle expressed in radians. Both the input value and return value are single-precision floating-point values.

> **Note:**
>
> You cannot call this function from a C program. You must call it from an assembly language program by using the CALLA instruction.

The single-precision argument to this function is passed via register A10 and the result is returned in A10. The prior contents of A10 and A8 are lost.

Example

```
.global  FL_COS        ;declare function external
MOVE     *A1+,A10,1    ;get angle in A10
CALLA    FL_COS        ;returns cos(angle) in A10
```

Syntax

```
.global FL_MULT
```

Description

The FL_MULT function multiplies two single-precision floating-point values. The result is also a single-precision floating-point value. Refer to discussion of single-precision floating-point format in Appendix D of the *TMS34010 C Compiler User's Guide*.

> **Note:**
>
> You cannot call this function from a C program. You must call it from an assembly language program by using the EXGPC instruction.

Arguments are passed to the function in registers A9 and A10, and the result is returned in A10. The prior contents of A9 through A12 are lost. The function is called with an EXGPC function using register A5.

Example

```
*********************************************************
* Vector multiply routine: multiply array y by array x.
*
*********************************************************
      .global   FL_MULT       ; declare function external
      MOVE      *-A14,A0,1    ; get count
      MOVE      *-A14,A1,1    ; get pointer to x array
      MOVE      *-A14,A2,1    ; get pointer to y array
LOOP:
      MOVE      *A1+,A9,1     ; get x[i]
      MOVE      *A2,A10,1     ; get y[i]
      MOVI      FL_MULT,A5    ; put entry point in A5
      EXGPC     A5
      MOVE      A10,*A2+,1    ; store x[i]*y[i] in y[i]
      DSJ       A0,LOOP       ; loop again if --count>0
```

Syntax

```
.global FL_SIN
```

Description

The FL_SIN function calculates the sine of a real number that represents an angle expressed in radians. Both the input value and return value are single-precision floating-point values.

> **Note:**
>
> You cannot call this function from a C program. You must call it from an assembly language program by using the CALLA instruction.

The single-precision argument to this function is passed via register A10 and the result is returned in A10. The prior contents of A10 and A8 are lost.

Example

```
.global   FL_SIN        ; declare function external
MOVE      *A1+,A10,1    ; get angle in A10
CALLA     FL_SIN        ; returns sin(angle) in A10
```

Syntax

```
FIX *float_to_fix(n, in_array, out_array)
   typedef long FIX;    /* put this before function    */
                        /* definition                  */
   int   n;             /* number of elements to be    */
                        /* converted                   */
   float in_array[];    /* array of float values       */
   FIX   out_array[];   /* array of fixed-point values */
```

Description

The float_to_fix function converts an array of single-precision floating-point values to fixed-point values. Elements of the input array are of type float. Elements of the output array are 32-bit, 2s complement, fixed-point numbers with the binary point located between the 16 LSBs and 16 MSBs. The input arguments include:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

A pointer to the first element of the output array is returned.

Example

```
typedef long FIX;
static  float xyz1[9] = {
            0.,-58.,0., 50.,29.,0., -50.,29.,0.
        };
FIX xyz2[9];

float_to_fix(9, xyz1, xyz2);
```

Syntax

```
double floor(x)
   double x;
```

Description

The floor function returns a double-precision floating-point number representing the largest integer less than or equal to the input argument x.

Example

```
extern double floor();

double answer;

answer = floor(3.1415, &exp);

/* after execution, answer will be 3.0 */
```

Syntax

```
double fmod(x, y)
   double x, y;
```

Description

The fmod function calculates the floating-point remainder of x/y. The input arguments and return value are all double-precision floating-point numbers. If the quotient x/y cannot be represented, the result is undefined, and the floating-point error routine fp_error is called. Otherwise, the function returns the value x-*i**y, where *i* is an integer such that, if y is nonzero, the result has the same sign as x and a magnitude less than the magnitude of y.

If input argument y is 0, the return value is 0, and an error code of 8 is transmitted to the fp_error routine. If input argument x is $+\infty$ or $-\infty$, the return value is 0, and an error code of 9 is transmitted to fp_error (see the floating-point description in the *TMS34010 C Compiler User's Guide*).

Example

```
/************************************/
/*     fmod returns double result   */
/************************************/
extern double fmod();
double x, y, r;

x = 1.0;
y = 2.0;
r = fmod(x, y);    /* fmod returns 1.0 */
```

Syntax

```
void frame_oval(w, h, xleft, ytop, dx, dy)
   int  w, h;              /* width and height of */
                           /* enclosing rectangle */
   int  xleft, ytop;       /* coordinates at top  */
                           /* left corner         */
   int  dx, dy;            /* width and height of */
                           /* frame border        */
```

Description

The frame_oval function solid-fills an ellipse-shaped frame with the current COLOR1. The frame consists of a filled region between two concentric ellipses. The portion of the screen enclosed by the frame is not altered.

The outer ellipse is specified in terms of the minimum enclosing rectangle in which it is circumscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The thickness of the frame in the X and Y dimensions is defined by two additional arguments, `dx` and `dy`, which specify the horizontal and vertical distances, respectively, between the outer and inner ellipses.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, dx, dy;

init_video(1);
init_grafix();
init_screen();
w  = 480;
h  = 360;
x  = 80;
y  = 60;
dx = 40;
dy = 30;
frame_oval(w, h, x, y, dx, dy);
```

Syntax

```
void frame_rect(w, h, xleft, ytop, dx, dy)
   int  w, h;              /* width and height of */
                           /* enclosing rectangle */
   int  xleft, ytop;       /* coordinates at top  */
                           /* left corner         */
   int  dx, dy;            /* width and height of */
                           /* frame border        */
```

Description

The frame_rect function solid-fills a rectangular frame with the current COLOR1. The frame consists of a filled region between two concentric rectangles. The portion of the screen enclosed by inner edge of the frame is not altered.

The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The thickness of the frame in the X and Y dimensions is defined by two additional arguments, `dx` and `dy`, which specify the horizontal and vertical distances, respectively, between the outer and inner rectangles.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, dx, dy;

init_video(1);
init_grafix();
init_screen();
w = 480;
h = 360;
x = 80;
y = 60;
dx = 40;
dy = 30;
frame_rect(w, h, x, y, dx, dy);
```

Syntax

```
double frexp(value, exp)
   double value;  /* input floating-point number */
   int *exp;      /* pointer to exponent */
```

Description

The frexp breaks a double-precision floating-point number into a normalized fraction and an exponent. The fraction is returned by the function, and the exponent is placed in the integer pointed to by `exp`.

Example

```
extern double frexp();

double fraction;
int exp;

fraction = frexp(3.1415, &exp);

/* after execution, fraction will be .1415, and
   exp will contain 3 */
```

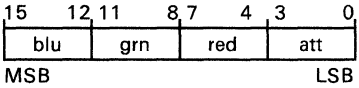
Syntax

```
void getall_palet(palet_array, reg_mask, y)
   short palet_array[16]; /* palette reg. values  */
   int   reg_mask;        /* register-select mask */
   int   y;                * scan line            */
```

Description

The getall_palet function reads multiple color palette registers into the destination array. The purpose of this function is to make selected palette register values available for inspection and modification. This function assumes a TMS34070 color palette or functional equivalent. The pixel size is therefore four bits, and the palette contains 16 registers. The values contained in the palette registers can change on a line-by-line basis.

Each 16-bit palette register is organized according to the following format:



The red, green, and blue intensity fields are 4-bit, unsigned binary numbers. The attribute field contains a color-repeat bit that is set to one to enable automatic filling by the palette device. See the *TMS34070 User's Guide* for details.

- The first argument, `palet_array`, is the array into which the register values from the color palette are written.
- The second argument, `reg_mask`, specifies which of the 16 palette registers are written to the destination array. Bit positions 0 to 15 in the mask enable (if 1) or disable (if 0) writing the corresponding palette registers. For example, a mask value of 0017_{16} enables writing of palette registers 0, 1, 2 and 4 into `palet_array` elements 0, 1, 2 and 4.
- The third argument, `y`, designates at which scan line the color palette is examined. This argument is necessary because the palette can differ from line to line. Scan lines are numbered in ascending order beginning with 0 at the top of the screen.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
short temp_array[16];

/*************************************/
/* Copy palette registers 5, 6, 8 and 9 */
/* from the 18th scan line             */
/*************************************/

getall_palet(temp_array, 0x0360, 17);
```

Syntax `int get_ascent()`

Description The get_ascent function returns the value of the ascent parameter for the current font. The ascent value is the number of vertical pixels from the baseline to the top of the tallest character in the font. The ascent for the current font is defined in the font structure; the get_ascent function retrieves the ascent value from the structure.

> **Note:**
>
> Before you call the get_ascent function, call the init_text function to initialize the text data structures.

Example

```
/*************************************/
/* Draw text flush with top of screen */
/*************************************/
static char *s = "Hello world.";
int w, h, x, y;

init_video(1);
init_grafix();
init_text();                /* sets up default font */
init_screen();
x = y = 0;                    /* top left corner      */
w = get_width(s);
h = get_ascent();
set_color1(0x11111111);  /* assume 4 bits/pixel  */
fill_rect(w, h, x, y);
transp_on();
set_color1(0x77777777);
draw_string(x, y+h, s);
```

Syntax `int get_descent()`

Description The get—descent function returns the value of the descent parameter for the current font. The descent descent is the the vertical distance measured in pixels from the baseline to the lowest descender in the character set. The descent for the current font is defined in the font structure; the get—descent function retrieves the descent value from the structure.

> **Note:**
>
> Before you call the get—descent function, call the init—text function to initialize the text data structures.

Example

```
/***********************************/
/* Draw line just below descenders */
/***********************************/
static char *s = "jumping jimminy";
int w, h, x, y;

init_video(1);
init_grafix();
init_text();                    /* sets up default font */
init_screen();
x = 0;
y = get_ascent();
w = get_width(s);
h = get_descent();
draw_string(x, y, s);
set_color1(0x11111111);         /* assume 4 bits/pixel  */
fill_rect(w, 1, x, y+h);
h = char_high() + 1;
```

Syntax `int get_first_ch()`

Description The get—first—ch function returns the ASCII character code for the first character present in the current font. All characters whose codes are less than the return value correspond to "missing" characters; that is, their character images are omitted from the font structure.

> **Note:**
>
> Before you call the get—first—ch function, call the init—text function to initialize the text data structures.

Example

```
/*****************************************************/
/* Draw all characters implemented in current font */
/*****************************************************/
unsigned char c;
int x, y;

init_video(1);
init_grafix();
init_text();                    /* sets up default font */
init_screen();
x = 10;
y = 100;
for (c = get_first_ch(); c <= get_last_ch();
    ++c, x += char_wide_max()) {
       if (x > 640) {
           x = 10;
           y += 50;
       }
     draw_char(x, y, c);
}
```

Syntax `int get_font_max()`

Description The get—font—max function returns the maximum number of fonts that can be installed simultaneously (see the description of the install—font function also). If a value of *n* is returned, font indices can range from 0 to *n*-1.

Example

```
int n;

n = get_font_max();
```

Syntax `int get_last_ch()`

Description The get_last_ch function returns the ASCII character code for the last character present in the current font. All characters whose codes are greater than the return value correspond to "missing" characters; that is, their character images are omitted from the font structure

> **Note:**
>
> Before you call the get_last_ch function, call the init_text function to initialize the text data structures.

Example

```
/*************************************************/
/* Draw all characters implemented in current font */
/*************************************************/
unsigned char c;
int x, y;

init_video(1);
init_grafix();
init_text();                    /* sets up default font */
init_screen();
x = 10;
y = 100;
for (c = get_first_ch(); c <= get_last_ch();
     ++c, x += char_wide_max()) {
        if (x > 640) {
            x = 10;
            y += 50;
        }
     draw_char(x, y, c);
}
```

Syntax `int get_leading()`

Description The get_leading function returns the leading value for the current font. The leading is the empty space between rows of text; that is, it is the number of vertical pixels from the lowest descenders of one row of text to the tallest characters of the line of text below it. This function retrieves the leading value from the font structure.

> **Note:**
>
> Before you call the get_leading function, call the init_text function to initialize the text data structures.

Example

```
/*********************************************/
/* Draw lines between successive rows of text */
/*********************************************/
static char s[] = "The quick brown fox...";
int x, y, dx;

init_video(1);
init_grafix();
init_text();          /* sets up default font */
init_screen();
transp_on();
dx = (640 - get_width(s)) / (480 / char_high());
for (x = 0, y = get_ascent(); y < 480;
     x += dx, y += char_high()) {
     draw_string(x, y, s);
     fill_rect(640, 1, 0, y + get_descent() +
                  get_leading()/2);
}
```

Syntax `int get_patn_max()`

Description The get—patn—max function returns the maximum number of patterns that can be installed at any one time. If return value is *n*, available range of vertices is 0 to *n*-1. The maximum number of patterns is a function of the size of the pattern table data structure.

Example

```
int w, h, x, y, dx, dy, pmax, patn;

init_video(1);
init_grafix();              /* Install default patterns */
init_screen();
set_color0(0x11111111);     /* Assume 4 bits/pixel      */

/*********************************/
/* Display all available patterns */
/*********************************/

pmax = get_patn_max();
x  = y = 0;
w  = 84;
h  = 68;
dx = 96;
dy = 80;
for (patn = 0; patn < pmax; select_patn(++patn)) {
    patnfill_rect(w, h, x, y);
    if ((x += dx) > 640 - w) {
         x = 0;
         y += dy;
    }
}
```

Syntax

```
int  get_pixel(x, y)
  int  x, y;      /* coordinates of pixel */
```

Description

The get—pixel function returns the value of the pixel at coordinates (x,y). The coordinates are relative to the viewport origin. Given a pixel size of *n* bits, the pixel is contained in the *n* LSBs of the return value (the MSBs are 0s).

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int    xs, ys, xd, yd;
static char s[] = "topsy turvy";
short  buf[640/4];            /* line buffer for zoom */

init_video(1);
init_grafix();
init_text();
init_screen();
draw_string(0, get_ascent(), s);

/*********************************/
/* Flip and mirror original text */
/*********************************/

for (ys = 0, yd = 29; ys <= 19; ++ys, --yd)
    for (xs = 0, xd = 89; xs <= 89; ++xs, --xd)
         put_pixel(get_pixel(xs, ys), xd, yd);
zoom_rect(40, 40, 0, 0, 160, 160, 240, 160, buf);
```

Syntax

```
long get_pmask()
```

Description

The get_pmask function returns the value of the plane mask (TMS34010 PMASK register). Although only the 16 LSBs of the PMASK register are implemented in the TMS34010, the plane mask is 32 bits for the sake of upward compatibility with future GSPs.

The plane mask designates which bits within a pixel are protected against writes, and affects all operations on pixels. The protected bits are replicated throughout the 32-bit plane mask in a manner similar to that used for the COLOR0 and COLOR1 values. The 1s in the plane mask specify protected bits in the destination pixel that cannot be modified, while the 0s specify bits that can be altered. The plane mask can be altered by means of a call to the set_pmask function. See the *TMS34010 User's Guide* for a further discussion of plane masking.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long mask;

/* Assume pixel size is 4 bits */
init_video(1);
/* sets PMASK = all 0s         */
init_grafix();
set_pmask(0x11111111);
/* Write protect pixel bit #0  */
mask = get_pmask();
/* Return value = 0x00001111   */
```

Syntax

```
long get_ppop()
```

Description

The get—ppop function returns the code for the current pixel processing operation (the PPOP field in the TMS34010's CONTROL register). The 5-bit PPOP code resides in the 5 LSBs of the return value; all higher order bits are 0s.

The PPOP code determines the manner in which pixels are combined (logically or arithmetically) during drawing operations. A new PPOP code can be selected by means of the set—ppop function. Legal PPOP codes are in the range 0 to 21. The effects of the 22 different codes are described in the *TMS34010 User's Guide*.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
long ppop;

init_video(1);       /* PPOP = all 0s (REPLACE) */
init_grafix();
set_ppop(10);        /* select XOR              */
ppop = get_ppop();   /* returns PPOP = 10       */
```

Syntax

```
int get_psize()
```

Description

The get_psize function returns the pixel size (contents of TMS34010 PSIZE register). The pixel size for the display system is typically a constant defined within the init_video function. The TMS34010 supports pixels of 1, 2, 4, 8, and 16 bits. Future GSPs may support additional pixel sizes.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long psize;

init_video(1);              /* sets PSIZE */
init_grafix();
psize = get_psize();
```

Syntax

```
void get_rect(w, h, xleft, ytop, darray, dpitch)
  int   w, h;          /* width and height of      */
                       /* source rectangle         */
  int   xleft, ytop;   /* coordinates at top left  */
                       /* corner                   */
  short darray[]       /* destination pixel array  */
  long  dpitch;        /* destination pitch        */
```

Description

The get—rect function copies pixels contained in a rectangular area of the screen into a packed pixel array. The first four arguments define the source rectangle (the rectangular area of the screen):

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

`w` and `h` must be nonnegative.

The last two arguments describe the destination array:

- The destination array, `darray`, **and**
- The pitch of the array, `dpitch`.

The array pitch is the difference in the starting addresses of two adjacent pixel rows of the array. The pitch must be a positive multiple of the pixel size. The minimum pitch is the product of `w` and the pixel size. The pixel size can be obtained by calling the get—psize function. The destination array must be large enough to contain the source array. The minimum number of bits required in the destination array is the product of `h` and `dpitch`.

The source rectangle is clipped to positive XY coordinate space before being copied. Portions of the source array lying in negative XY space are not copied, and the corresponding portions of the destination array remain unaltered. Portions of the source array lying outside the current viewport and clipping rectangle are **not** clipped unless they lie in negative XY space.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int   w, h, x, y, pitch, i;
short buf[80*60*4/16];

init_video(1);
init_grafix();
init_screen();
w = 80;
h = 60;
x = 280;
y = 210;
pitch = w * get_psize();
/***   Draw picture    ***/
fill_rect(w-2, h-2, x+1, y+1);
set_color0(0x11111111);   /* Assume 4 bits/pixel */
set_color1(0x44444444);
patnframe_oval(w-2, h-2, x+1, y+1, 20, 15);
/***   Capture picture from screen    ***/
get_rect(w, h, x, y, buf, pitch);
/***    Put picture onto screen    ***/
for (i = 25, x = 0, y = -150<<16; i > 0; --i) {
    x += y >> 2;
    y -= x >> 2;
    put_rect(buf, pitch, w, h, (x>>16)+280, (y>>16)+210);
}
```

Syntax `int get_transp()`

Description The get—transp function returns the state of the transparency enable bit (the T bit from the TMS34010's CONTROL register). A value of 1 is returned if transparency is enabled; otherwise, 0 is returned.

Transparency is an attribute that affects text drawing and pattern fills. If transparency is enabled, and the result of a pixel processing operation is 0, the destination pixel is not altered. If transparency is disabled, the destination pixel is replaced by the result of the pixel processing operation regardless of the value of that result. See the *TMS34010 User's Guide* for a further discussion of transparency.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int t;

init_video(1);       /* Disables transparency */
init_grafix();
t = get_transp();    /* Return value = 0      */
transp_on();
t = get_transp();    /* Return value = 1      */
```

Syntax

```
int get_vuport_max()
```

Description

The get_vuport_max function returns the maximum number of viewports that can be open at any one time. If the return value is a number *n*, the range of indices for available viewports is 0 to *n*-1.

Example

```
int v;

init_video(1);
init_grafix();
init_vuport();
v = get_vuport_max();
```

Syntax

```
int get_width(s)
   char *s;  /* ASCII char string terminated by NULL */
```

Description

The get_width function returns the width of a character string s. The width is the number of pixels from the left edge to the right edge of the string, where the string is drawn in the current font. The spacing between characters is included in this number, including any adjustments to the font's default spacing due to a previous call to the add_text_space function.

The character string is in standard C format; that is, the string is a sequence of ASCII character codes terminated by a NULL (ASCII code = 0).

> **Note:**
>
> Before you call the get_width function, call the init_text function to initialize the text data structures.

Example

```
#include "fntstruc.h"
#define  XC 320
#define  YC 240

extern FONT corpus_christi29;
static char *s[] = {
    "The get_width function",
    "is easily used",
    "to center text",
    "on your screen."
};
int x, y, i;

init_video(1);
init_grafix();
init_text();
init_screen();
set_color1(0x11111111);
draw_line(XC, 0, XC, 2*YC);   /* crosshairs          */
draw_line(0, YC, 2*XC, YC);
install_font(1,&corpus_christi29);
y = YC - 4*char_high()/2 + get_ascent();
transp_on();
set_color1(0x33333333);
for (i = 0; i <= 3; ++i) {
    x = XC - get_width(s[i])/2;
    draw_string(x, y, s[i]);  /* centered text       */
    y += char_high();
}
```

Syntax `void init_grafix()`

Description The init_grafix function initializes the graphics environment. It sets up the data structures for the graphics functions, and assigns default values to system parameters. You should call this function before performing any graphics or text drawing operations.

> **Note:**
>
> Call the init_video function before you call the init_grafix function.

The init_grafix function performs these tasks:

- Disables pixel transparency.
- Sets pixel processing operation to replace.
- Enables all color planes (PMASK all 0s).
- Moves XY origin to default position at top left corner of screen.
- Sets visibility rectangle (clipping window) to full screen.
- Sets drawing pen to default size.
- Initializes pattern data structures, and installs default patterns.

Example

```
int x, y, i;

init_video(1);          /* Initialize video    */
init_grafix();          /* Initialize graphics */
init_screen();          /* Initialize screen   */
/***   Ready to draw something...   ***/
for (i = 100, x = 0, y = 200<<16; i > 0; --i) {
     draw_line(320, 240, 320+(x>>16), 240+(y>>16));
     x += y >> 4;
     y -= x >> 4;
}
```

Syntax

```
void init_matrix(matrix)
   typedef long FIX;
   FIX matrix[16];
```

Description

The init_matrix function initializes a 16-element, fixed-point array to a 4×4 identity matrix. The format used for a 32-bit, 2s complement, fixed-point value places the binary point between the 16 LSBs and 16 MSBs. A fixed-point value of 1 can be represented in C as long integer constant 0x10000.

The resulting identity matrix is ready to be transformed by the rotate, scale and translate functions. See *Principles of Computer Graphics* (Newman and Sproull) for a detailed discussion of homogeneous 3D transformations.

The matrix elements are mapped into the array in row-major order as follows:

`matrix[0] = 1;`	matrix element a_{00}	begin 1st row
`matrix[1] = 0;`	matrix element a_{01}	
`matrix[2] = 0;`	matrix element a_{02}	
`matrix[3] = 0;`	matrix element a_{03}	
`matrix[4] = 0;`	matrix element a_{10}	begin 2nd row
`matrix[5] = 1;`	matrix element a_{11}	
`matrix[6] = 0;`	matrix element a_{12}	
`matrix[7] = 0;`	matrix element a_{13}	
`matrix[8] = 0;`	matrix element a_{20}	begin 3rd row
`matrix[9] = 0;`	matrix element a_{21}	
`matrix[10] = 1;`	matrix element a_{22}	
`matrix[11] = 0;`	matrix element a_{23}	
`matrix[12] = 0;`	matrix element a_{30}	begin 4th row
`matrix[13] = 0;`	matrix element a_{31}	
`matrix[14] = 0;`	matrix element a_{32}	
`matrix[15] = 1;`	matrix element a_{33}	

Example

```
typedef long FIX;
static  FIX rotation[3]  = { 0, 0, 0 };
static  FIX translat1[3] = { -320, -240, 0 };
static  FIX translat2[3] = { 320, 240, 0 };
static  long xyz[] = {
    320,40,0, 340,240,0, 320,260,0, 300,240,0
};
static short connect[8] = { 0,1, 1,2, 2,3, 3,0 };
FIX   matrix[16];
FIX   verts[12];
short xy[8];
int   angle;

init_video(1);
init_grafix();
long_to_fix(3, translat1, translat1);
long_to_fix(3, translat2, translat2);
for (;;)
    for (angle = 0; angle < 360; ++angle) {
        init_matrix(matrix);
        translate(matrix, translat1);
        rotation[0] = angle << 16;
        rotate(matrix, rotation);
        translate(matrix, translat2);
        long_to_fix(12, xyz, verts);
        transform(matrix, 4, verts);
        vertex_to_point(4, verts, xy);
        delay(0);
        init_screen();
        draw_oval(420, 420, 110, 30);
        draw_polyline(4, connect, xy);
    }
```

Syntax

```
void init_palet()
```

Description

The init_palet function initializes the color look-up table to default palette values.

> **Note:**
>
> Call the init_video and init_grafix functions before you call the init_palet function.

The results of this function are system dependent. In the case of the TMS34010 software development board, the pixel size is four bits, and the TMS34070 color palette contains 16 registers. When the SDB is configured for analog RGB output, the following default colors are assigned to pixel values 0 to 15 by the init_palet function:

Pixel Value	Color	Pixel Value	Color
0	black	8	black
1	red	9	light red
2	green	10	light green
3	yellow	11	light yellow
4	blue	12	light blue
5	magenta	13	light magenta
6	cyan	14	light cyan
7	white	15	gray

The current implementation of the init_palet function assumes that the TMS34070 color palette is configured in line-load mode (see *TMS34010 Software Development Board User's Guide* for description of hardware jumper options). The default palette is updated over the entire screen.

Example

```
long w, h, x, y, c, dc;

init_video(1);     /* Configure SDB for analog output */
init_grafix();
init_text();
clear_screen(0);   /* Fill frame buffer with 0s       */
init_palet();      /* Install default color palette   */
c = 0;
dc = 0x11111111;
w = 30;
h = 440;
for (x = 5, y = 30; x < 639; x += 40, c += dc) {
    set_color1(c);
    fill_rect(w, h, x, y);
    set_color1(0x77777777);
    draw_rect(w+1, h+1, x-1, y-1);
}
draw_string(5, 20, "Default Color Palette");
```

Syntax

```
void init_screen()
```

Description

The init_screen function initializes the screen. The entire frame buffer is cleared (filled with 0s). If the system contains a color lookup table, the table is loaded with the default color palette.

> **Note:**
>
> Call the init_video and init_grafix functions before you call the init_screen function.

The implementation of this function is system-dependent. In the case of a TMS34010 software development board configured for analog output, the RGB signals to the monitor are assumed to be generated by a TMS34070 color palette configured in line-load mode (see the *TMS34010 Software Development Board User's Guide* for a description of hardware jumper options). The init_screen function loads the palette region of the frame buffer (the first 256 bits of each scan line) with default color values, and clears the remainder of the frame buffer to 0s. If the SDB is instead configured for digital output, the TMS34070 is not used, and the init_screen function fills the entire frame buffer with 0s.

Example

```
long w, h, x, y, c, dc;

init_video(1);      /* Configure SDB for analog output
*/
init_grafix();
init_text();
init_screen();     /* Clear screen, load default palette
*/
c  = 0;
dc = 0x11111111;
w  = 30;
h  = 440;
for (x = 5, y = 30; x < 639; x += 40, c += dc) {
    set_color1(c);
    fill_rect(w, h, x, y);
    set_color1(0x77777777);
    draw_rect(w+1, h+1, x-1, y-1);
}
draw_string(5, 20, "Default Color Palette");
```

Syntax `void init_text()`

Description The init—text function initializes text data structures, and installs the default "system" font as font number 0. The additional text spacing increment is initialized to 0, but can be modified by a call to the add—text—space function.

Call the init—text function before you call any of the following functions:

add—text—space	get—first—char
char—high	get—last—char
char—wide—max	get—leading
draw—char	get—width
draw—string	install—font
get—ascent	select—font
get—descent	

Example

```
#include "fntstruc.h"  /* Define FONT type */
#define  XC 320
#define  YC 240
extern FONT corpus_christi29;
static char s[] = "34010";
int x, y;

init_text();     /* Initialize text      */
init_video(1);   /* Initialize video     */
init_grafix();   /* Initialize graphics  */
init_screen();   /* Initialize screen    */
install_font(1, &corpus_christi29); /* Remember the & */
y = YC + get_ascent()/2;
x = XC - get_width(s)/2;
draw_string(x, y, s);  /* center text   */
```

Syntax

```
int init_video(monitor_val)
   int monitor_val;   /* display monitor type */
```

Description

The init—video function initializes video display by setting up the TMS34010's video timing and screen refresh registers to drive a display monitor. This function initializes all system-dependent portions of the graphics environment. The other initialization routines (init—grafix, init—text, and init—vuport) initialize system-*in*dependent portions of the graphics environment.

Argument `monitor_val` specifies the type of monitor present. Arguments in the range 1 to 5 are currently supported, but the init—video function may be enhanced in the future to support additional monitor types. Monitor types that are currently supported include:

`monitor_val`	**Display Configuration**
1	Sets up the SDB to drive an analog RGB monitor for 640x480 resolution. Video crystal = 25 MHz. Monitors compatible with this configuration are: – Princeton Graphics SR-12P Analog RGB – NEC JC-1401P3A Multi-Sync Analog RGB – Sony CPD-1302 Multi-Scan Analog RGB Noninterlaced display with 60-Hz frame rate. Also assumes TMS34070 palette in line-load mode.
2	Sets up the SDB to drive an analog RGB monitor for 640x480 resolution. Video crystal = 25 MHz. Monitors compatible with this configuration are: – IBM 5175 Professional Graphics Display Noninterlaced display with 60-Hz frame rate. Also assumes TMS34070 palette in line-load mode.
3	Sets up the SDB to drive a TTL RGB monitor for 720x300 resolution. Video crystal = 18 MHz. Monitors compatible with this configuration are: – TI PC Color Display Monitor Noninterlaced display with 60-Hz frame rate. Assumes SDB configured for TTL RGB output levels.
4	Sets up the SDB to drive a TTL RGB monitor for 720x512 resolution. Video crystal = 18 MHz. Monitors compatible with this configuration are: – TI PC Color Display Monitor Interlaced display with 30-Hz frame rate. Assumes SDB configured for TTL RGB output levels.
5	Sets up the SDB to drive an analog RGB monitor 448x480 resolution. Video crystal = 25 MHz. This configuration is used to provide double buffering, and is compatible with these monitors: – Princeton Graphics SR-12P Analog RGB – NEC JC-1401P3A Multi-Sync Analog RGB – Sony CPD-1302 Multi-Scan Analog RGB Noninterlaced display with 60-Hz frame rate. Assumes SDB configured for analog RGB output.

> **Note:**
>
> Call the init_video function before calling the init_grafix or init_vuport function.

If an invalid argument is received, the function returns a value of -1 and aborts. If the argument is valid, the function returns a value of 0 as confirmation.

The init_video function performs these tasks:

- Sets up the horizontal and vertical video timing registers.
- Sets up the screen refresh registers.
- Defines screen horizontal and vertical dimensions.
- Defines the memory locations of the screen and workspace buffers.
- Sets the pixel size.
- Sets COLOR0 and COLOR1 to their default values (black and white).
- Sets up the DPTCH and CONVDP registers to the screen pitch.
- Loads screen base address into OFFSET register (B4).
- Sets up the default XY origin at top left of screen.
- Sets the DPYTAP register to 0.

Example

```
int x, y, i;
static char s[] = "Hello World.";

init_text();        /* Initialize text     */
init_video(1);      /* Initialize video    */
init_grafix();      /* Initialize graphics */
init_vuport();      /* Initialize vuport   */
init_screen();      /* Initialize screen   */
set_origin(320, 240);
/***   Ready to draw something...    ***/
for (i = 101, x = 0, y = 200<<16; i > 0; --i) {
     draw_line(x>>17, y>>17, x>>16, y>>16);
     x += y >> 4;
     y -= x >> 4;
}
draw_string(-get_width(s)/2, get_ascent()/2, s);
```

Syntax `void init_vuport()`

Description The init—vuport function initializes the viewport data structures and opens viewport 0, the "system" viewport. All other viewports remain closed until they are explicitly opened.

> **Note:**
>
> Before you call this function, call the init—grafix and init—video functions.

Call the init—vuport function before using any of the following viewport-specific functions:

close—vuport	set—origin
move—vuport	set—cliprect
open—vuport	size—vuport
select—vuport	

Immediately after calling init—vuport, the coordinate origin is located in the upper left corner of the screen, and the viewport and clipping rectangle encompass the entire screen.

Example

```
int index;

init_video(1);          /* Must precede init_vuport */
init_grafix();
init_vuport();          /* Opens viewport 0          */
index = open_vuport(); /* Opens 2nd viewport        */
move_vuport(100, 100);
size_vuport(200, 150);
patnfill_rect(999, 999, 0, 0);   /* Fills viewport  */
```

Syntax

```
int install_font(index, fontname)
  int  index;      /* index to be assigned to font */
  FONT *fontname;  /* font structure               */
```

Description

The install_font function installs a font structure in the font table. The font structure contains the character bitmap and other information necessary to extract the individual character patterns from the bitmap. (See the *TMS34010 Font Library User's Guide* for more information about the font structure.) The installed font becomes the current font and it is selected for subsequent text operations.

The font structure itself is *not* copied into the font table. Instead, a pointer to the font structure is inserted into the font table in the position indicated by argument `index`. This index identifies the font in subsequent transactions. Argument `fontname` is the pointer to the font structure.

The maximum number of fonts *n* that can be simultaneously installed is obtained from the get_font_max function. The legal range of indices is 0 to *n*-1. A value of -1 is returned if `index` is out of range; otherwise, a value of 0 is returned.

> **Note:**
>
> Before calling the install_font function, call the init_text function to initialize the text data structures.

Example

```
#include "fntstruc.h"
extern FONT corpus_christi29;  /* Corpus Christi font  */

 init_video(1);
 init_grafix();
 init_text();        /* default font = corpus_christi16 */
 init_screen();
 install_font(1, &corpus_christi29); /* Don't forget & */
 draw_string(50, 100, "Hello world.");
```

Syntax

```
int install_patn(index, pattern)
   int   index;
   short pattern[16];
```

Description

The install_patn function installs a pattern in the pattern table. The 256 bits of the 16 x 16 two-dimensional bit map are copied into the pattern table location indicated by `index`. The `index` argument identifies the pattern in subsequent transactions. Argument `pattern` is a pointer to the pattern bitmap. This pattern becomes the current pattern and it is selected for subsequent pattern filling operations.

When the installed pattern is later drawn to the screen, the 0s in the pattern are replaced with the current COLOR0, and the 1s in the pattern are replaced with the current COLOR1.

The maximum number of patterns n that can be simultaneously installed is obtained from the get_patn_max function. The legal range of indices is 0 to `n-1`. A value of -1 is returned if index is out of range. Otherwise, a value of 0 is returned.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
typedef enum { FIELDWIDTH = 1 } BIT;
static  BIT mypatn[] = {
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,
    0,0,1,1,1,0,0,0,0,0,0,1,1,1,0,0,
    0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,
    0,0,1,1,1,1,1,0,0,1,1,1,1,1,0,0,
    0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,
    0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,
    0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,
    0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,
    0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,
    0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
    0,0,1,1,1,1,1,1,0,0,1,1,1,1,1,0,0,
    0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,
    0,0,1,1,1,0,0,0,0,0,0,1,1,1,0,0,
    0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};

init_video(1);
init_grafix();          /* Installs default patterns */
init_screen();
install_patn(5, mypatn); /* Assign index = 5         */
set_color0(0x11111111);  /* Expand 0s to this color */
set_color1(0x33333333);  /* Expand 1s to this color */
patnfill_rect(448, 288, 96, 96);
```

Syntax

```
double ldexp(value, exp)
    double value;
    int exp;
```

Description

The ldexp function returns `value` multiplied by 2 raised t the power of `exp`, and is commonly used to rebuild a double-precision floating-point value.

Example

```
extern double ldexp();

double value,result;
int exp;

value = 1.5;
exp = 5;

result = ldexp(value,exp);

/* after execution, result will contain 48.0 */
```

Syntax `char *lib_id()`

Description The lib—id function returns character string identifying library and revision.

Example

```
init_video(1);
init_grafix();
init_text();
init_screen();
draw_string(50, 200, lib_id());
```

Syntax

```
int lmo(n)
   long n;      /* 32-bit integer */
```

Description

The lmo function calculates the bit number of the leftmost one in argument n. The argument is treated as a 32-bit number whose bits are numbered from 0 to 31, where bit 0 is the LSB (the rightmost bit position) and bit 31 is the MSB (the leftmost bit position).

For nonzero arguments, the return value is in the range 0 to 31. If the argument is 0, a value of -1 is returned.

Example

```
int i;

init_video(1);
init_grafix();
init_screen();
for (i = 1; i < 640; ++i)
    draw_line(i, i, 1 << lmo(i), i);
```

Syntax

```
double log(x)
   double x;
```

Description

The log function calculates the natural logarithm of real number x. Both the argument x and the return value are double-precision floating-point values.

If argument x is less than or equal to 0, a value of $-\infty$ is returned, and fp_error is called with an error code of 26 (see the *TMS34010 C Compiler User's Guide* for a description of the fp_error function).

Example

```
extern double log();
float x, y;

x = 2.718282;
y = log(x);          /* Return value = 1.0 */
```

Syntax

```
double log10(x)
   double x;
```

Description

The log10 function calculates the logarithm to the base 10 of x. Both the argument x and the return value are double-precision floating-point values.

If argument x is less than or equal to 0, a value of $-\infty$ is returned, and fp_error is called with an error code of 26 (see the *TMS34010 C Compiler User's Guide* for a description of the fp_error function).

Example

```
extern double log10();
float x, y;

x = 10.0;
y = log10(x);          /* Return value = 1.0 */
```

Syntax

```
FIX *long_to_fix(n, in_array, out_array)
   typedef long FIX;
   int  n;              /* number of elements to be */
                        /* converted                */
   long in_array[];     /* array of integers        */
   FIX  out_array[];    /* array of fixed-point     */
                        /* values                   */
```

Description

The long_to_fix function converts an array of long integers to fixed-point numbers. Elements of the input array are 32-bit, 2s complement integers (C type long). Elements of the output array are 32-bit, 2s complement, fixed-point numbers with the 16 LSBs to the right of the binary point. The conversion from integer format is done by simply shifting the elements left by 16 bits.

The function requires three arguments:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

The value returned by the function is a pointer to the output array, `out_array`.

Example

```
typedef long FIX;
static  short triangle[] = {0,1, 1,2, 2,0};
static  long  xyz1[9] = {0,-116,0, 100,58,0, -100,58,0};
FIX     xyz2[9];
short   xy[6];

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
init_screen();
long_to_fix(9, xyz1, xyz2);
vertex_to_point(3, xyz2, xy);
pen_polyline(3, triangle, xy);
```

Syntax

```
double modf(value, exp)
   double value; /* input floating point number */
   int *exp;     /* pointer to exponent */
```

Description

The modf function breaks a double-precision floating-point value into a signed fraction and a signed integer. The integer is stored at the integer object pointed to by `exp`, and the signed fractional `value` is returned.

Example

```
extern double modf();

double value, ipart, fpart;

value = -3.1415;

fpart = modf(value, &ipart);

/* after execution, ipart will contain -3.0, and
   fpart will contain -0.1415 */
```

Syntax

```
void move_pixel(xs, ys, xd, yd)
   int xs, ys;  /* coordinates of source pixel      */
   int xd, yd;  /* coordinates of destination pixel */
```

Description

The move_pixel function copies a pixel from one screen location to another. Arguments (xs,ys) are the coordinates of the source location. Arguments (xd,yd) are the coordinates of the destination location. Coordinates are relative to the viewport origin.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int    xs, ys, xd, yd;
static char s[] = "topsy turvy";
short  buf[640/4];      /* line buffer (640x4 bits) */

init_video(1);
init_grafix();
init_text();
init_screen();
draw_string(0, get_ascent(), s);
/***    Flip and mirror original text    ***/
for (ys = 0, yd = 29; ys <= 19; ++ys, --yd)
    for (xs = 0, xd = 89; xs <= 89; ++xs, --xd)
         move_pixel(xs, ys, xd, yd);
zoom_rect(40, 40, 0, 0, 160, 160, 240, 160, buf);
```

Syntax

```
void move_rect(w, h, xs, ys, xd, yd)
   int w, h;       /* width and height of      */
                   /* rectangle                */
   int xs, ys;     /* coordinates at top left */
                   /* of source                */
   int xd, yd;     /* coordinates at top left */
                   /* of destination           */
```

Description

The move_rect function copies pixels from one rectangular region of the screen to another.

- The first two arguments specify the width (`w`) and height (`h`) of the rectangle. These arguments must be nonnegative.
- The source rectangle is located by the coordinates (`xs,ys`) of its top left corner.
- The destination rectangle is located by the coordinates (`xd,yd`) of its top left corner.

If a portion of the source rectangle lies in negative X or Y coordinate space, that portion is not copied; only the portion lying in positive XY space is moved. Only the portion of the destination rectangle lying within the current visibility rectangle (the window corresponding to the intersection of the viewport and clipping rectangle) is modified on the screen.

The rectangle is copied correctly even when the source and destination rectangles overlap.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long w, h, xs, ys, xd, yd, i;

init_video(1);
init_grafix();
init_screen();
w = 80;
h = 60;
xs = 280;
ys = 210;
/*******      Draw picture      *******/
fill_rect(w, h, xs, ys);
set_color0(0x11111111);      /* Assume 4 bits/pixel */
set_color1(0x44444444);
patnframe_oval(w, h, xs, ys, 20, 15);
/* Move picture to several places on screen */
for (i = 25, xd = 0, yd = -150<<16; i > 0; --i) {
    xd += yd >> 2;
    yd -= xd >> 2;
    move_rect(w, h, xs, ys, (xd>>16)+280, (yd>>16)+210);
}
```

Syntax

```
void move_vuport(xleft, ytop)
   int  xleft, ytop;    /* new position */
```

Description

The move_vuport function moves the viewport to a new position on the screen. The viewport is rectangular. Its position is adjusted so that its top left corner coincides with the coordinates `xleft` and `ytop`. These coordinates are expressed as displacements from the screen origin, located at the top left corner of the screen.

> **Note:**
>
> Before you call the move_vuport function, call the init_vuport function to initialize the viewport data structures.

Example

```
static char *s[] = {
       "Coordinate origin",
       "and clipping rectangle",
       "move with viewport."
       };
int i, xtext, ytext, xvu, yvu;

init_video(1);
init_grafix();
init_vuport();          /* Initialize viewport 0 */
init_text();
init_screen();
set_color0(0xCCCCCCCC);
for (xvu = yvu = 0; xvu < 447; xvu += 8, yvu += 6) {
    move_vuport(xvu, yvu);
    set_color1(0xCCCCCCCC);
    fill_rect(192, 76, 0, 0);
    set_color1(0x44444444);
    frame_rect(188, 72, 2, 2, 4, 4);
    set_color1(0x77777777);
    xtext = char_wide_max();
    ytext = 12 + get_ascent();
    for (i = 0; i <= 2; ++i) {
         draw_string(xtext, ytext, s[i]);
         ytext += char_high();
    }
}
```

Syntax

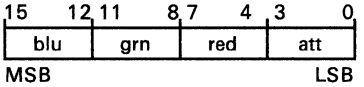
```
void new_screen(pixel, palet)
   long  pixel;
   short palet[16];
```

Description

The new–screen function clears screen (entire frame buffer) to a specified pixel value, and loads the color look-up table with an array of palette values. The implementation of this function is system dependent, and relies on features of the TMS34070 color palette and TMS4161 video RAM. Video RAM register-to-memory cycles are used to make this function execute rapidly.

The pixel value must be replicated throughout the 32 bits of the first argument, `pixel`. For example, at four bits per pixel, a pixel value of 9 is replicated as follows: 0x99999999.

The second argument, `palet`, is the array that contains the color palette. The array contains 16 elements, and each element is 16 bits. The fields within each element of the palet array are defined as follows:



Symbols red, grn, and blu represent 4-bit red, green, and blue intensity values, respectively. The rightmost field contains the color-repeat attribute bit. The attribute field is typically set to 0. Refer to the *TMS34070 Color Palette User's Guide* for a description of the color repeat attribute bit.

> **Note:**
>
> Before you call this function, call the init–grafix function to initialize the graphics environment.

Example

```
static short palet[16] = {        /* gray scale */
    0x0000, 0x1110, 0x2220, 0x3330,
    0x4440, 0x5550, 0x6660, 0x7770,
    0x8880, 0x9990, 0xAAA0, 0xBBB0,
    0xCCC0, 0xDDD0, 0xEEE0, 0xFFF0
};
long pixval, x;

init_video(1);
init_grafix();
pixval = 0x77777777;              /* 4 bits/pixel */
new_screen(pixval, palet);
for (x = 13, pixval = 0; x < 620; x += 39) {
     set_color1(pixval);
     fill_rect(29, 460, x, 10);
     set_color1(0);
     draw_rect(30, 461, x-1, 9);
     pixval += 0x11111111;
}
```

Syntax `int open_vuport()`

Description The open_vuport function opens a new viewport and returns an index that identifies the viewport in subsequent transactions. The new viewport automatically becomes the active viewport for subsequent drawing operations. The new viewport inherits the attributes of the viewport that was active at the time the call was initiated.

The new viewport automatically inherits the viewport attributes of the previous active viewport (that is, the viewport that was active just prior to the call to the open_vuport function). The inherited attributes include:

- Viewport size and position
- XY coordinate origin
- Clipping rectangle
- Font
- COLOR0 and COLOR1
- Pixel processing operation
- Plane mask
- Transparency attribute (on or off)
- Drawing pen width and height
- 16 x 16 pattern
- Incremental text spacing parameter (see add_text_space function)

If the maximum number of viewports are already opened at the time of the call, a value of -1 is returned in place of a valid viewport index.

> **Note:**
>
> Before you call the open_vuport function, call the init_vuport function to initialize the viewport data structures.

Example

```
#include "colors.h"
int vindex;

init_video(1);
init_grafix();
init_vuport();          /* Open vuport 0                */
init_screen();
set_color0(BLUE);       /* Viewport 0's color0 = BLUE */
vindex = open_vuport();/* Open vuport 1                 */
/***    Viewport 1 inherits color0 from viewport 0    ***/
set_color1(GRAY);       /* Viewport 1's color1 = GRAY */
/***     Fill square with blue-and-gray pattern       ***/
patnfill_rect(96, 96, 272, 192);
```

Syntax

```
int patnfill_convex(n, edgelist, ptlist)
   int   n;              /* number of polygon vertices */
   short edgelist[];     /* list of edges              */
   short ptlist[];       /* list of vertices (points)  */
```

Description

The patnfill—convex function fills a convex polygon with a pattern given a list of points representing the vertices. In order to be drawn correctly, the polygon must be convex; that is, it should contain no concavities. A polygon must have at least three vertices to be visible. An edge of the polygon is assumed between the first and last vertices specified. The polygon is pattern-filled with the current pattern, which is drawn in colors COLOR0 and COLOR1.

The function requires three input arguments:

- Argument `n` defines the number of vertices in the polygon.
- The second argument, `edgelist`, is an array of type short. The members of the array are indices that specify the order in which the vertices are traversed, moving in a clockwise direction around the edge of the polygon. (*Clockwise*, in this context, assumes X increasing from left to right and Y increasing from top to bottom.) Each element of the `edgelist` array is an index into the `ptlist` array.
- The third argument, `ptlist`, is an array of type short. Each pair of adjacent 16-bit elements contains the X and Y coordinates, respectively, of a vertex.

For example, `edgelist[k]` contains the index for vertex k, where k is in the range 0 to `n-1`. The X and Y coordinates for vertex k are contained in `ptlist[2*n]` and `ptlist[2*n+1]`.

The patnfill—convex function does automatic culling of back faces to support 3D applications. In other words, a polygon is drawn only if its front side is visible; that is, if it is facing toward the screen. If the vertices are supplied in counterclockwise order, the polygon is assumed to be facing away from the screen and is therefore not drawn. In this case, a value of 0 is returned by the function. Otherwise, a value of 1 is returned to indicate that the polygon is visible.

The back face test is done by first comparing vertices `n-2`, `n-1`, and 0 to determine whether the polygon vertices are specified in clockwise (front face) or counterclockwise (back face) order. This test relies on the polygon containing no concavities. If the three vertices are found to be colinear, the back face test is made again using the next three vertices, `n-1`, 0 and 1. The test repeats until three vertices are found that are not colinear. If all the vertices are colinear, the polygon is invisible and a value of 0 is returned.

This function is similar to the patnfill—polygon routine, but is specialized for rapid drawing of convex polygons. Note that the `edgelist` array format for the patnfill—convex function differs from the `linelist` array format for the patnfill—polygon function. While the patnfill—convex function is more specialized than the patnfill—polygon function, it also executes more rapidly.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long   i, hue, patn;
static short connect[] = { 0, 1, 2 };
static short xy[] = { 0,-170, 196,170, -196,170 };

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
init_screen();
for (i = 15, hue = patn = 0; i > 0; --i) {
     set_color0(hue += 0x11111111);
     set_color1(~hue);
     select_patn(patn++);
     patnfill_convex(3, connect, xy);
     xy[0] += xy[1] >> 3;
     xy[1] -= xy[0] >> 3;
     xy[2] += xy[3] >> 3;
     xy[3] -= xy[2] >> 3;
     xy[4] += xy[5] >> 3;
     xy[5] -= xy[4] >> 3;
}
```

Syntax

```
void patnfill_oval(w, h, xleft, ytop)
   int w, h;          /* width and height of enclosing */
                      /* rectangle                     */
   int xleft, ytop; /* XY coordinates of top left    */
                      /* corner                        */
```

Description

The patnfill_oval function fills an ellipse with a pattern. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes. The ellipse is filled with the current pattern in colors COLOR0 and COLOR1.

The ellipse is defined by the minimum enclosing rectangle in which it is inscribed. Four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, hue, patn;

init_video(1);
init_grafix();
init_screen();
/***      Pattern fill ellipses of various sizes       ***/
x = y = hue = patn = 0;
for (w = 640, h = 480; w >= 192; w -= 32, h -= 24) {
    set_color0(hue += 0x11111111);   /* pixel = 4 bits */
    set_color1(~hue);
    select_patn(patn++);
    patnfill_oval(w, h, x, y);
}
```

Syntax

```
void patnfill_piearc(w, h, xleft, ytop, theta, arc)
   int w, h,          /* width and height          */
   int xleft, ytop;   /* top left corner           */
   int theta;         /* starting angle (degrees)  */
   int arc;           /* extent of angle (degrees) */
```

Description

The patnfill—piearc function fills a pie-shaped wedge with a pattern. The wedge is bounded by an arc and two straight edges. The arc is taken from an ellipse in standard position, with its major and minor axes parallel to the coordinate axes. The two straight edges are defined by lines connecting the end points of the arc with the center of the ellipse. The arc is filled with the current pattern in colors COLOR0 and COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.
- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
/***************************/
/*      Draw a pie chart      */
/***************************/
static int tarc[] = { 50, 40, 35, 100, 80, 55 };
long   i, w, h, x, y, tstart, patn;
long   hue = 0xEEEEEEEE;  /* Assume 4 bits/pixel */

init_video(1);
init_grafix();
init_screen();
w      = 480;
h      = 360;
x      = 50;
y      = 60;
tstart = 25;
patn   = 0;
for (i = 0; i <= 5; ++i) {
    set_color1(hue -= 0x11111111);
    set_color0(~hue);
    select_patn(++patn);
    if (i == 5)
        x += w/8;
    patnfill_piearc(w, h, x, y, tstart, tarc[i]);
    tstart += tarc[i];
}
```

Syntax

```
void patnfill_polygon(n, linelist, ptlist)
   int n;                  /* number of edges              */
   short linelist[];       /* list of edges                */
   short ptlist[];         /* list of vertex coordinates */
```

Description

The patnfill—polygon function fills a polygon with a pattern given a list of lines representing the edges of the polygon. No restrictions are placed on the shape of the polygons filled by the function: edges can cross each other, filled areas can contain holes, and two or more filled regions can be disconnected from each other. The polygon is filled with the current pattern in colors COLOR0 and COLOR1.

The function requires three input arguments:

- Argument `n` defines the number of vertices in the polygon.
- The second argument, `linelist`, is an array of type short. Each pair of array elements defines an edge: the first of the two elements defines the starting vertex of the edge, and the second defines the ending vertex. Each element of the `linelist` array is an index into the `ptlist` array.
- The third argument, `ptlist`, is an array of type short. Each pair of adjacent 16-bit elements contains the X and Y coordinates, respectively, of a vertex.

Each pair of adjacent 16-bit elements in the `ptlist` array is an X coordinate followed by a Y coordinate. Each pair of adjacent 16-bit elements in the `linelist` array is a pair of indices into the `ptlist` array.

For example, the first edge drawn is specified in array elements, `linelist[0]` and `linelist[1]`. Assume that these contain index values 4 and 7, respectively. The starting coordinates for the line defining the edge are contained in `ptlist[2*4]` and `ptlist[2*4+1]`. The ending coordinates are contained in `ptlist[2*7]` and `ptlist[2*7+1]`.

The individual elements of the `linelist` array are assigned as follows:

```
linelist[0]    = starting vertex for edge 0
linelist[1]    = ending vertex for edge 0
linelist[2]    = starting vertex for edge 1
linelist[3]    = ending vertex for edge 1
      .
      .
      .
linelist[2n]   = starting vertex for edge n-1
linelist[2n+1] = ending vertex for edge n-1
```

The individual elements for the `ptlist` array are assigned as follows:

```
ptlist[0]      = x coordinate value for point 0
ptlist[1]      = y coordinate value for point 0
ptlist[2]      = x coordinate value for point 1
ptlist[3]      = y coordinate value for point 1
      .
      .
      .
ptlist[2m]     = x coordinate value for point m-1
ptlist[2m+1]   = y coordinate value for point m-1
```

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
static short xy[] = {
    193,60, 460,60, 377,440, 524,423,
    15,233, 570,100, 98,382
};
static short shape[] = {
    0,1, 1,2, 2,3, 3,0, 4,5, 5,6, 6,4
};
init_video(1);
init_grafix();
init_screen();
set_color0(0x44444444); /* pixel = 4 bits */
set_color1(0x77777777);
select_patn(4);
patnfill_polygon(7, shape, xy);
for (;;) ;
```

Syntax

```
void patnfill_rect(w, h, xleft, ytop)
   int w, h;          /* width and height of rectangle */
   int xleft, ytop;   /* XY coord at top left corner   */
```

Description

The patnfill_rect function fills a rectangle with a pattern. The rectangle is filled with the current pattern in colors COLOR0 and COLOR1. Four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
init_video(1);
init_grafix();
init_screen();
set_color1(0x11111111); /* Assume 4 bits/pixel */
select_patn(8);
patnfill_rect(440, 280, 100, 100);
set_color0(0x44444444);
set_color1(0x77777777);
select_patn(4);
patnfill_rect(420, 30, 110, 110);
patnfill_rect(220, 220, 110, 150);
patnfill_rect(190, 150, 340, 150);
patnfill_rect(190, 60, 340, 310);
```

Syntax

```
void patnframe_oval(w, h, xleft, ytop, dx, dy)
   int w, h;          /* width and height of enclosing   */
                      /* rectangle                       */
   int xleft, ytop; /* coordinates at top left corner  */
   int dx, dy;        /* width and height of frame border*/
```

Description

The patnframe_oval function fills an ellipse-shaped frame with a pattern. The frame consists of a filled region between two concentric ellipses. The frame is filled with the current pattern in colors COLOR0 and COLOR1. The portion of the screen enclosed by the frame is not altered.

The outer ellipse is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The thickness of the frame in the X and Y dimensions is defined by two additional arguments, `dx` and `dy`, which specify the horizontal and vertical distances, respectively, between the outer and inner ellipses.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, dx, dy;

init_video(1);
init_grafix();
init_screen();
w  = 480;
h  = 360;
x  = 80;
y  = 60;
dx = 40;
dy = 30;
set_color0(0x11111111);
set_color1(0x77777777);
select_patn(7);
patnframe_oval(w, h, x, y, dx, dy);
```

Syntax

```
void patnframe_rect(w, h, xleft, ytop, dx, dy)
   int w, h;          /* width and height of enclosing   */
                      /* rectangle                       */
   int xleft, ytop; /* coordinates at top left corner  */
   int dx, dy;        /* width and height of frame border*/
```

Description

The patnframe_rect function fills a rectangular frame with a pattern. The frame consists of a filled region between two concentric rectangles. The frame is filled with the current pattern in colors COLOR0 and COLOR1. The portion of the screen enclosed by inner edge of the frame is not altered.

The first four arguments define the outer rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The thickness of the frame in the X and Y dimensions is defined by two additional arguments, `dx` and `dy`, which specify the horizontal and vertical distances, respectively, between the outer and inner rectangles.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, dx, dy;

init_video(1);
init_grafix();
init_screen();
w  = 480;
h  = 360;
x  = 80;
y  = 60;
dx = 40;
dy = 30;
set_color0(0x11111111);
set_color1(0x77777777);
select_patn(7);
patnframe_rect(w, h, x, y, dx, dy);
```

Syntax

```
void patnpen_line(x1, y1, x2, y2)
   int  x1, y1;      /* starting coordinates */
   int  x2, y2;      /* ending coordinates   */
```

Description

The patnpen_line function uses the pen to draw a patterned line. Arguments `x1` and `y1` specify the starting coordinates of the line, and `x2` and `y2` specify the ending coordinates.

The pen is a rectangle whose width and height can be modified by means of the set_pensize function. At each point on the line drawn by the patnpen_line function, the pen is located such that its top left corner touches the line. The area covered by the pen is filled with the current pattern in colors COLOR0 and COLOR1.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long x, y, patn, hue;

init_video(1);
init_grafix();
init_screen();
set_pensize(20, 16);
patn = hue = 0;
for (x = 8, y = 455; x < 631; x += 43) {
    set_color0(~hue);
    set_color1(hue += 0x11111111); /* pixel = 4 bits */
    select_patn(patn++);
    patnpen_line(8, 8, x, y);
    patnpen_line(610, 8, x, y);
}
```

Syntax

```
void patnpen_ovalarc(w, h, xleft, ytop, theta, arc)
   int w, h,            /* width and height          */
   int xleft, ytop;     /* top left corner           */
   int theta;           /* starting angle (degrees) */
   int arc;             /* angle extent (degrees)    */
```

Description

The patnpen—ovalarc function uses the pen to draw a patterned arc of an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on the arc drawn by the patnpen—line function, the pen is positioned such that its top left corner touches the arc. The area covered by the pen is filled with the current pattern in colors COLOR0 and COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.

- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, tstart, tarc;
long hue, patn;

init_video(1);
init_grafix();
init_screen();
set_pensize(32, 24);
x = 320;
y = 240;
w = h = 0;
tarc = 35;
for (tstart = 0; tstart < 1500; tstart += 30) {
    set_color0(~hue);
    if ((hue -= 0x11111111) == 0)
        hue = 0xFFFFFFFF;
    set_color1(hue);
    if (++patn >= get_patn_max())
        patn = 0;
    select_patn(patn);
    patnpen_ovalarc(w, h, x, y, tstart, tarc);
    w += 16;
    h += 12;
    x -= 8;
    y -= 6;
}
```

Syntax

```
void patnpen_piearc(w, h, xleft, ytop, theta, arc)
   int w, h,              /* width and height          */
   int xleft, ytop;       /* top left corner           */
   int theta;             /* starting angle (degrees) */
   int arc;               /* angle extent (degrees)   */
```

Description

The patnpen—piearc function uses the pen to draw a patterned, pie-shaped wedge from an ellipse. The wedge is formed by an arc of the ellipse, and by two straight lines that connect the two end points of the arc with the center of the ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on the arc drawn by the patnpen—line function, the pen is positioned such that its top left corner touches the arc. The two lines from the center are drawn in similar fashion. The area covered by the pen is filled with the current pattern in colors COLOR0 and COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.

- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static t[] = {
    110,100, 30,80, 15,15, 210,90, 300,30, 330,45
};
long i, w, h, x, y, patn, hue;

init_video(1);
init_grafix();
init_screen();
w = 280;
h = 440;
x = 105;
y = 20;
patn = 16;
set_pensize(80, 1); /* long, skinny pen */
for (i = hue = 0; i <= 5; ++i) {
     if (i == 5)
         x += w/4;
     select_patn(patn++);
     set_color0(~hue);
     set_color1(hue += 0x11111111); /* 4-bit pixel */
     patnpen_piearc(w, h, x, y, t[2*i], t[2*i+1]);
}
for (i = hue = 0, x -= w/4; i <= 5; ++i) {
     if (i == 5)
         x += w/4;
     set_color1(hue += 0x11111111);
     fill_piearc(w, h, x, y, t[2*i], t[2*i+1]);
}
```

Syntax

```
void patnpen_point(x, y)
   int  x, y;          /* pen coordinates */
```

Description

The patnpen—point function uses the pen to draw a patterned point. The resulting figure is a rectangle the width and height of the pen, and filled with the current pattern in colors COLOR0 and COLOR1. The top left corner of the rectangle is located at coordinates (x, y).

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int i, w, h, x, y, patn;
long hue;                    /* Assume 4 bits/pixel */

init_video(1);
init_grafix();
init_screen();
w = 12;
h = 9;
x = 16 << 16;
y = 12 << 16;
patn = 0;
hue = 0xFFFFFFFF;
for (i = 100; i > 0; --i) {
    set_color0(~hue);
    if ((hue -= 0x11111111) == 0)
        hue = 0xFFFFFFFF;
    set_color1(hue);
    set_pensize(++w, ++h);
    if (++patn == get_patn_max())
        patn = 0;
    select_patn(patn);
    patnpen_point((x>>16)+350, (y>>16)+200);
    x += y >> 3;
    y -= x >> 3;
    x += x >> 5;
    y += y >> 5;
}
```

Syntax

```
void patnpen_polyline(n, linelist, ptlist)
   int   n;                  /* number of lines */
   short linelist[];         /* list of lines   */
   short ptlist[];           /* list of points  */
```

Description

The patnpen—polyline function uses the pen to draw multiple patterned lines.

- `n` specifies the number of lines that are drawn.
- `linelist` is an array of type short; it specifies the list of lines that are drawn. Each element in the `linelist` array is an index into the `ptlist` array.
- The third argument, the `ptlist` array, contains the XY coordinates of the starting and ending points for each line.

Each pair of adjacent 16-bit elements in the `ptlist` array is an X coordinate followed by a Y coordinate. Each pair of adjacent 16-bit elements in the `linelist` array is a pair of indices into the `ptlist` array. all the line starting and ending points.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on a line drawn by the patnpen—line function, the pen is located such that its top left corner touches the line. The area covered by the pen is filled with the current pattern in colors COLOR0 and COLOR1.

The individual elements of the `linelist` array are assigned as follows:

```
linelist[0]     = starting point of line 0
linelist[1]     = ending point of line 0
linelist[2]     = starting point of line 1
linelist[3]     = ending point of line 1
      .
      .
      .
linelist[2n]    = starting point of line n-1
linelist[2n+1]  = ending point of line n-1
```

The individual elements of the `ptlist` array are assigned as follows:

```
ptlist[0]       = x coordinate value for point 0
ptlist[1]       = y coordinate value for point 0
ptlist[2]       = x coordinate value for point 1
ptlist[3]       = y coordinate value for point 1
      .
      .
      .
ptlist[2m]      = x coordinate value for point m-1
ptlist[2m+1]    = y coordinate value for point m-1
```

For example, the first line drawn is specified in the first two elements, `linelist[0]` and `linelist[1]`. Assume that these contain index values 4 and 7, respectively. The starting coordinates for the line are contained in `ptlist[2*4]` and `ptlist[2*4+1]`. The ending coordinates are contained in `ptlist[2*7]` and `ptlist[2*7+1]`.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short xy[] = {
    380,200, 480,200, 480,300, 380,300,
    340,270, 340,170, 440,170,
    230,180, 280,300, 160,300, 146,263
};
static short cube[] = {
    0,1, 1,2, 2,3, 3,4, 4,5, 5,6, 6,1, 3,0, 5,0,
};
static short pyramid[] = {
    7,8, 8,9, 9,10, 10,7, 7,9
};

/*** Draw a cube and a pyramid sitting side by side ***/
init_video(1);
init_grafix();
init_screen();
set_pensize(9, 8);
select_patn(21);
patnpen_polyline(9, cube, xy);
set_color0(0x11111111);  /* 4-bit pixel */
set_color1(0x33333333);
set_pensize(7, 6);
select_patn(15);
patnpen_polyline(5, pyramid, xy);
```

Syntax

```
int peek(address)
   long address;     /* 32-bit memory address */
```

Description

The peek function returns the contents of a memory location. The value of the memory word is returned in the 16 LSBs of the 32-bit return value. The 16 MSBs are 0s.

Example

```
#define VCLK    160          /* video clock period (ns) */
#define HTOTAL 0xC0000030
#define VTOTAL 0xC0000070
int tick;

init_video(1);
/***  Calculate frame duration in ns  ***/
tick = (peek(HTOTAL) + 1) * (peek(VTOTAL) + 1) * VCLK;
```

Syntax

```
long peek_breg(breg)
   int breg;       /* B-file register number */
```

Description

The peek_breg function returns the contents of a B-file register. Argument `breg` is a register number in the range 0 to 15. The function ignores all but the 4 LSBs of `breg`. The return value is 32 bits.

Example

```
#define DPTCH  3          /* B3 register */
long dest_pitch;

init_video(1);
/***   Read screen pitch  ***/
dest_pitch = peek_breg(3);
```

Syntax

```
void pen_line(x1, y1, x2, y2)
   int  x1, y1;       /* starting coordinates */
   int  x2, y2;       /* ending coordinates   */
```

Description

The pen_line function uses the pen to draw a line. Arguments `x1` and `y1` specify the starting coordinates of the line, and `x2` and `y2` specify the ending coordinates.

The pen is a rectangle whose width and height can be modified by means of the set_pensize function. At each point on the line drawn by the pen_line function, the pen is located such that its top left corner touches the line. The area covered by the pen is solid-filled in the current COLOR1.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long color;            /* Assume 4 bits/pixel */
int  x1, y1, x2, y2;

init_video(1);
init_grafix();
patnfill_rect(640, 480, 0, 0);
init_palet();
set_pensize(20, 16);
color = 0;
x1 = y1 = 8;
for (x2 = 8, y2 = 455; x2 < 631; x2 += 43) {
    set_color1(color += 0x11111111);
    pen_line(x1, y1, x2, y2);
}
```

Syntax

```
void pen_ovalarc(w, h, xleft, ytop, theta, arc)
  int w, h,           /* width and height          */
  int xleft, ytop;    /* top left corner           */
  int theta;          /* starting angle (degrees) */
  int arc;            /* angle extent (degrees)   */
```

Description

The pen—ovalarc function uses the pen to draw an arc taken from an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on the arc drawn by the pen—line function, the pen is located such that its top left corner touches the arc. The area covered by the pen is solid-filled in the current COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.

- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, tstart, tarc;

init_video(1);
init_grafix();
init_screen();
set_pensize(4, 3);
x = 320;
y = 240;
w = h = 0;
tarc = 35;
for (tstart = 0; tstart < 1500; tstart += 30) {
    pen_ovalarc(w, h, x, y, tstart, tarc);
    w += 16;
    h += 12;
    x -= 8;
    y -= 6;
}
```

Syntax

```
void pen_piearc(w, h, xleft, ytop, theta, arc)
   int w, h,            /* width and height          */
   int xleft, ytop;     /* top left corner           */
   int theta;           /* starting angle (degrees) */
   int arc;             /* angle extent (degrees)    */
```

Description

The pen—piearc function uses the pen to draw a pie-shaped wedge from an ellipse. The wedge is formed by an arc of the ellipse, and by two straight lines that connect the two end points of the arc with the center of the ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on the arc drawn by the pen—line function, the pen is located such that its top left corner touches the arc. The two lines are drawn in similar fashion. The area covered by the pen is solid-filled in the current COLOR1.

The ellipse from which the arc is taken is specified in terms of the minimum enclosing rectangle in which it is inscribed. The first four arguments define the rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments define the limits of the arc:

- The starting angle, `theta`, is measured from the center of the right side of the enclosing rectangle, and is treated as modulus 360. Positive angles are in the clockwise direction, negative angles are counterclockwise.
- The arc extent, `arc`, specifies the number of degrees (positive or negative) spanned by the angle. If the arc extent is outside the range [-359,+359], the entire ellipse is drawn.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, t, dt, dx;

init_video(1);
init_grafix();
t = dx = dt = 8;
w = h = 80;
for (x = -w, y = 350; x < 640; x += dx) {
     if ((t += dt) > 80 || t <= 0)
          dt = -dt;
     delay(0);
     init_screen();
     pen_piearc(w, h, x, y, t/2, 360-t);
     pen_piearc(w, h, x+w/2, y, -15, 30);
}
```

Syntax

```
void pen_point(x, y)
   int  x, y;  /* pen coordinates */
```

Description

The pen_point function uses the pen to draw a point. The resulting figure is a rectangle the width and height of the pen, and solid-filled with the current COLOR1. The top left corner of the rectangle is located at coordinates (x, y).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int  i, w, h, x, y;
long c;              /* Assume 4 bits/pixel */

init_video(1);
init_grafix();
init_screen();
w = 12;
h = 9;
x = 16 << 16;
y = 12 << 16;
c = 0xFFFFFFFF;
for (i = 100; i > 0; --i) {
    if ((c -= 0x11111111) == 0)
        c = 0xFFFFFFFF;
    set_color1(c);
    set_pensize(++w, ++h);
    pen_point((x>>16)+350, (y>>16)+200);
    x += y >> 3;
    y -= x >> 3;
    x += x >> 5;
    y += y >> 5;
}
```

Syntax

```
void pen_polyline(n, linelist, ptlist)
   int   n;                /* number of lines */
   short linelist[];       /* list of lines   */
   short ptlist[];         /* list of points  */
```

Description

The pen—polyline function uses the pen to draw multiple lines.

- `n` specifies the number of lines that are drawn.
- `linelist` is an array of type short; it specifies the list of lines that are drawn. Each element in the `linelist` array is an index into the `ptlist` array.
- The third argument, the `ptlist` array, contains the XY coordinates of the starting and ending points for each line.

Each pair of adjacent 16-bit elements in the `ptlist` array is an X coordinate followed by a Y coordinate. Each pair of adjacent 16-bit elements in the `linelist` array is a pair of indices into the `ptlist` array.

The pen is a rectangle whose width and height can be modified by means of the set—pensize function. At each point on a line drawn by the pen—line function, the pen is located such that its top left corner touches the line. The area covered by the pen is solid-filled in the current COLOR1.

For example, the first line drawn is specified in the first two elements, `linelist[0]` and `linelist[1]`. Assume that these contain index values 4 and 7, respectively. The starting coordinates for the line are contained in `ptlist[2*4]` and `ptlist[2*4+1]`. The ending coordinates are contained in `ptlist[2*7]` and `ptlist[2*7+1]`.

The individual elements of the `linelist` array are assigned as follows:

```
linelist[0]    = starting point of line 0
linelist[1]    = ending point of line 0
linelist[2]    = starting point of line 1
linelist[3]    = ending point of line 1
      .
      .
      .
linelist[2n]   = starting point of line n-1
linelist[2n+1] = ending point of line n-1
```

The individual elements of the `ptlist` array are assigned as follows:

```
ptlist[0]      = x coordinate value for point 0
ptlist[1]      = y coordinate value for point 0
ptlist[2]      = x coordinate value for point 1
ptlist[3]      = y coordinate value for point 1
      .
      .
      .
ptlist[2m]     = x coordinate value for point m-1
ptlist[2m+1]   = y coordinate value for point m-1
```

Note:

Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short xy[] = {
    380,200, 480,200, 480,300, 380,300,
    340,270, 340,170, 440,170,
    230,180, 280,300, 160,300, 146,263
};
static short cube[] = {
    0,1, 1,2, 2,3, 3,4, 4,5, 5,6, 6,1, 3,0, 5,0,
};
static short pyramid[] = {
    7,8, 8,9, 9,10, 10,7, 7,9
};

/*************************************************/
/* Draw a cube and a pyramid sitting side by side */
/*************************************************/
init_video(1);
init_grafix();
init_screen();
set_pensize(5, 5);
pen_polyline(9, cube, xy);
set_color1(0x11111111); /* Assume 4 bits/pixel */
set_pensize(7, 7);
pen_polyline(5, pyramid, xy);
```

Syntax

```
void perspec(n, vertlist, ptlist, xview, yview, zview)
   typedef long FIX;              /* 32-bit fixed-point format  */
   int   n;                       /* number of vertices in list */
   FIX   vertlist[];              /* list of 3D vertices in xyz */
   short ptlist[];                /* list of 2D points in xy    */
   int   xview, yview, zview;     /* viewer's xyz coordinates   */
```

Description

The perspec function performs perspective transformations on an input list of 3D vertices, mapping them to an output list of 2D points.

- The first argument, `n`, specifies the number of vertices that are transformed.
- The second argument, `vertlist`, is a list of three-dimensional vertices. Each vertex in the list is represented as a 96-bit quantity consisting of three 32-bit X, Y, and Z fixed-point coordinate values. The fixed-point format assumes that the 16 LSBs lie to the right of the binary point.
- The third argument, `ptlist`, is a list of two-dimensional points. Each point in the list is represented as a 32-bit quantity consisting to two 16-bit X and Y coordinate values.
- The last three arguments, `xview`, `yview`, and `zview`, are the coordinates of the viewer's position.

The positive Z direction is into the screen. The face of the screen is at Z=0. The viewer's position is typically in negative Z space. If a vertex to be transformed is at the same Z value as the viewer or behind (that is, more negative Z) the viewer, the results are unpredictable.

The perspec function scales the X and Y displacements of each three-dimensional point. The displacements are measured from the viewer's X and Y coordinates, and are scaled in inverse proportion to the point's distance in Z from the viewer. The X and Y displacements become smaller as the distance from the viewer increases. The result is a two-dimensional array of points representing 3D objects which are scaled in accordance to their distance from the viewer. Please refer to *Principles of Interactive Graphics* (Newman and Sproull) for more information on perspective transformations.

The figure on the next page shows the perspective scaling of a point in three-dimensional space. For simplicity, only the scaling of the Y coordinate is shown. The face of the screen is at Z = 0, and is in the plane containing the X and Y axes. The perspective calculations are performed assuming the viewer is at coordinates (X_{view},Y_{view},Z_{view}), where Z_{view} is negative. The 3D point is located behind the screen at coordinates (Z_{object},Y_{object},Z_{object}), where Z is positive. The projection of the 3D point on the face of the screen is obtained by simple linear interpolation:

$$Y_{screen} = \frac{(Y_{object} - Y_{view}) \times (Z_{screen} - Y_{view})}{Z_{object} - Z_{view}} \times Y_{view}$$

where:

Y_{screen} = projection of 3D point on screen

Y_{view},Z_{view} = viewer's Y and Z coordinates

Y_{object},Z_{object} = 3D point's Y and Z coordinates

The value Y_{screen} is the scaled version of Y_{object} returned by the *perspec* function. An X_{screen} value is calculated in similar fashion.

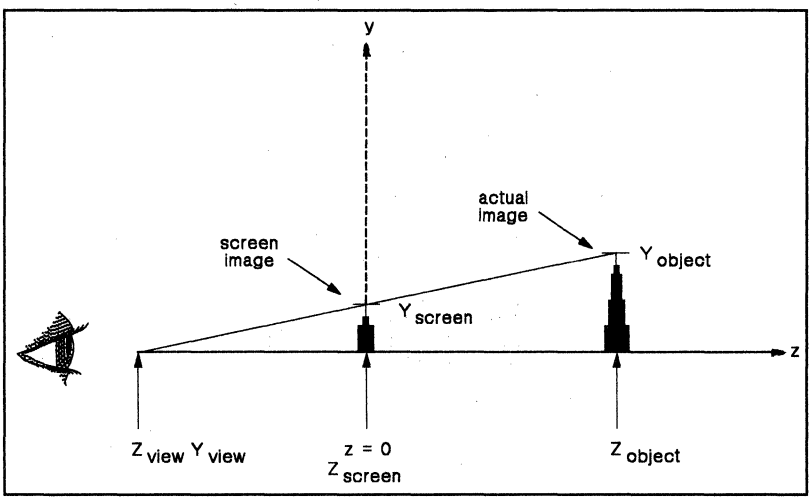


Figure 5-1. Perspective Transformation

Example

```
typedef long FIX;
static  FIX  rotation[3]  =  { 0, 0, 0 };
static long xyz[] = {
    -60,-60,-60,   60,-60,-60,
     60, 60,-60,  -60, 60,-60,
    -60,-60, 60,   60,-60, 60,
     60, 60, 60,  -60, 60, 60
};
static short faces[6][4] = {
    { 0, 1, 2, 3 }, { 7, 6, 5, 4 },
    { 4, 5, 1, 0 }, { 5, 6, 2, 1 },
    { 6, 7, 3, 2 }, { 4, 0, 3, 7 }
};
FIX   matrix[16], verts[24];
short xy[16];
long  i, c, angle;

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240); /* center origin */
for (;;)
    for (angle = 0; angle < 360; angle += 4) {
        init_matrix(matrix);
        rotation[2] = angle << 16;
        rotate(matrix, rotation);
        long_to_fix(24, xyz, verts);
        transform(matrix, 8, verts);
        perspec(8, verts, xy, 0, -80, -200);
        delay(0);
        init_screen();
        for (i = c = 0; i <= 5; ++i) {
            set_color1(c += 0x11111111); /*pixel=4 bits*/
            fill_convex(4, faces[i], xy);
        }
    }
```

Syntax

```
void poke(address, value)
   long address;      /* 32-bit memory address */
   int  value;        /* value to be poked     */
```

Description The poke function stores the 16 LSBs of `value` at location `address`.

Example

```
init_video(1);
/*************************/
/* Change pixel size to 1 */
/*************************/
poke(0xC0000150, 1);
```

Syntax

```
void poke_breg(breg, value)
   long breg;      /* B-file register number    */
   int  value;     /* 32-bit register contents */
```

Description

The poke_breg function stores the 32-bit value in a B-file register. Argument `breg` is a register number from 0 to 15.

Example

```
init_video(1);
/*************************/
/* Change OFFSET register */
/*************************/
poke_breg(4, 512*4);
```

Syntax

```
double pow(x, y)
   double  x, y;  /* Raise x to power y */
```

Description

The pow function calculates x raised to the power y (x^y). The two arguments and the return value are all double-precision floating-point values. The value returned is 0 if both x and y are 0.

Several error conditions are detected:

- If x < 0, the return value is $(-x)^y$, and the _fp_error function is called with an error code of 24.
- If x = 0 and y <= 0, the return value is $-\infty$, and _fp_error is called with an error code of 25.
- If arithmetic overflow occurs, the return value is $+\infty$, and _fp_error is called with an error code of 27.
- If arithmetic underflow occurs, the return value is 0, and _fp_error is called with an error code of 28 (see the *TMS34010 C Compiler User's Guide* for a description of the _fp_error function).

Example

```
extern double pow();
double x, y, z;

x = 2.0;
y = 3.0;
z = pow(x, y);  /* return value = 8.0 */
```

Syntax

```
void put_pixel(val, x, y)
   int val;       /* pixel value          */
   int x, y;      /* coordinates of pixel */
```

Description

The put_pixel function writes a value to a pixel on the screen. Argument `val` is the value that is written to the pixel located at coordinates (x,y). Given a pixel size of *n* bits, the pixel is contained in the *n* LSBs of `val` (higher order bits are ignored).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
static char *s[] = {
    "Flip all", "the pixels", "in this box."
};
int i, j, val, w, x1, y1, x2, y2;

init_video(1);
init_grafix();
init_text();
init_screen();
/***   Draw picture to be flipped   ***/
y1 = 98;
for (i = 0; i <= 2; ++i, y1 += char_high())
     draw_string(126, y1, s[i]);
select_patn(16);
w = 114;
x1 = 114;
y1 = 53;
patnframe_rect(w, w, x1, y1, 5, 14);
/***   Now use put_pixel function to flip pixels   ***/
x2 = x1 + 320;
y2 = y1 + 240;
for (i = 0; i <= 113; ++i)
    for (j = 0; j <= 113; ++j) {
        val = get_pixel(x1+i, y1+j);
        put_pixel(val, x2+w-j, y1+i);
        put_pixel(val, x2+w-i, y2+w-j);
        put_pixel(val, x1+w-j, y2+w-i);
    }
```

Syntax

```
void put_rect(sarray, spitch, w, h, xleft, ytop)
   short sarray[];     /* source pixel array            */
   long  spitch;       /* source pitch                  */
   int   w, h;         /* destination width and height */
   int   xleft, ytop; /* destination top left corner  */
```

Description

The put—rect function copies pixels from a packed pixel array to a rectangular area on the screen.

- The last four arguments define the destination rectangle:
  - The width `w`,
  - The height `h`, **and**
  - The coordinates of the top left corner (`xleft,ytop`).

  `w` and `h` must be nonnegative.
- Argument `sarray` is a two-dimensional array of pixels.
- Successive rows of the source array do not necessarily occupy contiguous locations in memory. The source pitch parameter, `spitch`, specifies the difference in memory addresses between adjacent rows of the source array. Argument `spitch` must be greater than or equal to width `w` times the pixel size.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, pitch, i;
short buf[80*60*4 / 16];

init_video(1);
init_grafix();
init_screen();
w = 80;
h = 60;
x = 280;
y = 210;
pitch = w * get_psize();
/*******************************/
/*          Draw picture       */
/*******************************/
fill_rect(w-2, h-2, x+1, y+1);
set_color0(0x11111111);     /* Assume 4 bits/pixel */
set_color1(0x44444444);
patnframe_oval(w-2, h-2, x+1, y+1, 20, 15);
/*******************************/
/* Capture picture from screen */
/*******************************/
get_rect(w, h, x, y, buf, pitch);

/*******************************/
/*   Put picture onto screen   */
/*******************************/
for (i = 25, x = 0, y = -150<<16; i > 0; --i) {
    x += y >> 2;
    y -= x >> 2;
    put_rect(buf, pitch, w, h, (x>>16)+280, (y>>16)+210);
}
```

Syntax

```
long rep_pixel(val)
   int val;        /* pixel value */
```

Description

The rep—pixel function replicates a pixel value val throughout a 32-bit integer. Given a pixel size of *n* bits, the *n* LSBs of `val` are replicated 32/*n* times to fill the 32-bit return value. (The higher order bits of val are ignored by the function.)

For example, given a pixel size of 4 bits and an input value of 5, the return value is 0x55555555. An input value of 0x1234567 produces a return value of 0x77777777, and so on.

The output of this function can be used as input for the set—color0, set—color1, and set—pmask functions.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int w, h, x, y, val;
static char s[] = "big";

init_video(1);
init_grafix();
init_text();
init_screen();
w = get_width(s);
h = get_ascent() + get_descent();
set_color1(rep_pixel(4));
fill_rect(w, h, 0, 0);
set_color0(rep_pixel(4));
set_color1(rep_pixel(7));
draw_string(0, get_ascent(), s);
for (x = 0; x < w; ++x)
    for (y = 0; y < h; ++y)  {
        val = rep_pixel(get_pixel(x,y));
        set_color1(val);
        fill_rect(18, 18, 80+20*x, 80+20*y);
    }
```

Syntax

```
int rmo(n)
   long n;       /* 32-bit integer */
```

Description

The rmo function calculates the bit number of the rightmost one in argument n. The argument is treated as a 32-bit number whose bits are numbered from 0 to 31, where bit 0 is the LSB (the rightmost bit position) and bit 31 is the MSB (the leftmost bit position).

For nonzero arguments, the return value is in the range 0 to 31. If the argument is 0, a value of -1 is returned.

Example

```
int i;

init_video(1);
init_grafix();
init_screen();
for (i = 1; i < 640; ++i)
    draw_line(0, i, 2 << rmo(i), i);
```

Syntax

```
void rotate(matrix, angle)
   typedef long FIX;   /* define fixed-point type          */
   FIX matrix[16];     /* 4×4 transformation matrix        */
   FIX angle[3];       /* angles of rotation in radians */
```

Description

The rotate function applies rotations in the XY, YZ, and ZX planes to a 4×4 homogeneous transformation matrix. Once the rotation information is embedded in the matrix, the matrix can be used to transform the X, Y, and Z coordinates that represent the position of a three-dimensional object.

- Array `matrix` is a 16-element transformation matrix in row-major order.
- Argument `angle` is a 3-element array that contains the angles of rotation in the three planes. Specifically, elements `angle[0]`, `angle[1]` and `angle[2]` contain the angles for the XY, YZ, and ZX planes, respectively. The rotation angles are applied to the matrix in the order XY first, YZ second, and ZX third. The angles are expressed in degrees. Array elements are 32-bit fixed-point numbers whose 16 LSBs lie to the right of the binary point.

The rotate function multiplies the input matrix by the following three rotation matrices:

- Rotation in the XY plane (about the Z axis). Let ∠XY = `angle[0]`:

$$\begin{bmatrix} \cos(\angle XY) & -\sin(\angle XY) & 0 & 0 \\ \sin(\angle XY) & \cos(\angle XY) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation in the YZ plane (about the X axis). Let ∠YZ = `angle[1]`:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\angle YZ) & -\sin(\angle YZ) & 0 \\ 0 & \sin(\angle YZ) & \cos(\angle YZ) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation in the ZX plane (about the Y axis). Let ∠ZX = `angle[2]`.

$$\begin{bmatrix} \cos(\angle ZX) & 0 & \sin(\angle ZX) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\angle ZX) & 0 & \cos(\angle ZX) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The function is designed to trivially detect rotation angles of 0 degrees in order to avoid unnecessary computations. See *Principles of Interactive Graphics* (Newman and Sproull) for more information on homogeneous transformations.

Example

```
typedef long FIX;
static  FIX  rot[3] = { 0, 0, 0 };
static  long xyz[] = {
    -50,-50,0, 50,-50,0, 50,50,0, -50,50,0
};
static short front[4] = { 0,1,2,3 };
static short back[4] = { 3,2,1,0 };
FIX    matrix[16], verts[12];
short  xy[8];
int    i, angle;

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);     /* Center origin */
/*********************************************/
/* Rotate the square in each of the 3 planes */
/*********************************************/
for (i = 0; i <= 2; ++i)
    for (angle = 0; angle <= 360; ++angle) {
        init_matrix(matrix);
        rot[i] = angle << 16;
        rotate(matrix, rot);
        long_to_fix(12, xyz, verts);
        transform(matrix, 4, verts);
        perspec(4, verts, xy, 0, -100, -300);
        delay(0);
        init_screen();
        set_color1(0x11111111); /* 4-bit pixel */
        fill_convex(4, front, xy);
        set_color1(0x66666666);
        fill_convex(4, back, xy);
    }
```

Syntax

```
void run_decode(xleft, ytop, image)
   int   xleft, ytop;  /* screen rectangle coords */
   short image[];      /* compressed image        */
```

Description

The run—decode function decodes an image previously encoded by the run—encode function. The decoded image is drawn to the designated area of the screen.

The image is rectangular; its top left corner is positioned at coordinates (`xleft,ytop`). The image produced by the run—decode function has the same dimensions as the original image captured by the run—encode function. The width and height are embedded in the `image` array along with the compressed image.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
#define MAXSIZE   4000
char picture[MAXSIZE];
int  x, y;

init_video(1);
init_grafix();
init_screen();
/*************************************/
/*          Draw a picture           */
/*************************************/
set_color1(rep_pixel(1));
frame_rect(98, 78, 1, 1, 8, 8);
set_color1(rep_pixel(9));
fill_rect(84, 64, 8, 8);
set_color0(rep_pixel(4));
set_color1(rep_pixel(7));
patnframe_oval(92, 72, 4, 4, 26, 20);
/*************************************/
/*         Capture the picture       */
/*************************************/
run_encode(100, 80, 0, 0, picture, MAXSIZE);

/*************************************/
/* Draw the picture back to the screen */
/*************************************/
for (x = 0, y = 394; y > 0; x += 77, y -= 56)
    run_decode(x, y, picture);
```

Syntax

```
int run_encode(w, h, xleft, ytop, image, maxbytes)
   int   w, h ;          /* source width and height       */
   int   xleft, ytop;    /* screen rectangle coordinates */
   short image[];        /* compressed image              */
   int   maxbytes;       /* array capacity in bytes       */
```

Description

The run_encode function compresses an image using run-length encoding. It saves the image contained in a rectangular area of the screen. The image is stored in the `image` array.

Run-length encoding stores each horizontal line of the image as a series of color transitions: the color for each transition is paired with the number of times the color is repeated (that is, the number of pixels in that color) before a transition to a new color occurs.

Once an image is encoded using the run_encode function, it can be decoded and drawn to the screen using the run_decode function.

The first four arguments specify a rectangular area of the screen that is compressed:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The last two arguments specify the destination array for the compressed image data.

- Argument `image` is an array large enough to contain the compressed image.
- Argument `maxbytes` is the number of 8-bit bytes available in the array for storing the compressed image.

The value returned by the function is the number of 8-bit bytes actually required to store the compressed image. If the return value is larger than `maxbytes`, the function ceases writing data to the `image` array following the point at which it runs out of room in the array.

Table 5-1 shows the format for the `image` array.

Table 5-1. Image Array Format

Byte	Information
0–1	Image format identifier
2–4	Length of array in bytes
5–6	Width of image rectangle
7–8	Height of image rectangle
9–10	X coordinate at left side of rectangle
11–12	Y coordinate at top of rectangle
13...	Run-length encoded image data

The first 14 bytes contain header information. The initial word is a format identifier that specifies the type of encoding used to compress the image. The next four bytes specify the number of bytes actually used to encode the image (including the 14 header bytes). The next four bytes specify the width (two bytes) and height (two bytes) of the original rectangular region of the screen from which the image is taken. The next four bytes contain the x (two bytes) and the y (two bytes) coordinate values (relative to the active viewport at the time the image was encoded) at the top left corner of the screen rectangle.

Following the header information is the actual run-length-encoded image. Each byte of the encoded image represents either a run-length code (*rlcode*) or a pixel value (*pvalue*). A positive rlcode means that the byte following the rlcode is a pvalue representing the pixel value for the next rlcode pixels. For example, if an rlcode value of +10 is followed by a pvalue of 0x55, this represents a string of 10 pixels, each of color 0x55. A negative rlcode means that the next abs(rlcode) pixels are specified as individual bytes. For example, an rlcode value of -4 followed by 0x11, 0x22, 0x33, and 0x44 means that the next four pixels are of colors 0x11, 0x22, 0x33, and 0x44, respectively. the first byte in the image array (following the header) is always an rlcode.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example Refer to the example in the run—decode definition.

Syntax

```
void scale(matrix, factor)
   typedef long FIX;   /* fixed-point type definition */
   FIX matrix[16];     /* 4x4 transformation matrix   */
   FIX factor[3];      /* scaling factors in x, y, z  */
```

Description

The scale function applies scaling transformations in the X, Y, and Z dimensions to a 4 x 4 homogeneous transformation matrix. Once the scaling information is embedded in the matrix, the matrix can be used to transform the X, Y, and Z coordinates representing the position of a three-dimensional object.

- Array `matrix` is a 16-element transformation matrix specified in row-major order.
- Argument `factor` is a 3-element array containing the specified scaling factors for the X, Y, and Z dimensions. Specifically, elements `factor[0]`, `factor[1]` and `factor[2]` contain the factors for the X, Y, and Z dimensions, respectively. Array elements are 32-bit fixed-point numbers whose 16 LSBs lie to the right of the binary point.

A scaling matrix is constructed from the three scaling factors, and the matrix specified by the `matrix` argument is multiplied by the scaling matrix. See *Principles of Interactive Graphics* (Newman and Sproull) for more information on homogeneous transformations.

Example

```
typedef long FIX;
static  FIX factor[3] = { 0x8000, 0x8000, 0x8000 };
static  long xyz[] = {
    -50,-50,50, 50,-50,50, 50,50,-50, -50,50,-50
};
static short object[4] = { 0,1,2,3 };
FIX    matrix[16], verts[12];
short  xy[8];
int    i, f;

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);          /* Center origin */
set_color1(0x11111111);        /* 4-bit pixel   */
/*********************************************/
/* Scale the object in each of the 3 dimensions */
/*********************************************/
for (i = 0; i <= 2; ++i)
    for (f = 0x8000; f <= 0x18000; f += 0x200) {
        init_matrix(matrix);
        factor[i] = f;
        scale(matrix, factor);
        long_to_fix(12, xyz, verts);
        transform(matrix, 4, verts);
        perspec(4, verts, xy, 0, 0, -200);
        delay(0);
        init_screen();
        fill_convex(4, object, xy);
    }
```

Syntax

```
void seed_fill(xseed, yseed, buffer, maxbytes)
   int  xseed, yseed;  /* coordinates of seed pixel     */
   char buffer[];      /* working storage for function */
   int  maxbytes;      /* size of buffer in bytes       */
```

Description

The seed_fill function fills a connected region of pixels starting at a specified seed pixel. The seed color is the color of the specified seed pixel at the time the function is called. The connected region is solid-filled with the current COLOR1 value.

- The first two arguments, `xseed` and `yseed`, specify the coordinates of the seed pixel.
- The last two arguments specify a buffer used as working storage during the seed fill.
  - Argument `buffer` is a buffer large enough to contain the temporary data that the function uses.
  - Argument `maxbytes` is the number of 8-bit bytes available in the buffer array.

  Storage requirements can be expected to increase with the complexity of the connected region being filled.

The connected region filled by the function always includes the seed pixel. To be considered part of the connected region, a pixel must both match the seed color, and be horizontally or vertically adjacent to another pixel that is part of the connected region. (A diagonally adjacent neighbor is not sufficient.)

The seed_fill function aborts (returns immediately) if any of these conditions are detected:

- The seed pixel matches the current COLOR1 value.
- The seed pixel lies outside the current visibility rectangle (or window).
- If at any point the storage buffer space specified by `maxbytes` is insufficient to continue.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
char buf[800];
int  i;

init_video(1);
init_grafix();
init_screen();
/*********************************/
/*    Draw a connected region    */
/*********************************/
set_color1(rep_pixel(15));
draw_rect(63, 63, 0, 0);
set_color0(rep_pixel(1));
set_color1(0);
select_patn(8);
patnfill_rect(62, 62, 1, 1);
zoom_rect(64, 64, 0, 0, 320, 320, 160, 80, buf);
/*********************************/
/* Now fill the connected region */
/*********************************/
for (i = 2; i < 15; ++i) {
    set_color1(rep_pixel(i));
    seed_fill(320, 240, buf, sizeof(buf));
}
```

Syntax

```
void seed_patnfill(xseed, yseed, buffer, maxbytes)
   int  xseed, yseed;  /* coordinates of seed pixel     */
   char buffer[];      /* working storage for function */
   int  maxbytes;      /* size of buffer in bytes       */
```

Description

The seed—patnfill function fills a connected region of pixels with a pattern, starting at a specified seed pixel. The seed color is the color of the specified seed pixel at the time the function is called. The connected region is filled with the current pattern in colors COLOR0 and COLOR1.

- The first two arguments, (`xseed`, `yseed`0, specify the coordinates of the seed pixel.
- The last two arguments specify a buffer used as working storage during the seed fill.
  - Argument `buffer` is a buffer large enough to contain the temporary data that the function uses.
  - Argument `maxbytes` is the number of 8-bit bytes available in the buffer array.

  Storage requirements can be expected to increase with the complexity of the connected region being filled.

The connected region filled by the function always includes the seed pixel. To be considered part of the connected region, a pixel must both match the seed color, and be horizontally or vertically adjacent to another pixel that is part of the connected region. (A diagonally adjacent neighbor is not sufficient.)

The seed—patnfill function aborts (returns immediately) if any of these conditions are detected:

- The seed pixel matches either the current COLOR0 value or the current COLOR1 value.
- The seed pixel lies outside the current visibility rectangle (or window).
- If at any point the storage buffer space specified by `maxbytes` is insufficient to continue.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
char buf[800];
int  i;

init_video(1);
init_grafix();
init_screen();
/*******************************/
/*    Draw a connected region    */
/*******************************/
set_color1(rep_pixel(15));
draw_rect(63, 63, 0, 0);
set_color0(rep_pixel(1));
set_color1(0);
select_patn(8);
patnfill_rect(62, 62, 1, 1);
zoom_rect(64, 64, 0, 0, 400, 400, 120, 40, buf);
/*******************************/
/* Now fill the connected region */
/*******************************/
set_color0(rep_pixel(4));
set_color1(rep_pixel(7));
seed_patnfill(320, 240, buf, sizeof(buf));
```

Syntax

```
int select_font(index)
   int index;   /* identifies a font previously opened */
```

Description

The select_font function selects the font identified by `index`. The designated font is used in all subsequent text-drawing operations within the current viewport until a new font is selected.

If the font is not currently open, a value of -1 is returned to indicate an error, and the system font is selected as a default. Otherwise, a value of 0 is returned as confirmation.

> **Note:**
>
> Before you call the select_font function, call the init_text function to initialize the text data structures.

Example

```
#include "fntstruc.h"           /* FONT type definition */
extern FONT corpus_christi29,   /* Corpus Christi font  */
extern FONT montrose28,         /* Montrose font        */
extern FONT north_pole30,       /* North Pole font      */
extern FONT san_marcos21;       /* San Marcos font      */
int i, x, y;
char *s;

init_video(1);
init_grafix();
init_text();  /* Corpus Christi 16 selected as default font
    */
init_screen();
install_font(1, &corpus_christi29); /* Don't forget the & */
install_font(2, &montrose28);
install_font(3, &north_pole30);
install_font(4, &san_marcos21);
s = "Hello World.";
for (i = 0, y = 0; i <= 4; ++i) {
    select_font(i);
    x = 320 - get_width(s)/2;
    y += char_high();
    draw_string(x, y, s);
}
```

Syntax

```
int select_patn(index)
  int index;      /* index into pattern table */
```

Description

The select_patn function selects the pattern identified by the `index`. The pattern is used in all subsequent pattern-filling functions in the current viewport until another pattern is selected.

The pattern is a 16 x 16 bit map that is bit-expanded to the current COLOR0 and COLOR1 values when drawn to the screen. Argument index represents the position of the pattern within the pattern table. Runtime initialization loads the pattern table with a number of default patterns. You can use the install_patn function to install custom patterns.

If argument `index` is negative or is greater than or equal to the value returned by the get_patn_max function, a value of -1 is returned to flag the error condition. Otherwise, a value of 0 is returned to confirm that the designated pattern was selected.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int x, y, dx, dy, i, hue0, hue1;

init_video(1);
init_grafix();   /* Initialize patterns */
init_screen();
i  = get_patn_max();
dx = 480 / i;
dy = 320 / i;
x  = y = 0;
hue0 = hue1 = 0;
for (--i ; i >= 0; --i, x += dx, y += dy) {
     select_patn(i);
     set_color0(rep_pixel(hue0++));
     set_color1(rep_pixel(--hue1));
     patnfill_rect(160, 160, x, y);
}
```

Syntax

```
int select_vuport(index)
   int index;       /* viewport number */
```

Description

The select_vuport function selects the viewport identified by `index`. The designated viewport becomes the active viewport for subsequent drawing operations.

The viewport must have been previously opened. At the time the viewport was opened, the open_vuport function returned the index that identifies the viewport in subsequent transactions. The system viewport is opened automatically by the viewport initialization function, init_vuport, and is identified by an index value of 0.

The attributes of the specified viewport replace those of the previously active viewport. Refer to the description of the open_vuport function for a list of viewport attributes.

A value of 0 is returned if the index is valid. In the event of an invalid index, a value of -1 is returned and the active viewport is not changed.

> **Note:**
>
> Before you call the select_vuport function, call the init_vuport function to initialize the viewport data structures.

Example

```
int v;

init_video(1);
init_grafix();
init_vuport();   /* Open viewport 0 */
init_screen();
set_cliprect(320, 480, 0, 0);
/*****************************************/
/* Change viewport 0's colors and pattern */
/*****************************************/
set_color0(rep_pixel(1));
set_color1(rep_pixel(7));
select_patn(1);
/*****************************************/
/* Change viewport 1's colors and pattern */
/*****************************************/
v = open_vuport(); /* Open viewport 1 */
move_vuport(320, 0);
set_color0(rep_pixel(2));
set_color1(rep_pixel(12));
select_patn(4);
/*****************************************/
/* Display viewport 0's colors and pattern */
/*****************************************/
select_vuport(0);
patnfill_oval(300, 460, 10, 10);
/*****************************************/
/* Display viewport 1's colors and pattern */
/*****************************************/
select_vuport(v);
patnfill_rect(300, 460, 10, 10);
```

Syntax

```
void setall_palet(palet, reg_mask, n, y)
   short palet[16];          /* color palette          */
   int  reg_mask;            /* register-load mask     */
   int  n;                   /* no. of lines affected  */
   int  y;                   /* starting scan line     */
```

Description

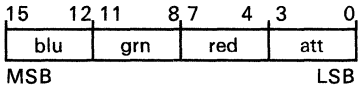
The setall_palet function loads multiple color palette registers. This function allows a designated subset of the color palette registers to be loaded from an array.

The function does not necessarily alter the color palette over the entire screen; if desired, the function can alter the palette over only a specified group of scan lines. (This is facilitated by the line-load feature of the TMS34070 color palette chip.)

- Argument `n` specifies the number of scan lines affected by the color palette load.
- Argument `y` is the first line (lowest line number) in the affected group of lines. Scan lines are numbered in ascending order from top to bottom, with line 0 at the top of the screen.
- A register mask, `reg_mask`, specifies which of the 16 color palette registers are loaded from the `palet` array. Only registers corresponding to 1s in the mask are loaded. Mask bits 0 through 15 control the loading of registers 0 through 15. For example, a mask value of 0x0027 enables the loading of registers 0, 1, 2, and 5 from `palet` array members 0, 1, 2, and 5. Only the 16 LSBs of the mask are used; the 16 MSBs are ignored.

This function assumes that the system contains a TMS34070 color palette or functional equivalent. The pixel size is therefore four bits, and the palette contains 16 registers. The values contained in the palette registers can change on a line-by-line basis.

Each 16-bit palette register is organized according to the following format:



The red, green, and blue intensity fields are 4-bit, unsigned binary numbers. The attribute field contains a color-repeat bit that is set to one to enable automatic filling by the palette device. See the *TMS34070 User's Guide* for details.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
static short mypalet[] = {
       0x0FF0, 0x0EF0, 0x0DF0, 0x0CF0,
       0x0BF0, 0x0AF0, 0x09F0, 0x08F0,
       0x07F0, 0x06F0, 0x05F0, 0x04F0,
       0x03F0, 0x02F0, 0x01F0, 0x00F0,
       0x10F0, 0x20F0, 0x30F0, 0x40F0,
       0x50F0, 0x60F0, 0x70F0, 0x80F0,
       0x90F0, 0xA0F0, 0xB0F0, 0xC0F0,
       0xD0F0, 0xE0F0, 0xF0F0
       };
int i;

init_video(1);
init_grafix();
init_screen();
/*************************************/
/* Fill horizontal strips in 16 colors */
/*************************************/
for (i = 0; i <= 15; ++i) {
     set_color1(rep_pixel(i));
     fill_rect(40, 480, 40*i, 0);
}
/*************************************/
/*    Change palette every 27 lines    */
/*************************************/
for (i = 0; i < 15; ++i)
    setall_palet(&mypalet[i], 0xFFFF, 27, 40+i*27);
```

Syntax

```
void set_cliprect(w, h, xleft, ytop)
   int w, h;          /* width and height of */
                      /* clipping rectangle  */
   int xleft, ytop;   /* top left corner of  */
                      /* clipping rectangle  */
```

Description

The set_cliprect function sets the size and position of the clipping rectangle for subsequent drawing operations. Drawing operations can alter only pixels within the visibility rectangle formed by the intersection of:

- The viewport,
- The clipping rectangle, **and**
- The screen.

Four arguments define the size and position of the clipping rectangle:

- The width `w`,
- The height `h`, **and**
- The coordinates of the top left corner (`xleft,ytop`).

The coordinates are expressed as displacements from the origin of the active viewport. If the viewport or viewport-relative origin is moved, the clipping rectangle moves accordingly.

> **Note:**
>
> Before you call the set_cliprect function, call the init_vuport function to initialize the viewport data structures.

Example

```
init_video(1);
init_grafix();
init_vuport();
init_screen();
set_cliprect(128, 96, 64, 48);
patnfill_rect(1000, 1000, -100, -100);
set_origin(175, 150);
set_color0(rep_pixel(1));
set_color1(rep_pixel(3));
select_patn(4);
patnfill_oval(256, 192, -64, -48);
move_vuport(160, 120);
set_color1(rep_pixel(6));
fill_rect(1000, 1000, -100, -100);
```

Syntax

```
void set_color0(pixel_val)
   long pixel_val;     /* pixel value replicated */
                       /* to 32 bits             */
```

Description

The set_color0 function changes the COLOR0 value. This is the pixel value to which 0s in bit maps for text fonts and two-dimensional bit patterns are expanded as they are drawn to the screen.

The pixel value is replicated through the entire 32-bit argument. Given a pixel size of *n* bits, the *n*-bit pixel value must be replicated 32/*n* times. For example, at four bits per pixel, a value of 5 is replicated 32/4 = 8 times to form the pixel_val argument value 0x55555555.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
#include "fntstruc.h"              /* Define FONT type     */
extern FONT corpus_christi29;

static char *s[] = {
        "0",  "1",  "2",  "3",  "4", "5",
        "6",  "7",  "8",  "9", "10",
       "11", "12", "13", "14", "15"
       };
int i;

init_video(1);
init_grafix();                      /* Set default colors */
init_text();
install_font(1,&corpus_christi29); /* Remember the &     */
init_screen();
for (i = 0; i <= 15; ++i) {
    select_patn(i);
    set_color0(rep_pixel(i));
    draw_string(i*40+5, 25, s[i]);
    patnfill_oval(30, 430, i*40+5, 50);
}
```

Syntax

```
void set_color1(pixel_val)
   long pixel_val;     /* pixel value replicated */
                       /* to 32 bits             */
```

Description

The set_color1 function changes the COLOR1 value. This is the pixel value that is used for lines and solid fills. It is also the value to which 1s in bit maps for text fonts and two-dimensional bit patterns are expanded as they are drawn to the screen.

The pixel value is replicated through the entire 32-bit argument. Given a pixel size of *n* bits, the *n*-bit pixel value must be replicated 32/*n* times. For example, at four bits per pixel, a value of 5 is replicated 32/4 = 8 times to form the pixel_val argument value 0x55555555.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
#include "fntstruc.h"          /* Define FONT type   */
extern FONT corpus_christi29;

static char *s[] = {
        "0", "1", "2", "3", "4", "5",
        "6", "7", "8", "9", "10",
       "11", "12", "13", "14", "15"
       };
int i;

init_video(1);
init_grafix();                 /* Set default colors */
init_text();
install_font(1,&corpus_christi29); /* Remember the & */
init_screen();
for (i = 0; i <= 15; ++i) {
    select_patn(i);
    set_color1(rep_pixel(i));
    draw_string(i*40+5, 25, s[i]);
    fill_rect(30, 215, i*40+5, 50);
    patnfill_oval(30, 215, i*40+5, 265);
}
```

Syntax

```
void set_origin(x0, y0)
   int x0, y0;     /* displacement from viewport */
                   /* top left corner            */
```

Description

The set_origin function sets the position of the XY coordinate origin for the active viewport. This origin is used for subsequent drawing operations to the viewport.

Arguments `x0` and `y0` define the new position of the origin as displacements from the top left corner of the active viewport. The clipping rectangle, whose position is relative to the origin, is automatically adjusted to follow the change in position of the origin. If the viewport is subsequently moved, the origin and clipping rectangle move with it.

> **Note:**
>
> Before you call the set_origin function, call the init_vuport function to initialize the viewport data structures.

Example

```
static short ptlist[] = {
        0,-10, 0,70, -4,62,  4,62,
      -10,0,  70,0,  62,-4, 62,4
};
static short axes[] = {
    0,1, 1,2, 1,3, 4,5, 5,6, 5,7
};
int i, x, y;

init_video(1);
init_grafix();
init_screen();
init_vuport();          /* Set default origin */
/*********************************************/
/* Move origin to various positions on screen */
/*********************************************/
for (x = 10; x < 639; x += 100)
    for (y = 10; y < 479; y += 100) {
        set_origin(x, y);
        draw_polyline(6, axes, ptlist);
    }
```

Syntax

```
void set_palet(reg, red, grn, blu)
   int reg;           /* color palette register (0-15) */
   int red, grn, blu; /* RGB intensities (0-15)        */
```

Description

The set_palet function loads the designated color palette register with specified red, green, and blue intensities. The color palette is updated over the entire screen (all scan lines are affected).

- Argument `reg` specifies a color palette register number in the range 0 to 15.
- Arguments `red`, `grn`, and `blu` specify 4-bit intensities in the range 0 to 15. Only the four LSBs of the intensities are used; higher-order bits are ignored.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int i, r, g, b;

init_video(1);
init_grafix();
clear_screen(0);
/*******************************/
/*    Fill vertical stripes    */
/*******************************/
for (i = 0; i <= 15; ++i) {
    set_color1(rep_pixel(i));
    fill_rect(40, 480, i*40, 0);
}
/*******************************/
/* Change color palette values. */
/*******************************/
r = 15;
g = b = 0;
for (i = 0; i <= 15; ++i)
    set_palet(i, r--, g++, b++);
```

Syntax

```
void set_pensize(w, h)
   int w, h;     /* pen width and height */
```

Description

The set—pensize function specifies the width and height of a rectangular pen for the active viewport. These pen dimensions are used for any subsequent drawing operations that use the pen.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int i, x1, y1, x2, y2;

init_video(1);
init_grafix();
init_vuport();
set_origin(320,240);
init_screen();
/*****************************************/
/* Draw lines with 50 different pen sizes */
/*****************************************/
x2 = 0;
y2 = -200;
for (i = 50; i > 0; --i) {
    x2 -= y2 >> 3;
    y2 += x2 >> 3;
    x1 = x2 >> 3;
    y1 = y2 >> 3;
    set_color1(rep_pixel(i));
    set_pensize(i, i);
    pen_line(x1, y1, x2, y2);
}
```

Syntax

```
void set_pmask(mask)
   long pmask;       /* plane mask */
```

Description

The set—pmask function specifies the plane mask that is used in subsequent drawing operations. The `mask` determines which bits in a pixel can be modified during drawing operations. The 0s in the mask enable modification of the corresponding bit planes. The 1s in the mask write-protect the corresponding planes.

The mask size is in principle the same as the pixel size, but it must be replicated through the entire 32-bit mask argument. Given a pixel size of *n* bits, the *n*-bit mask value must be replicated 32/*n* times. For example, at four bits per pixel, a mask value of 6 is replicated 32/4 = 8 times to form the pixel—val argument value 0x66666666.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short palet[] = {
       0x0000, 0x00F0, 0x6F60, 0x08E0,
       0xF000, 0xF0F0, 0xF800, 0x8880,
       0xFFF0, 0xFFF0, 0xFFF0, 0xFFF0,
       0xFFF0, 0xFFF0, 0xFFF0, 0xFFF0
};
int i, x, y, dx, dy;

init_video(1);
init_grafix();
new_screen(0, palet);
/*************************************/
/* Assume 4 bits per pixel           */
/* Write only to plane 3             */
/*************************************/
set_pmask(rep_pixel(7));
set_color1(rep_pixel(8));
patnfill_rect(480, 352, 80, 64);
/*************************************/
/* Write only to planes 0, 1 and 2   */
/* Note that plane 3 remains unaltered */
/*************************************/
set_pmask(rep_pixel(8));
x = y = 0;
dx = 28;
dy = 13;
i = 0;
for ( ; ; ) {
    if (cpw(x, y) & 0x3)
        dx = -dx;
    if (cpw(x, y) & 0xC)
        dy = -dy;
    x += dx;
    y += dy;
    set_color1(rep_pixel(i++));
    fill_rect(100, 100, x-50, y-50);
}
```

Syntax

```
void set—ppop(ppop—code)
   int ppop—code;   /* pixel processing operation code */
```

Description

The set—ppop function defines the pixel processing operation for subsequent drawing operations. The specified Boolean or arithmetic operation determines the manner in which source and destination pixel values are combined. The range for the ppop—code argument is 0 to 21. The codes are described in the following table:

Code	Replace Destination Pixel with:	Code	Replace Destination Pixel with:
0	source	11	NOT source AND destination
1	source AND destination	12	all 1s
2	source AND NOT destination	13	NOT source OR destination
3	all 0s	14	source NAND destination
4	source OR NOT destination	15	NOT source
5	source EQU destination	16	source + destination
6	NOT destination	17	ADDS(source, destination)
7	source NOR destination	18	destination - source
8	source OR destination	19	SUBS(destination, source)
9	destination	20	MAX(source, destination)
10	source XOR destination	21	MIN(source, destination)

The details of these operations are described in the *TMS34010 User's Guide*.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
static short  x[] = { 300, 500, 20, 123 };
static short  y[] = { 400, 100, 50, 321 };
static short dx[] = { 1, 2, 3, 4 };
static short dy[] = { 1, 2, 1, 2 };
int    ppop, j, i;

init—video(1);
init—grafix();
init—screen();
/*************************************/
/* Show effects of ppops 0 through 21 */
/*************************************/
for (ppop = 0; ; ) {
    set—ppop(ppop);
    if (++ppop > 21)
        ppop = 0;
    for (j = 0; j < 1000; ++j)
        for (i = 0; i <= 3; ++i) {
            if (cpw(x[i], y[i]) & 0x3)
                dx[i] = -dx[i];
            if (cpw(x[i], y[i]) & 0xC)
                dy[i] = -dy[i];
            x[i] += dx[i];
            y[i] += dy[i];
            set—color1(rep—pixel(i+1));
            fill—oval(80, 60, x[i]-40, y[i]-30);
        }
}
```

Syntax

```
FIX *short_to_fix(n, in_array, out_array)
   typedef long FIX;
   int   n;              /* number of elements to be    */
                         /* converted                   */
   short in_array[];     /* array of integers           */
   FIX   out_array[];    /* array of fixed-point values */
```

Description

The short_to_fix function converts an array of short integers to fixed-point numbers. Elements of the input array are 16-bit, 2s complement integers (C type short). Elements of the output array are 32-bit, 2s complement, fixed-point numbers whose 16 LSBs are to the right of the binary point. The conversion to fixed-point format is done by simply shifting the integer elements left by 16.

Three input arguments are specified:

- The number of elements `n` that are converted,
- The input array `in_array`, **and**
- The output array `out_array`.

A pointer to the first element of the output array is returned.

The value returned by the function is a pointer to the output array, `out_array`.

Example

```
typedef long  FIX;
static  short ptlist[] = { 0,-20, 30,15, -30,15 };
static  short triangle[] = { 0,1, 1,2, 2,0 };
FIX xy[6];
int i, j;

init_video(1);
init_grafix();
init_vuport();
init_screen();
short_to_fix(6, ptlist, xy);
set_origin(320, 240);
for (j = 0; j < 100; ++j) {
    for (i = 0; i <= 5; ++i)
        xy[i] += xy[i] >> 4;
    fix_to_short(6, xy, ptlist);
    draw_polyline(3, triangle, ptlist);
}
```

Syntax

```
double sin(x)
   double x;
```

Description

The sin function calculates the sine of an angle expressed in radians. Both argument x and the return value are double-precision floating-point values.

For arguments greater than 1.0 E+8, a value of 0 is returned, and fp_error is called with error code = 17 (see the description of the floating-point facility in the *TMS34010 C Compiler User's Guide*).

Example

```
extern double sin();
double radian, sval;  /* sin returns sval */
radian = 3.1415927;
sval = sin(radian);   /* sin returns 0    */
```

Syntax

```
double sinh(x)
   double x;
```

Description

The sinh function returns the hyperbolic sine of a real number `x`. Both the argument `x` and return value are double-precision floating-point values.

Example

```
extern double sinh();
double x, y;

x = 0.0;
y = sinh(x);        /* return value = 0.0 */
```

Syntax

```
int size_vuport(w, h)
   int w, h;      /* width and height of viewport */
```

Description

The size_vuport function changes the size of the active viewport. The viewport is rectangular. The top left corner of the viewport remains fixed at its original position; the lower right corner moves to expand or contract the viewport to w and height h.

> **Note:**
>
> Before you call the size_vuport function, call the init_vuport function to initialize the viewport data structures.

Example

```
int i, w, h;

init_video(1);
init_grafix();
init_vuport();
init_screen();
i = 0;
/***********************************/
/* Show changes in size of viewport */
/***********************************/
for (w = 640, h = 480; w > 0; w -= 32, h -= 24) {
    size_vuport(w, h);
    set_color1(rep_pixel(++i));
    fill_rect(2000, 2000, -1000, -1000);
}
```

Syntax

```
double sqrt(x)
   double x;
```

Description

The sqrt function calculates the square root of a real number x.

If the argument x is negative, the square root of the absolute value of x is returned, and fp—error is called with error code = 10 (see the description of the floating-point facility in the *TMS34010 C Compiler User's Guide*).

Example

```
extern double sqrt();
double x, y;

x = 100.0;
y = sqrt(x);          /* return value = 10.0 */
```

Syntax

```
long styled_line(x1, y1, x2, y2, style, mode)
   int  x1, y1;    /* start coordinates            */
   int  x2, y2;    /* end coordinates              */
   long style;     /* 32-bit repeating line-style */
                   /* pattern                      */
   int  mode;      /* selects 1 of 2 drawing modes*/
```

Description

The styled_line function uses Bresenham's algorithm to draw a styled line from point (x1,y1) to point (x2,y2). The line is a single pixel in thickness, and is drawn in the specified line-style pattern.

- Arguments x1 and y1 specify the starting coordinates.
- Arguments x2 and y2 specify the ending coordinates.
- The last two arguments, style and mode, specify the line style and drawing mode.
  - Argument style is a long integer containing a 32-bit repeating line-style pattern. Pattern bits are used in the order 0,1,...,31, where 0 is the rightmost bit (the LSB). The pattern is repeated modulo 32 as the line is drawn. A bit value of 1 in the pattern specifies that COLOR1 is used to draw the corresponding pixel. A value of 0 in one of the line-style pattern bits means that the corresponding pixel is either drawn in COLOR0 (if mode = 1) or not drawn (if mode = 0).
  - The mode argument selects one of two drawing modes. If mode = 1, pixel positions corresponding to 0s in the pattern are drawn in COLOR0. If mode = 0, the COLOR0 pixels are not drawn; that is, the line skips over pixel positions corresponding to 0s in the line-style pattern. The function uses only the LSB of mode; the function ignores higher-order bits of the argument.

The value returned is the style pattern rotated left (n-1) modulo 32, where n is the number of pixels in the line drawn by the function (counting both the COLOR1 and COLOR0 pixels). This return value can be used as the line-style pattern for a new line that continues from the end point of the line just drawn. The line-style pattern will be continuous from the old line to the new line.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
long x1, y1, x2, y2, i, mask;

init_video(1);
init_grafix();
init_vuport();
init_screen();
set_origin(320, 240);
/*****************************************/
/* Draw spiral using styled line segments */
/*****************************************/
x2 = 0;
y2 = -20 << 16;
mask = 0x93E493E4;  /* line-style pattern */
for (i = 5000; i > 0; --i) {
    x1  = x2;
    y1  = y2;
    x2 += y1 >> 4;
    y2 -= x1 >> 4;
      mask = styled_line(x1>>16, y1>>16, x2>>16, y2>>16,
mask, 0);
}
```

Syntax

```
double tan(x)
   double x;
```

Description

The tan function calculates the tangent of an angle x expressed in radians. Both argument x and the return value are double-precision floating-point values.

If the absolute value of argument x is greater than 1.0E+8, a value of 0 is returned, and fp_error is called with error code = 20 (see the description of the floating-point facility in the *TMS34010 C Compiler User's Guide*).

Example

```
extern double tan();
double x, y;

x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

Syntax

```
double tanh(x)
   double x;
```

Description

The tanh function returns the hyperbolic tangent of a real number x Both the argument x and the return value are double-precision floating-point numbers.

Example

```
extern double tanh();
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

Syntax

```
void transform(matrix, n, verts)
   typedef long FIX; /* fixed-point type definition  */
   FIX matrix[16];   /* 4×4 transformation matrix    */
   int n;            /* number of vertices in vertex */
                     /* list                         */
   FIX verts[];      /* vertex list (x, y, and z)    */
```

Description

The transform function uses a 4×4 transformation matrix to transform (rotate, scale and translate) a list of three-dimensional vertices.

- The 4×4 transformation matrix, `matrix`, is a 16-element array of fixed-point values. A fixed-point value is 32 bits long, and the 16 LSBs lie to the right of the binary point. Embedded in the matrix transformation is a sequence of scaling, rotation and translation operations.
- Argument `n` specifies the number of vertices that are transformed by the matrix.
- Vertex list `verts` is an array containing the three-dimensional coordinates of the n vertices. Each vertex in the array is a 96-bit value consisting of X, Y, and Z coordinate values in fixed-point format.

The data structure for the vertex list, `verts`, is organized as follows:

```
verts[0]     = X coordinate at vertex 0
verts[1]     = Y coordinate at vertex 0
verts[2]     = Z coordinate at vertex 0
verts[3]     = X coordinate at vertex 1
verts[4]     = Y coordinate at vertex 1
verts[5]     = Z coordinate at vertex 1
  .
  .
  .
verts[3n]    = X coordinate at vertex n-1
verts[3n+1]  = Y coordinate at vertex n-1
verts[3n+2]  = Z coordinate at vertex n-1
```

Example

```
typedef long  FIX;
static  FIX   rotation[] = { 0, 0, 0 };
static  FIX   xyz[] = { -150,-200,0, 150,-200,0, 0,0,0 };
static  short connect[] = { 0, 1, 2 };
FIX   matrix[16], verts[3*3];
short xy[2*3];
int   angle;

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
for (;;)
    for (angle = 0; angle < 360; ++angle) {
        init_matrix(matrix);
        rotation[0] = angle << 16;
        rotate(matrix, rotation);
        long_to_fix(3*3, xyz, verts);
        transform(matrix, 3, verts);
        vertex_to_point(3, verts, xy);
        delay(0);
        init_screen();
        fill_convex(3, connect, xy);
    }
```

Syntax

```
void translate(matrix, disp)
   typedef long FIX;
   FIX matrix[16]          /* 4x4 transformation matrix */
   FIX disp[3]             /* displacements in x, y, z  */
```

Description

The translate function multiplies a 4×4 transformation matrix by a translation matrix constructed from displacements in the X, Y, and Z directions.

- Argument `matrix` is a 4×4 homogeneous transformation matrix. Each matrix element is stored as a 32-bit fixed-point value whose 16 LSBs lie to the right of the binary point. Refer to the definition of the init_matrix function for a description of the matrix structure.

- Argument `disp` is a three element array containing the displacements in X, Y, and Z (in that order). Each displacement is stored as a 32-bit fixed-point value. The translation matrix by which the transformation matrix is multiplied is formed from the elements of the `disp` array as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ disp[0] & disp[1] & disp[2] & 1 \end{bmatrix}$$

Refer to *Principles of Interactive Graphics* (Newman and Sproull) for additional information on homogeneous three-dimensional transformations.

Example

```
typedef long FIX;
static  FIX  rotation[3]  = { 0, 0, 0 };
static  FIX  translat1[3] = { -320, -240, 0 };
static  FIX  translat2[3] = { 320, 240, 0 };
static  long xyz[] = {
    320,40,0, 340,240,0, 320,260,0, 300,240,0
};
static short connect[8] = { 0,1, 1,2, 2,3, 3,0 };
FIX   matrix[16];
FIX   verts[12];
short xy[8];
int   angle;

init_video(1);
init_grafix();
long_to_fix(3, translat1, translat1);
long_to_fix(3, translat2, translat2);
for (;;)
    for (angle = 0; angle < 360; ++angle) {
        init_matrix(matrix);
        translate(matrix, translat1);
        rotation[0] = angle << 16;
        rotate(matrix, rotation);
        translate(matrix, translat2);
        long_to_fix(12, xyz, verts);
        transform(matrix, 4, verts);
        vertex_to_point(4, verts, xy);
        delay(0);
        init_screen();
        draw_oval(420, 420, 110, 30);
        draw_polyline(4, connect, xy);
    }
```

Syntax

```
void transp_off()
```

Description

The transp_off function disables transparency for subsequent drawing operations.

When transparency is enabled, and the result of a pixel operation involving the source and destination pixels is 0, the destination pixel is not altered. The transp_off disables transparency, and the result of a subsequent pixel operation is written to the destination regardless of its value.

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
init_video(1);
init_grafix();
init_screen();              /* Sets 0 = black */
/*********************************************/
/*          Construct source array           */
/*********************************************/
frame_rect(190, 190, 25, 145, 2, 2);
set_color1(rep_pixel(6));
frame_oval(150, 150, 45, 165, 35, 35);
/*********************************************/
/*     Construct 2 copies of destination     */
/*********************************************/
set_color1(rep_pixel(12));
fill_rect(190, 190, 225, 145);
set_color1(rep_pixel(1));
fill_rect(30, 190, 305, 145);
fill_rect(190, 30, 225, 225);
move_rect(200, 200, 220, 140, 420, 140);
/*********************************************/
/* Copy source to 1st dest. with transp. ON  */
/*********************************************/
transp_on();
move_rect(160, 160, 40, 160, 240, 160);
/*********************************************/
/* Copy source to 2nd dest. with transp. OFF */
/*********************************************/
transp_off();
move_rect(160, 160, 40, 160, 440, 160);
```

Syntax `void transp_on()`

Description The transp—on function enables transparency for subsequent drawing operations.

When transparency is enabled, and the result of a pixel operation involving the source and destination pixels is 0, the destination pixel is not altered. The transp—on enables transparency, and the result of a subsequent pixel operation is written to the destination only if it is not 0.

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example Refer to example in description of transp—off function.

Syntax

```
void vertex_to_point(n, verts, ptlist)
   int    n;           /* number of vertices to convert */
   FIX    verts[];     /* list of 3D vertices (x,y,z)   */
   short  ptlist[];    /* list of 2D points (x,y)       */
```

Description

The vertex_to_point function converts a list of three-dimensional vertices to a list of two-dimensional points. Each three-dimensional vertex is represented in terms of its X, Y, and Z coordinates. Each two-dimensional point is represented in terms of its X and Y coordinates.

- Argument `n` specifies the number of vertices that are converted.
- Each three-dimensional vertex is represented in the `verts` array as three adjacent array elements containing the X, Y, and Z coordinate values, respectively. Each element is a 32-bit fixed-point value whose 16 LSBs lie to the right of the binary point. The number of elements in the `verts` array is three times `n` the number of vertices.
- Each two-dimensional point is represented in the `ptlist` array as two adjacent elements representing the X and Y coordinate values, respectively. Each element is a 16-bit integer. The number of elements in the `ptlist` array is two times `n`, the number of vertices.

Each 96-bit vertex in the `verts` array is converted to a 32-bit point in the `ptlist` array. The function converts the integer parts of the X and Y coordinate values in the `verts` array to X and Y coordinate values in the `ptlist` array. The Z values are excluded from the `ptlist` array.

Refer also to the definition of the perspec function, which similarly converts a list of 3D vertices to a list of 2D points, but first performs a perspective transformation on the vertices.

Example

```
typedef long FIX;
static  long xyz[] = {
    0,-100,0,  100,0,0, 0,100,0, -100,0,0
};
static  short diamond[] = { 0, 1, 2, 3 };
FIX   verts[12];
short xy[8];

init_video(1);
init_grafix();
init_vuport();
set_origin(320, 240);
init_screen();
long_to_fix(12, xyz, verts);
vertex_to_point(4, verts, xy);
patnfill_convex(4, diamond, xy);
```

Syntax

```
void wait_scan(line)
   int line;  /* wait until this scan line is reached */
```

Description

The wait—scan function waits for a scan line on the CRT to be refreshed. This function does not return control to the calling routine until the specified line is scanned by the electron beam. Control is returned at the start of the horizontal blanking interval that follows the designated line. Scan lines are numbered in ascending order, starting with line 0 at the top of the screen. Only visible scan lines are counted.

You can use this function to synchronize drawing operations to the electron beam of a CRT display. For example, when drawing an animated sequence of frames, transitions from one frame to the next appear smoother if an area of the screen is not redrawn at the same time it is output to the CRT.

If argument `line` < 0, the function uses the value 0 in place of the argument value. If `line` is greater than the bottom scan line, the function uses the number of the bottom scan line in place of the argument value.

The wait—scan function cannot be used in an application in which the display interrupt is enabled. If the display interrupt is enabled, the function returns immediately (no error code is returned).

> **Note:**
>
> Before you call this function, call the init—grafix function to initialize the graphics environment.

Example

```
int r, x, y, vx, vy;

init_video(1);
init_grafix();
init_vuport();
r  = 100;
x  = 320;
y  = 240;
vx = 1;
vy = -3;
/*******************************************/
/* Wait only if ball is on left side of CRT */
/*******************************************/
for ( ; ; ) {
    if (cpw(x, y) & 3)
        vx = -vx;
    if (cpw(x, y) & 12)
        vy = -vy;
    x += vx;
    y += vy;
    if (x < 320)
        wait_scan(y+r);
    clear_screen(0);
    init_palet();
    fill_oval(2*r, 2*r, x-r, y-r);
}
```

Syntax

```
long xytoaddr(x, y)
   int x, y;  /* viewport-relative x and y coordinates */
```

Description

The xytoaddr function calculates the 32-bit memory address of the pixel located at viewport-relative coordinates (x,y).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
int  w, h, x, y, saddr, sptch;
char *s;

init_video(1);
init_grafix();
init_text();
init_screen();
x = y = 100;
s = "Use cheap trick to slant characters.";
draw_string(x, y, s);
/*************************************************/
/* Calculate address at top left corner of text */
/*************************************************/
saddr = xytoaddr(x - char_high(), y - get_ascent());
sptch = peek_breg(3) + get_psize(); /* SPTCH+PSIZE */
h = char_high();
w = get_width(s) + h;
/*************************************************/
/* Treat region of screen as source pixel array */
/*************************************************/
put_rect(saddr, sptch, w, h, x, y + h);
```

Syntax

```
void zoom_rect(ws, hs, xs, ys, wd, hd, xd, yd, linebuf)
   int   ws, hs;     /* source width and height       */
   int   xs, ys;     /* source top left corner        */
   int   wd, hd;     /* destination width and height */
   int   xd, yd;     /* destination top left corner  */
   short linebuf[]; /* scan line buffer              */
```

Description

The zoom_rect function expands or shrinks a source rectangle on the screen to fit the dimensions of a destination rectangle that is on the screen. Horizontal zooming is accomplished by replicating (if expanding) or deleting (if shrinking) columns of pixels from the source array. Vertical zooming is accomplished by replicating or deleting rows of pixels. This type of function is sometimes referred to as a "stretch blit."

- The first four arguments define the source rectangle:
  - The width `ws`,
  - The height `hs`, **and**
  - The coordinates (`xs`,`ys`) at the top left corner of the rectangle.

  `ws` and `hs` must be nonnegative.

- The next four arguments define the destination rectangle:
  - The width `wd`,
  - The height `hd`, **and**
  - The coordinates (`xs`,`ys`) at the top left corner of the rectangle.

  `wd` and `hd` must be nonnegative.

- The final argument, `linebuf`, is a buffer large enough to contain one complete line of the display. The function uses the buffer as temporary working storage; the buffer's original contents are destroyed. The minimum buffer size (in bits) is the number of pixels per line times the number of bits per pixel.

This zoom function either expands or shrinks the source array, depending on its dimensions relative to the destination rectangle. Shrinking collapses several pixels in the source array into a single pixel in the destination rectangle. The source pixels collapsed in this manner are combined according to the current pixel processing operation (see the set_ppop function). For example, the replace operation simply selects a single source pixel to represent all the source pixels in the region being collapsed. A better result can often be obtained using a logical-OR operation (at one bit per pixel) or a max operation (at multiple bits per pixel).

> **Note:**
>
> Before you call this function, call the init_grafix function to initialize the graphics environment.

Example

```
/*****************************************/
/*       Assume pixel size is 4 bits     */
/*****************************************/
typedef struct { unsigned pixelsize : 4; } PIXEL;
static PIXEL image[] = {
    6,7,6,7,6,7,7,7,6,7,6,6,6,7,6,6,6,7,7,7,6,
    4,7,4,7,4,7,4,4,4,7,4,4,4,7,4,4,4,7,4,7,4,
    5,7,7,7,5,7,7,5,5,7,5,5,5,7,5,5,5,7,5,7,5,
    1,7,1,7,1,7,1,1,1,7,1,1,1,7,1,1,1,7,1,7,1,
    3,7,3,7,3,7,7,7,3,7,7,7,3,7,7,7,3,7,7,7,3
};
PIXEL buf[4*640];    /* screen width = 640 */
int wd, hd, xd, yd;

init_video(1);
init_grafix();
init_screen();
put_rect(image, 21*4, 21, 5, 0, 0);
wd = 42;
hd = 10;
xd = 4;
yd = 6;
while (wd < 300) {
    zoom_rect(21, 5, 0, 0, wd, hd, xd, yd, buf);
    xd += wd / 8;
    yd += hd + 1;
    wd += wd / 8;
    hd += hd / 8;
}
```

Index

G

H

I

K

L

M

N

O

P

R

S

T

V

W

X

Z

3

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-8187; **Sacramento:** 1900 Point West Way, Suite 171, Sacramento, CA 95815, (916) 929-1521; **San Diego:** 4333 View Ridge Ave., Suite B, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Ft. Lauderdale: 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland:** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 660-4600; **Tampa:** 5010 W. Kennedy Blvd., Suite 101, Tampa, FL 33609, (813) 870-6420.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Indianapolis:** 2346 S. Lynhurst, Suite J-400, Indianapolis, IN 46241, (317) 248-8555.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 200, Cedar Rapids, IA 52402, (319) 395-9550.

MARYLAND: Baltimore: 1 Rutherford Pl., 7133 Rutherford Rd., Baltimore, MD 21207, (301) 944-8600.

MASSACHUSETTS: Waltham: 504 Totten Pond Rd., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: Kansas City: 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis:** 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Endicott:** 112 Nanticoke Ave., P.O. Box 618, Endicott, NY 13760, (607) 754-3900; **Melville:** 1 Huntington Quadrangle, Suite 3C10, P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23408 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Dayton:** Kingsley Bldg., 4124 Linden Ave., Dayton, OH 45432, (513) 258-3877.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington: 260 New York Dr., Ft. Washington, PA 19034, (215) 643-6450; **Coraopolis:** 420 Rouser Rd., 3 Airport Office Park, Coraopolis, PA 15108, (412) 771-8550.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00919, (809) 753-8700.

TEXAS: Austin: P.O. Box 2909, Austin, TX 78769, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75080, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green SE, Suite 200, Murray, UT 84107, (801) 266-8972.

VIRGINIA: Fairfax: 2750 Prosperity, Fairfax, VA 22031, (703) 849-1400.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 785-7140.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181, **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 335-8392.

ARGENTINA: Texas Instruments Argentina S.A.I.C.F.: Esmeralda 130, 15th Floor, 1035 Buenos Aires, Argentina, 1 + 394-3008.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: Mercure Centre, Raketstraat 100, Rue de la Fusee, 1130 Brussels, Belgium, 2/720.80.00.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, DK-2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Teollisuuskatu 19D 00511 Helsinki 51, Finland, (90) 701-3133.

FRANCE: Texas Instruments France: Headquarters and Prod. Plant, BP 05, 06270 Villeneuve-Loubet, (93) 20-01-01; Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay, (3) 946-97-12; Lyon Sales Office, L'Oree D'Ecully, Batiment B, Chemin de la Forestiere, 69130 Ecully, (7) 833-04-40; Strasbourg Sales Office, Le Sebastopol 3, Quai Kleber, 67055 Strasbourg Cedex, (88) 22-12-66; Rennes, 23-25 Rue du Puits Mauger, 35100 Rennes, (99) 31-54-86; Toulouse Sales Office, Le Peripole—2, Chemin du Pigeonnier de la Cepiere, 31100 Toulouse, (61) 44-18-19; Marseille Sales Office, Noilly Paradis—146 Rue Paradis, 13006 Marseille, (91) 37-25-30.

TEXAS INSTRUMENTS

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, D-8050 Freising, 8161 + 80-4591; Kurfuerstendamm 195/196, D-1000 Berlin 15, 30 + 882-7365; III, Hagen 43/Kibbelstrasse, 19, D-4300 Essen, 201-24250; Frankfurter Allee 6-8, D-6236 Eschborn 1, 06196 + 8070; Hamburgerstrasse 11, D-2000 Hamburg 76, 040 + 220-1154, Kirchhorsterstrasse 2, D-3000 Hannover 51, 511 + 648021; Maybachstrabe 11, D-7302 Ostfildern 2-Nelingen, 711 + 547001; Mixikoring 19, D-2000 Hamburg 60, 40 + 637 + 0061; Postfach 1309, Roonstrasse 16, D-5400 Koblenz, 261 + 35044.

HONG KONG (+ PEOPLES REPUBLIC OF CHINA): Texas Instruments Asia Ltd., 8th Floor, World Shipping Ctr., Harbour City, 7 Canton Rd., Kowloon, Hong Kong, 3 + 722-1223.

IRELAND: Texas Instruments (Ireland) Limited: Brewery Rd., Stillorgan, County Dublin, Eire, 1 831311.

ITALY: Texas Instruments Semiconduttori Italia Spa: Viale Delle Scienze, 1, 02015 Cittaducale (Rieti), Italy, 746 694.1; Via Salaria KM 24 (Palazzo Cosma), Monterotondo Scalo (Rome), Italy, 6 + 9003241; Viale Europa, 38-44, 20093 Cologno Monzese (Milano), 2 2532541; Corso Svizzera, 185, 10100 Torino, Italy, 11 774545; Via J. Barozzi 6, 40100 Bologna, Italy, 51 355851.

JAPAN: Texas Instruments Asia Ltd.: 4F Aoyama Fuji Bldg., 6-12, Kita Aoyama 3-Chome, Minato-ku, Tokyo, Japan 107, 3-498-2111; Osaka Branch, 5F, Nissho Iwai Bldg., 30 Imabashi 3- Chome, Higashi-ku, Osaka, Japan 541, 06-204-1881; Nagoya Branch, 7F Daini Toyota West Bldg., 10-27, Meieki 4-Chome, Nakamura-ku Nagoya, Japan 450, 52-583-8691.

KOREA: Texas Instruments Supply Co.: 3rd Floor, Samon Bldg., Yuksam-Dong, Gangnam-ku, 135 Seoul, Korea, 2 + 462-8001.

MEXICO: Texas Instruments de Mexico S.A.: Mexico City, AV Reforma No. 450 — 10th Floor, Mexico, D.F., 06600, 5 + 514-3003.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 + 274681.

NETHERLANDS: Texas Instruments Holland B.V., P.O. Box 12995, (Bullewijk) 1100 CB Amsterdam, Zuid-Oost, Holland 20 + 5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 131, Oslo 1, Norway, (2) 155090.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., 8747 Paseo de Roxas, Makati, Metro Manila, Philippines, 2 + 8188987.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Asia Ltd.: 12 Lorong Bakar Batu, Unit 01-02, Kolam Ayer Industrial Estate, Republic of Singapore, 747-2255.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 16, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): Box 39103, 10054 Stockholm, Sweden, 8 - 235480.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co.: Room 903, 205 Tun Hwan Rd., 71 Sung-Kiang Road, Taipei, Taiwan, Republic of China, 2 + 521-9321.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 67466; St. James House, Wellington Road North, Stockport, SK4 2RT, England, 61 + 442-7162.

BM

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 624-3276.

CALIFORNIA: Irvine (714) 660-1200;
Sacramento (916) 929-0197;
San Diego (619) 278-9600;
Santa Clara (408) 980-9000;
Torrance (213) 217-7000;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116;
Ft. Lauderdale (305) 973-8502;
Tampa (813) 286-0420.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-3000.

INDIANA: Carmel (317) 573-6400;
Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Baltimore (301) 944-8600.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1500;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600; **Pittsford** (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0930;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Dayton (513) 258-3877.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-6769;
Houston (713) 778-6592; **Richardson** (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

VIRGINIA: Fairfax (703) 849-1400.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8140;
Santa Clara (408) 748-2220;
Torrance (213) 217-7019.

COLORADO: Aurora (303) 368-8000.

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (313) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kierulff Electronics Group
Arrow Canada (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(617) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955;
Hall-Mark (205) 837-8700; Marshall (205) 881-9235;
Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750;
Hall-Mark (602) 437-1200; Marshall (602) 496-0290;
Schweber (602) 997-4874; Wyle (602) 866-2888.

CALIFORNIA: Los Angeles/Orange County:
Arrow/Kierulff (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 716-7300, (714) 669-4100,
(213) 217-8400; Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 999-4702;
(714) 863-0200, (213) 320-8090; Wyle (213) 322-9953,
(818) 880-9000, (714) 863-9953; Zeus (714) 921-9000;
Sacramento: Hall-Mark (916) 722-8600;
Marshall (916) 635-9700; Schweber (916) 929-9732;
Wyle (916) 638-5282;
San Diego: Arrow/Kierulff (619) 565-4800;
Hall-Mark (619) 268-1201; Marshall (619) 578-9600;
Schweber (619) 450-0454; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow/Kierulff (408) 745-6600,
Hall-Mark (408) 432-0900; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444;
Hall-Mark (303) 790-1662; Marshall (303) 451-8383;
Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741;
Hall-Mark (203) 269-0100; Marshall (203) 265-3822;
Schweber (203) 748-7080.

FLORIDA: Ft. Lauderdale:
Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow/Kierulff (305) 725-1480, (305) 682-6923;
Hall-Mark (305) 855-4020; Marshall (305) 767-8585;
Schweber (305) 331-7555; Zeus (305) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399.

GEORGIA: Arrow/Kierulff (404) 449-8252;
Hall-Mark (404) 447-8000; Marshall (404) 923-5750;
Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500;
Hall-Mark (312) 860-3800; Marshall (312) 490-0155;
Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: Indianapolis: Arrow/Kierulff (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483.

IOWA: Arrow/Kierulff (319) 395-7230;
Schweber (319) 373-1417.

KANSAS: Kansas City: Arrow/Kierulff (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002;
Hall-Mark (301) 988-9800; Marshall (301) 840-9450;
Schweber (301) 840-5900; Zeus (301) 997-1118.

TEXAS INSTRUMENTS

MASSACHUSETTS Arrow/Kierulff (617) 935-5134;
Hall-Mark (617) 667-0902; Marshall (617) 658-0810;
Schweber (617) 275-5100, (617) 657-0760;
Time (617) 532-6200; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 971-8220;
Marshall (313) 525-5850; Newark (313) 967-0600;
Schweber (313) 525-8100;
Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800;
Hall-Mark (612) 941-2600; Marshall (612) 559-2211;
Schweber (612) 941-5280.

MISSOURI: St. Louis: Arrow/Kierulff (314) 567-6888;
Hall-Mark (314) 291-5350; Marshall (314) 291-4650;
Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968;
Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900,
(609) 596-8000; GRS Electronics (609) 964-8560;
Hall-Mark (201) 575-4415, (609) 235-1900;
Marshall (201) 882-0320, (609) 234-9100;
Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: Long Island:
Arrow/Kierulff (516) 231-1000; Hall-Mark (516) 737-0600;
Marshall (516) 273-2424; Schweber (516) 334-7555;
Zeus (914) 937-7400;
Rochester: Arrow/Kierulff (716) 427-0300;
Hall-Mark (716) 244-9290; Marshall (716) 235-7620;
Schweber (716) 424-2222;
Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990;
Hall-Mark (216) 349-4632; Marshall (216) 248-1788;
Schweber (216) 464-2970;
Columbus: Arrow/Kierulff (614) 436-0928;
Hall-Mark (614) 888-3313;
Dayton: Arrow/Kierulff (513) 435-5563;
Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537;
Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456;
Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000,
(215) 928-1800; GRS Electronics (215) 922-7037;
Schweber (215) 441-0600, (412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180;
Hall-Mark (512) 258-8848; Marshall (512) 837-1991;
Schweber (512) 339-0088; Wyle (512) 834-9957;
Dallas: Arrow/Kierulff (214) 380-6464;
Hall-Mark (214) 553-4300; Marshall (214) 233-5200;
Schweber (214) 661-5010; Wyle (214) 235-9953;
Zeus (214) 783-7010;
Houston: Arrow/Kierulff (713) 530-4700;
Hall-Mark (713) 781-6100; Marshall (713) 895-9200;
Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913;
Hall-Mark (801) 972-1008; Marshall (801) 485-1551;
Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420;
Marshall (206) 747-9100; Wyle (206) 453-8300.

WISCONSIN: Arrow/Kierulff (414) 792-0150;
Hall-Mark (414) 797-7844; Marshall (414) 797-8400;
Schweber (414) 784-9020.

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 687-4231;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771;
Vancouver: Future (604) 294-1166;
Winnipeg: Future (204) 339-0554.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

Printed in U.S.A.
1604898-9706

SPVU006