TEXAS INSTRUMENTS

TMS340 Graphics Library

SPVU027

Graphics Products

TEXAS INSTRUMENTS

TMS340 Graphics Library User's Guide

SPVU027
August 1990

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.

Texas Instruments products are not intended for use in life-support appliances, devices, or systems. Use of a TI product in such applications without the written consent of the appropriate TI officer is prohibited.

Preface

Read This First

How to Use This Manual

This document contains the following chapters:

Chapter 1 Introduction
Briefly introduces the TMS340 Graphics Library and describes the related software and hardware development tools.

Chapter 2 Getting Started
Contains instructions for installing the graphics library on PC systems, describes the files that are shipped with the library package, provides examples of compiling, assembling and linking C programs that call the library functions, and illustrates the use of the archiver utility with the library.

Chapter 3 Graphics Library Overview
Describes the text and graphics capabilities of the library, the bit-mapped fonts bundled with the library, the library's relationship to the TIGA Core and Extended Primitives, and the system implementation issues involved in porting or extending the library.

Chapter 4 Graphics Operations
Presents the library's conventions regarding coordinate systems, the mapping of pixels to coordinates, operations on pixels, clipping, and the geometric figures and rendering styles supported by the library.

Chapter 5 Bit-Mapped Text
Describes the text capabilities of the library, the types of fonts supported, the internal structure of the database for each font, and an alphabetical listing of the available fonts.

Chapter 6 Core Primitives
Provides a page-by-page alphabetical listing of the TIGA Core Primitives included as part of the graphics library, along with detailed descriptions and programming examples.

Chapter 7 Extended Primitives
Provides a page-by-page alphabetical listing of the TIGA Extended Primitives included as part of the graphics library, along with detailed descriptions and programming examples.

Appendix A Data Structures
Describes the key data structures in the graphics library environment.

Appendix B Reserved Symbols
Lists the global symbols defined within the graphics library.

Appendix C Glossary
Defines the technical terms used in this manual.

Related Documentation

The following documents are available from Texas Instruments:

- *TMS34010 User's Guide* (literature number SPVU001A)

  Describes the internal architecture, hardware interfaces, programmable registers and instruction set of the TMS34010 32-bit graphics processor chip.

- *TMS34020 User's Guide* (literature number SPVU019)

  Describes the internal architecture, hardware interfaces, programmable registers and instruction set of the TMS34020 32-bit graphics processor chip.

- *TMS340 Family Code Generation Tools User's Guide* (literature number SPVU020)

  Describes the C compiler, assembler, linker, and archiver for the TMS340x0 Graphics System Processors.

- *TIGA-340 Interface User's Guide* (literature number SPVU015A)

  Describes the architecture of the TIGA (Texas Instruments Graphics Architecture) software interface between a host processor and a TMS340 graphics processor, which includes the applications interface, communications driver, and graphics manager.

- *TMS340 Family Third Party Guide* (literature number SPVB066C)

  Lists the TMS340x0-based hardware and software products available from third parties.

- *TMS34010 Software Development Board User's Guide* (literature number SPVU002)

  Describes the TMS34010 SDB, which is a PC add-in graphics card that serves as a hardware platform for developing and testing TMS34010 software.

- *TMS34020 Software Development Board User's Guide* (literature number SPVU016)

  Describes the TMS34020 SDB, which is a PC AT add-in graphics card that serves as a hardware platform for developing and testing TMS34020 software.

- *TMS34092 VGA Interface Chip User's Guide* (literature number SPVU026)

  Describes the TMS34092, which is a memory and pixel pipeline peripheral for low-cost PC video adapters with VGA pass-through capability and TMS34010 graphics processing power.

- *TMS34010 CCITT Data Compression Library User's Guide* (literature number SPVU009)

  Describes a library of TMS340x0 assembly-coded functions for compressing and decompressing black-and-white images according to the Consultative Committee for International Telegraph and Telephone Group 3 and Group 4 standards.

- *TMS34010/8514 Adapter Interface Emulation User's Guide* (literature number SPVU010)

  Describes the library of TMS340x0 functions for emulating IBM's 8514/A Adapter Interface.

- *TMS34010 Applications Guide* (literature number SPVA007A)

  Provides examples of hardware and software applications developed for the TMS34010.

- *TMS34010 XDS User's Guide* (literature number SPVU008)

  Describes the XDS (extended development system) for emulating the TMS34010 Graphics System Processor chip in a target hardware environment.

- *TMS34020 XDS User's Guide* (literature number SPVU028)

  Describes the XDS (extended development system) for emulating the TMS34020 Graphics System Processor chip in a target hardware environment.

- *TMS34070 User's Guide* (literature number SPPU016A)

  Describes the TMS34070 16-color palette chip used on the TMS34010 software development board.

- *TMS340 Family Assembler Support for the TMS34082* (literature number SPVU029)

  Summarizes the TMS34082 Floating-Point Processor internal instruction set.

To obtain any of TI's product literature listed above, please contact the Texas Instruments Customer Response Center at toll-free telephone number (800) 232–3200.

You may also find the documents listed below to be helpful. The list is organized according to subject:

TMS34010 and TMS34020 Graphics System Processors

- Asal, Mike, Graham Short, Tom Preston, Derek Roskell, and Karl Guttag. "The Texas Instruments 34010 Graphics System Processor." *IEEE Computer Graphics & Applications*, vol. 6, no. 10 (October 1986), pages 24–39.
- Guttag, Karl, Jerry Van Aken, and Mike Asal. "Requirements for a VLSI Graphics Processor." *IEEE Computer Graphics & Applications*, vol. 6, no. 1 (January 1986), pages 32–47.
- Killebrew, Carrell R., Jr., "The TMS34010 Graphics System Processor." *Byte*, vol. 11, no. 12 (December 1986), pages 193–204.
- Peterson, Ron, Carrell R. Killebrew, Jr., Tom Albers and Karl Guttag. "Taking the Wraps off the 34020. " *Byte*, vol. 11, no. 9 (September 1986), pages 257–272.

The C Programming Language

- American National Standards Institute. *Draft Proposed American National Standard for Information Systems — Programming Language C*. Document no. X3J11/88–002, January 11, 1988.
- Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Second Edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
- Kochan, Stephen G. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, 1983.
- Sobelman, Gerald E., and David E. Krekelberg. *Advanced C: Techniques and Applications*. Indianapolis, Indiana: Que Corporation, 1985.

Computer Graphics Algorithms

- Blinn, James, "How Many Ways Can You Draw a Circle?" *IEEE Computer Graphics and Applications*, vol. 7, no. 8 (August 1987), pages 39–44.
- Bresenham, J. E. "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal*, vol. 4, no. 1 (1965), pages 25–30.
- Bresenham, J. E. "A Linear Algorithm for Incremental Display of Circular Arcs." *Communications of the ACM*, vol. 20, no. 2 (February 1977), pages 100–106.
- Foley, James, and Andries van Dam. *Fundamentals of Computer Graphics*. Reading, Massachussetts: Addison-Wesley, 1982.
- Ingalls, D. H. "The Smalltalk Graphics Kernel." Special issue on Smalltalk. *Byte*, vol. 6, no. 8 (August 1981), pages 168–194.
- Knuth, Donald E., "Digital Halftones by Dot Diffusion." *ACM Transactions on Graphics*, vol. 6, no. 4 (October 1987), pages 245–273.
- Newman, William M., and Robert F. Sproull. *Principles of Interactive Computer Graphics*. 2nd ed. New York: McGraw-Hill, 1979.
- Pike, Rob. "Graphics in Overlapping Bitmap Layers." *ACM Transactions on Graphics*, vol. 2 (April 1983), pages 135–160.
- Pitteway, M. L. V. "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter." *Computer Journal*, vol. 10, no. 3 (November 1967), pages 24–35.
- Porter, T., and T. Duff. "Composing Digital Images." *Computer Graphics, SIGGRAPH Proceedings*, vol. 18, no. 3 (July 1984), pages 253–259.
- Sproull, Robert F., and Ivan E. Sutherland. "A Clipping Divider." Fall Joint Computer Conference, 1968, Thompson Books, Washington, D. C., pages 765–775.
- Van Aken, Jerry R. "An Efficient Ellipse-Drawing Algorithm." *IEEE Computer Graphics & Applications,* vol. 4, no. 9 (September 1984), pages 24–35.
- Van Aken, Jerry R., and Mark Novak. "Curve-Drawing Algorithms for Raster Displays." *ACM Transactions on Graphics,* vol. 4, no. 2 (April 1985), pages 147–169.
- Van Aken, Jerry R., and Carrell R. Killebrew, Jr., "Better Bit-Mapped Lines." *Byte*, vol. 13, no. 3, March 1988, pages 249–253.

Video Memories and Displays

- Conrac Corporation. *Raster Graphics Handbook*. 2nd ed. New York: Van Nostrand Reinhold Company Inc., 1985.
- Pinkham, Ray, Mark Novak, and Karl Guttag. “Video RAM Excels at Fast Graphics.” *Electronic Design*, vol. 31, no. 17 (August 1983), pages 161–182.
- Whitton, Mary C. “Memory Design for Raster Graphics Displays.” *IEEE Computer Graphics & Applications*, vol. 4, no. 3 (March 1984), pages 48–65.

Style and Symbol Conventions

This document uses the following conventions.

- Program listings, program 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:

```
main ( )
{
     set_config(0, 1);
     clear_screen(–1);
     set_text_xy(5, 5);
     text_outp(”hello, world”);
}
```

Trademarks

Aegis, Domain/IX, and Apollo are trademarks of Apollo Computer, Inc.
EGA and VGA are trademarks of IBM Corp.
Macintosh and *MPW* are trademarks of Apple Computer Corp.
MS and *MS-DOS* are trademarks of Microsoft Corp.
NEC is a trademark of NEC Corp.
Sony is a trademark of Sony Corp.
Sun 3, *Sun 4, and Sun Workstation* are trademarks of Sun Microsystems, Inc.
UNIX and *COFF* are trademarks of AT&T Bell Laboratories.
VAX, VMS, and *Ultrix* are trademarks of Digital Equipment Corp.
XDS, TIGA, and *TIGA-340* are trademarks of Texas Instruments, Inc.

Contents

Figures

Tables

Chapter 1

Introduction

The TMS340 Graphics Library is a collection of software functions that execute on the TMS34010 and TMS34020 Graphics System Processors. The functions in the library are designed to be called from C programs executing on a TMS340 graphics processor, but they can also be called from TMS340 assembly language programs that mimic the C compiler's calling conventions. The library provides capabilities for outputting text and graphics to video monitors and other raster graphics display devices controlled by TMS340 graphics processors. Library functions are provided for performing pixel-aligned block transfers (or "PixBlts"), for printing bit-mapped text, and for drawing lines, polygons, ellipses, arcs and other figures.

The TMS34010 and TMS34020 are advanced single-chip, 32-bit graphics microprocessors. They combine 32-bit general-purpose processing power with features that accelerate computer graphics applications and decrease the size and cost of graphics systems. Both are software-compatible members of the TMS340 Family of graphics products from Texas Instruments.

The TMS34010 and TMS34020 are well supported by a full set of hardware and software development tools. The available software tools include a C compiler, assembler, linker and archiver. These are described in the *TMS340 Family Code Generation Tools User's Guide*. Hardware tools include full-speed hardware emulators and IBM PC-compatible software development boards. These are described in the user's guides for the TMS34010 and TMS34020 extended development systems (XDSs) and software development boards (SDBs).

Texas Instruments bundles software debugging tools with its software development board products for the TMS34010 and TMS34020. In addition, a variety of debugging tools are available from third parties. Consult the *TMS340 Family Third Party Guide* for an up-to-date listing of debugging tools from third parties.

Texas Instruments provides a hotline to assist you with technical questions about TMS340 Family products and development tools. The telephone number for the hotline is (713) 274-2340.

1.1 Supported Cross-Development Systems

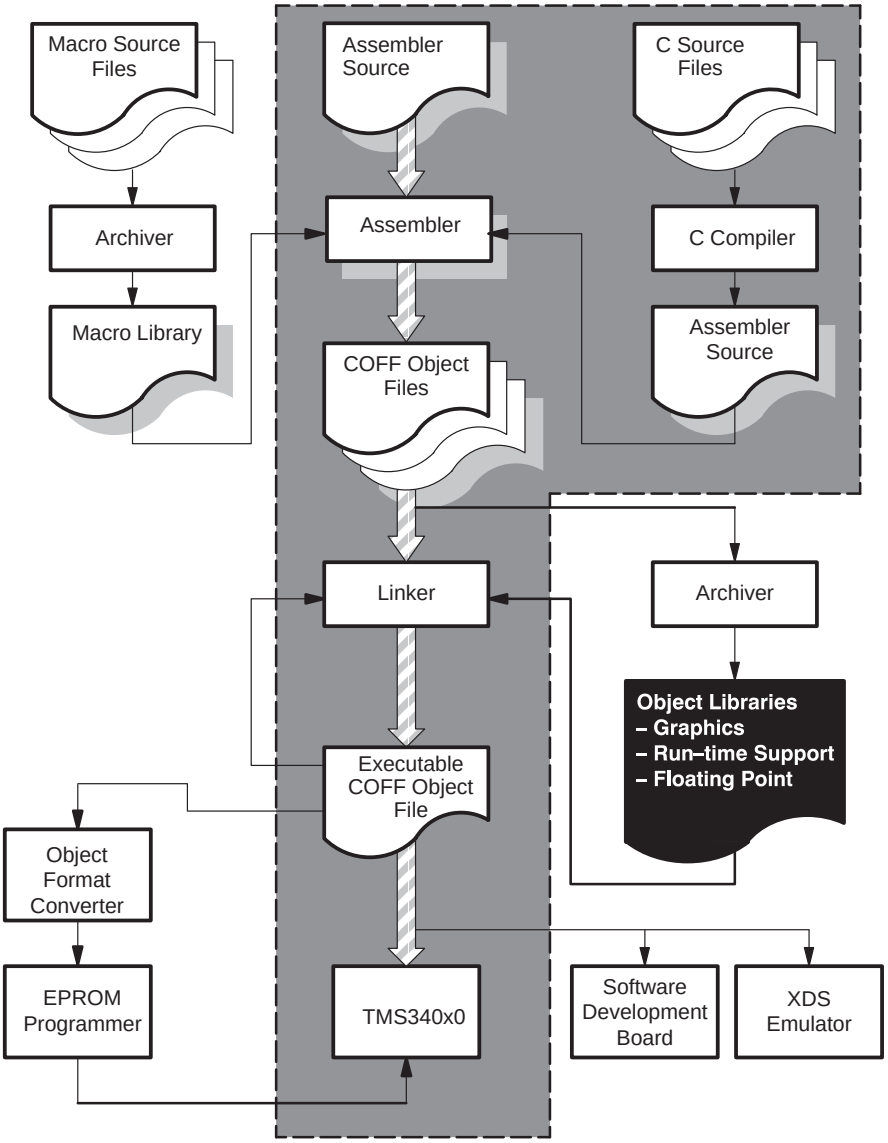
The compilation, assembly, and linking of software modules for execution on a TMS340x0 (TMS34010 or TMS34020) processor is performed on a system other than the target TMS340x0 display system. The TMS340x0 software tools can be installed on the following systems:

- IBM-PC or 100% compatible system
  - with PC-DOS or MS-DOS
  - with OS/2
- VAX
  - VMS
  - Ultrix
- Apollo Workstations
  - Domain/IX
  - AEGIS
- Sun-3 and Sun-4 Workstations with Unix
- Macintosh with MPW

1.2 An Overview of Development Tools

Figure 1–1 shows the flow of TMS340x0 software development with the TMS340 Family Code Generation Tools. The most widely used software development paths are highlighted in the figure; the other paths are optional.

Figure 1–1. TMS340x0 Software Development Flow



The TMS340 Graphics Library (lower right side of Figure 1–1) in object form is linked with applications programs written in C and compiled using the C compiler. (The library can also be linked with applications written in assembly language that mimic the C compiler's subroutine-calling conventions.) The graphics library complements the runtime-support and floating-point libraries shipped with the C compiler. Other libraries available from Texas Instruments as separate products include the CCITT Image Compression/ Decompression Library and the 8514 Adapter Interface Emulation Function Library.

The complete C and assembly language source code for the TMS340 Graphics Library is provided. This makes possible another development path in which the programmer modifies the original library source code to accommodate the requirements of a proprietary application. The modified source library can be compiled, assembled, and archived to form a new object library that can be linked with an application.

The C compiler at the top right of Figure 1–1 accepts C source code and produces TMS340x0 assembly source code. The C compiler consists of four stages: a preprocessor, a parser, a code generator, and an optional optimizer.

The assembler near the top center of Figure 1–1 translates TMS340x0 assembly language source files into machine language object files. The object files are in COFF (common object file format), as described in the *TMS340 Family Code Generation Tools User's Guide*.

The archiver is used to collect a group of files (source or object) into a single archive file. The resulting archive file is referred to as a source or object library, as appropriate.

The linker shown in the middle of Figure 1–1 combines object files into a single executable object module. As the linker creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. The linker is capable of extracting the modules it requires from object libraries constructed using the archiver.

The resulting executable TMS340x0 program can be executed either on a proprietary TMS340x0-based display system or on one of the hardware products provided by Texas Instruments to support software development. The TMS34010 and TMS34020 Software Development Boards from Texas Instruments are high-performance graphics cards that can be inserted into IBM-compatible PCs. The SDBs and accompanying software-debugging tools are sufficient to support the software development needs of many applications. More extensive software and hardware development capabilities are provided in the XDS (extended development system) products for the TMS34010 and TMS34020. For more information, refer to the user's guides for these products.

Chapter 2

Getting Started

This chapter tells you how to install and use the TMS340 Graphics Library. The installation procedures for the IBM PC (or compatible) are described. The procedures for installing the library on VAX, Apollo, Sun and Macintosh systems are similar, but the details are presented in the `readme.1st` documentation files on the distribution media containing the library software.

Installation and testing of the library package should be straightforward if you have either of the following:

1) One of the TMS34010- or TMS34020-based graphics cards supported by Texas Instruments (listed in the next section)

2) A proprietary port of the library from a graphics card manufacturer

If you are planning to port the library yourself to a proprietary card, you may want to begin by checking out the library on a card to which it has already been ported, if one is available to you.

A collection of library-based demonstration programs is included in both source and executable form. You can verify that your graphics card works correctly by running the programs on it. Once this is done, you can verify that your graphics library and the other TMS340 software tools are configured correctly by recompiling and relinking the demos. At this point, you should be ready to begin developing your own application programs, porting the library to a new graphics card, or adding proprietary functions to the graphics library.

Also described in this chapter is the organization of the library files, which are partitioned into several subdirectories. The library directory structure isolates the system-dependent library functions from the system-independent ones. This serves to clearly identify which files need to be modified when the library is ported to a new graphics card. The directory structure also allows the library to be used simultaneously for development with two or more TMS34010- or TMS34020-based graphics cards. For instance, your system might contain both a proprietary graphics card that you are in the process of developing, and a software development board from TI to support initial software development.

The procedure for converting bit-mapped fonts between the TIGA and TMS340 Graphics Library file formats is described at the end of this chapter. Distributed with the graphics library are 108 fonts that have already been converted to the library's format.

2.1 Supported Graphics Cards

At the time of this writing, the distribution media for the TMS340 Graphics Library contain hardware-specific support for the following TMS34010- and TMS34020-based graphics cards from Texas Instruments:

1) TMS34010 TDB (TIGA Development Board) and compatibles. This TMS34010-based PC add-in card contains a TMS34092 VGA Interface Chip, which is a memory and pixel pipeline peripheral for low-cost PC video adapters using the TMS34010. VGA pass-through capability is provided. The current library implementation supports the TDB with a minimum memory configuration of 512 kilobytes of video RAM (and no DRAM). The card is physically socketed for up to 1 megabyte of video RAM and up to 1 megabyte of DRAM. The TDB is software-configurable to drive 640-by-480 and 1024-by-768-resolution analog RGB monitors at 1, 2, 4, and 8 bits/pixel.

2) TMS34010 SDB (Software Development Board). This PC add-in card serves as a hardware platform for developing and testing TMS34010 software. It contains 256 kilobytes of video RAM and 512 kilobytes of DRAM and is configured to drive a 640-by-480-resolution RGB monitor at 4 bits/pixel.

3) TMS34020 SDB (Software Development Board). This PC AT add-in card serves as a hardware platform for developing and testing TMS34020 software. The card provides VGA pass-through capability and can be configured for a data bus width of 8 or 16 bits. On-card memory consists of 1 megabyte of video RAM and 1 megabyte of DRAM, and the video output is software-configured to drive 640-by-480- and 1024-by-768-resolution RGB monitors at 4 and 8 bits/pixel. The card is socketed for an optional TMS34082 Floating-Point Coprocessor.

Ports to additional TMS340x0-based graphics cards may be available by the time you read this. Refer to the documentation files on the library distribution media for updates. You may also wish to consult the TMS340 Graphics Bulletin Board System (telephone number (713) 274-2417) or your graphics card vendor for the latest developments.

The VGA pass-through capability provided by some TMS340 graphics processor cards permits a single monitor to be shared between application programs that run through the TIGA interface and programs that run through the VGA. In a graphics development environment, however, two monitors are virtually a necessity. When debugging a program that runs on a graphics card under either the TIGA or the graphics library environment, one monitor is typically used to display the graphics card's output and a second monitor, driven by the PC display adapter, is used to display the debugger status.

2.2 Installation

To install the TMS340 Graphics Library on the hard disk of your IBM PC or compatible, create a directory named `c:\glib340`. (You can pick another name if you prefer, but you will later have to modify the `makelib.bat` batch file.) Switch to directory `\glib340` and download all the files from the distribution floppy disks to this directory.

Among the downloaded files are several compressed archive files, identified by their `*.zip` file name extensions. The following is a list of the archive files and their contents:

- `corprims.zip` Core Primitives Library functions
- `extprims.zip` Extended Primitives Library functions
- `include.zip` Include files (with `*.h` and `*.inc` extensions)
- `fonts.zip` Bit-mapped fonts (108 fonts in 20 typefaces)
- `tdb10.zip` Support for TMS34010 TIGA Development Board
- `sdb10.zip` Support for TMS34010 Software Development Board
- `sdb20.zip` Support for TMS34020 Software Development Board

The last three archive files above contain hardware-specific support for TMS34010- and TMS34020-based graphics cards from Texas Instruments. If you do not require support for a particular card at this time, you may delete the corresponding `*.zip` file from your hard disk directory. (The file will still be on your distribution floppy disks if you later find that you need it.)

2.2.1 Library Directory Structure

Each of the `*.zip` archive files listed above contains not only files, but also the directory structure to contain the files. When dearchived, each archive file creates a subdirectory to the main library directory, `\glib340`, with the same name as the archive file. For example, the subdirectory contained in archive file `corprims.zip` is `\glib340\corprims`. When you dearchive all the `*.zip` files in the `\glib340` directory, the following directory structure will be created:

`glib340`	Main library directory
`....corprims`	Core Primitives Library
`....extprims`	Extended Primitives Library
`....include`	Include files (`*.h` and `*.inc`)
`....fonts`	Bit-mapped fonts
`....tdb10`	TMS34010 TDB-specific support
`........oemprims`	TMS34010 TDB hardware-dependent functions

`........demos`	Demo programs for TMS34010 TDB
`....sdb10`	TMS34010 SDB-specific support
`........oemprims`	TMS34010 SDB hardware-dependent functions
`........demos`	Demo programs for TMS34010 SDB
`....sdb20`	TMS34020 SDB-specific support
`........oemprims`	TMS34020 SDB hardware-dependent functions
`........demos`	Demo programs for TMS34020 SDB

The hardware-dependent functions contained in the `\oemprims` directories are really part of the Core Primitives Library but are segregated from the hardware-independent core primitives in the `\corprims` directory for convenience. If you port the TMS340 Graphics Library to a proprietary TMS34010- or TMS34020-based graphics card, you will need to modify only the handful of hardware-independent functions in the `\oemprims` directory. The functions in the `\corprims` and `\extprims` directories can be used without modification.

Wherever possible, the graphics library uses the same source files as TIGA. The source files for all library functions that differ in implementation from their TIGA counterparts are isolated in the `\oemprims` directory. The source files for all the library functions contained in the `\corprims` and `\extprims` directories are identical to their TIGA counterparts. However, the `coredefs.c` source file in the `\corprims` directory, which defines the library's global variables *env*, *pattern,* and *sysfont*, differs from the TIGA version of this file, which defines additional variables not used in the library.

The graphics library's `\include` directory contains C and assembly language include files (with `*.h` and `*.inc` file name extensions). Some of these files differ from the TIGA include files with the same names. The files in the `\include` directory are similar to the original TIGA include files but have been edited to eliminate references to TIGA functions and data structures not used in the library. These edits have been made strictly for the sake of clarity, and you should be able to directly replace the library's include files with the original TIGA files if you choose. (The exception to this statement is the `oem.inc` file, which is discussed in Section 2.5.)

The `\fonts` directory contains the 108 bit-mapped fonts that are distributed with the library. These fonts have been converted to the file format required for use with the library. The method for converting files between the TIGA and graphics library font formats is straightforward and is described in Section 2.6.

2.2.2 Dearchiving the Library Files and Subdirectories

The procedure for dearchiving the contents of all the `*.zip` files in the `\glib340` directory is to enter the following single MS-DOS command:

```
for %x in (*.zip) do pkunzip –d –o %x
```

Alternately, if you wish to dearchive the `*.zip` files one at a time, you may do so. For example, to dearchive the Extended Primitives Library (and create the `\extprims` subdirectory), enter the following MS-DOS command:

```
pkunzip –d –o extprims.zip
```

Note that the dearchiving utility `pkunzip.exe` used above is included in the distribution floppy disks containing the library. For a brief explanation of the capabilities of the `pkunzip.exe` utility, enter the MS-DOS command "pkunzip" with no command-line arguments.

2.2.3 Running the Library Demos

Once you have dearchived the subdirectories, you can test your graphics card by running the appropriate set of demonstration programs. The demos for a particular card are contained in the subdirectory for that card. For example, the demos for the TMS34010 TIGA Development Board reside in the subdirectory `\glib340\tdb10\demos`. You can invoke the batch file `demo.bat` in this subdirectory to run the demos for you.

In each `\demos` subdirectory is a copy of the COFF loader utility, `gspl.exe`. The `demo.bat` batch file described in the preceding paragraph uses the loader to download a TMS340x0 executable file in COFF format from disk to the graphics card and execute it. The `gspl.exe` loader is configurable, and each `\demos` directory contains a copy of the loader configured for a particular TMS34010- or TMS34020-based graphics card. The configuration data is stored in a second file, `gspl.ini`, that resides in the same `\demos` subdirectory as the `gspl.exe` file. For more details on the use of the loader, refer to the description in the `gspl.doc` text file in the main library directory, `\glib340`.

2.3 Using and Modifying the Library

You may plan to use the TMS340 Graphics Library in some or all of the following ways:

- ❑ Write your own application program that calls the functions in the library.
- ❑ Port the library to a proprietary graphics card.
- ❑ Write customized graphics functions to be used as extensions to the standard library.

Each of these three usages of the library is discussed, in turn, below.

2.3.1 Writing Your Own Application Program

If you plan to write your own applications program to call the functions in the graphics library, you may want to begin by verifying that your software tools are configured correctly to compile and link programs that run on the TMS340 graphics processor. A convenient way to do this is to compile and link the demonstration programs contained in the `\demos` subdirectory for the target graphics card. The `makedem.bat` batch file that resides in the `\demos` subdirectory is available to automate the process of rebuilding the demos. To update the demos, change to the appropriate `\demos` subdirectory and enter "makedem" at the MS-DOS command line. Running this batch file automatically compiles and links any demos that need to be updated. If you have just installed the graphics library on your system, `makedem.bat` will update all the demos.

The `makedem.bat` batch file invokes the `make.exe` utility program that is distributed with the graphics library. A *make* program is a utility for automating program development. It can automatically update an executable file whenever changes are made to one of its source or object files. For example, if you make an edit to demo source file `test01.c` and run the `makedem.bat` batch file again, the make utility will detect the fact that the `test01.c` file is more recent than the `test01.obj` relocatable object file, and will update this file. It will then detect that the new `test01.obj` file is more recent than the `test01.out` executable file, and update this file as well. Specified as a command-line argument to the make program is the name of a make description file, typically with a `*.mak` file name extension. This file specifies to the make utility which modules are required to update the executable file. The Texas Instruments make utility is similar to the one from Microsoft but has some additional capabilities described in the `make.doc` text file that resides in the main library directory.

The `makedem.bat` batch file, which invokes the make utility, specifies the make description file `demos.mak` as the input file to the make utility. The

demos.mak file in turn refers to the lc.cmd link command file, which directs the linker to the object library archive files that need to be linked with the demo program. These object archives include the corprims.lib, extprims.lib, and oemprims.lib libraries shipped with the TMS340 Graphics Library, and also the rts.lib and flib.lib libraries shipped with the TMS340 Family C Compiler.

At this point, you should be able to create your own C-language program (perhaps by modifying an existing demo program), and compile and link it. If you name the source file "test.c", you can invoke the maketst.bat batch file, which resides in the \demos subdirectory, to automatically compile, link, and execute the program for you. You can verify this by copying one of the demo source files to test.c and entering "maketst" at the MS-DOS command line.

2.3.2 Porting the Library

To port the TMS340 Graphics Library to a proprietary TMS34010- or TMS34020-based graphics card, you will need to modify the hardware-dependent functions in the Core Primitives Library. For convenience, these functions have been isolated in the \oemprims subdirectory corresponding to each of the graphics cards supported in the library. The functions located in the \corprims and \extprims directories do not in general require porting.

The \oemprims directory also isolates all graphics library functions that differ in implementation from their TIGA counterparts. The source code files for the functions in the \corprims and \extprims directories are identical to those distributed in the TIGA interface package.

As an example, the following is a list of the C and assembly-language source code files contained in the \oemprims directory for the TMS34010 TIGA Development Board:

clearfrm.asm	Source code for *clear_frame_buffer* routine
clearpag.asm	Source code for *clear_page* routine
clearscr.asm	Source code for *clear_screen* function
delay.asm	Called indirectly by *set_config* routine
getneare.asm	Source code for *get_nearest_color* routine
nullpatn.asm	Referred to within *set_config* routine
getrev.asm	Called by *set_config* routine
setdptch.asm	Called by *set_config* routine

`trapvect.asm`	Source code for *get_vector and set_vector* routines
`config.c`	Source code for *set_config* routine
`getmode.c`	Source code for *get_modeinfo* routine
`getpalet.c`	Source code for *get_palet and get_palet_entry* routines
`initpale.c`	Source code for *init_palet* routine
`initvide.c`	Called by *set_config* routine
`oem.c`	Data structures accessed by *set_config* routine
`oemdata.c`	Data structures accessed by *set_config* routine
`pagewait.c`	Source code for *page_flip, page_busy,* and *wait_scan* routines
`setpalet.c`	Source code for *set_palet* and *set_palet_entry* routines
`oem.h`	C include file containing system-dependent parameters
`oem.inc`	Assembly-language include file containing system-dependent parameters

The lists for the TMS34010 and TMS34020 SDBs are similar.

The `oem.h` include file defines the hardware-dependent data structure, SETUP. The `oemdata.c` file contains an array of SETUP structures, each of which contains the parameters needed to configure the graphics card in a particular graphics mode. These parameters include screen width and height, pixel size, video timing, display pitch, video page (or frame buffer) addresses, and any card-specific initialization data. The SETUP structure for a proprietary graphics card may need to be customized to accommodate the proprietary features of that card. For instance, the SETUP structure can be modified to contain the initial values loaded into on-card registers.

The `getrev.asm` file listed above contains the `getrev` routine that is called by the library function `set_config` to obtain the silicon revision number of the TMS34010 or TMS34020 processor. Both the TIGA and TMS340 Graphics Library versions of `getrev` obtain the revision number by executing the assembly language instruction REV. The graphics library version of this file, however, differs from TIGA in that it also contains a default illegal opcode interrupt service routine (ISR) that is installed at the time the program is loaded into TMS340 graphics processor memory. The primary purpose of this ISR is to support the earliest versions of the TMS34010, which treat REV as an illegal opcode. If your graphics card contains a TMS34020, or a later version of the TMS34010 that recognizes REV as a valid instruction, you can delete the ISR from the `getrev.asm` file if you wish. (At the time of this writing, one potential reason for removing the ISR is that not all

TMS34010 and TMS34020 debuggers understand how to deal intelligently with an ILLOP trap vector installed at load time.) If you delete the ISR, you will also need to modify the `lc.cmd` link command file that resides in the `\demos` directory. As shipped with the library, `lc.cmd` is set up to install the ILLOP trap vector at the time a library-based application is loaded into TMS340 graphics processor memory. Refer to the SECTIONS directive at the end of `lc.cmd`.

2.3.3 Developing Custom Graphics Functions

For the benefit of programmers who wish to modify the existing graphics functions or write their own custom functions, the TMS340 Graphics Library is distributed in both source and object form. Before undertaking development of new functions, you may wish to verify that the software tools are configured correctly by recompiling the object library files from the existing source files. The `makelib.bat` batch file in the main library directory, `\glib340`, automates the process of remaking the object libraries. This batch file creates the following object library archive files:

`corprims.lib`	Resides in `\corprims` subdirectory.
`extprims.lib`	Resides in `\extprims` subdirectory.
`oemprims.lib`	Resides in `\oemprims` subdirectory for specified target graphics card.

Each archive file is built using the `gspar.exe` archive utility program described in the *TMS340 Family Code Generation Tools User's Guide*. The `makelib.bat` batch file also recompiles and relinks the demo programs for the target graphics card. Note that the core and extended primitives contained in the `\corprims` and `\extprims` subdirectories are hardware-independent and run without modification on any of the graphics cards currently supported by the library. The hardware-specific core primitives are contained in the `\oemprims` directory for each supported card.

The `makelib.bat` batch file can be invoked from the main library directory, `\glib340`, with an MS-DOS command-line argument specifying the target graphics card. For example, to configure the library to run on the TMS34010 TIGA Development Board, the MS-DOS command is

```
makelib tdb10
```

If you intend to configure the library for one of the other supported graphics cards, you can obtain instructions on how to do this by entering the MS-DOS command "makelib" with no command-line arguments.

Once the object library has been remade by the `makelib.bat` batch file, it can be updated to reflect changes in the source files. For example, if you

edit the `cpw.asm` source file in the `\corprims` directory and invoke `makelib.bat` a second time, the batch file will detect that `cpw.asm` has been updated and will update the corresponding object file, `cpw.obj`, contained in the `corprims.lib` object archive. (None of the other object files will be updated unless they need to be.)

The `makelib.bat` batch file invokes the `make.exe` utility program, described previously. Each of the object archives—`corprims.lib`, `extprims.lib`, and `oemprims.lib`—is built by a make description file with a `*.mak` filename extension that resides in the same subdirectory as the object archive file. You may want to use an existing make description file as a model for writing a make description file to build your proprietary object library.

2.4 Symbolic Debugging

If you have access to a symbolic debugger, you may wish to try stepping through one or more of the demo programs provided with the graphics library. As distributed, the executable demo programs contain embedded symbolic information for the demo code itself, but not for the graphics library functions they are linked with.

The object libraries `corprims.lib`, `extprims.lib`, and `oemprims.lib` are distributed without embedded symbols in order to reduce the storage requirements of the distributed package. If you have a symbolic debugger and you need to access the symbols for one or more library functions, you can recompile the source code for those functions with the symbols option enabled and update the object libraries. The procedure for retaining symbolic information during the compilation, assembly, and linking stages is described in the *TMS340 Family Code Generation Tools User's Guide*.

If you are releasing a software product based on the graphics library, you should be aware that including embedded symbols in the object or executable files may significantly increase their storage requirements on disk and increase the time required to download the code to the TMS340 graphics processor. (The symbolic information is removed by the COFF loader, and it therefore has no impact either on the size of the code downloaded to the TMS340 graphics processor for execution, or on the execution speed of the code.)

2.5 TMS34010 and TMS34020 Code Compatibility

The core and extended primitives contained in the `\corprims` and `\extprims` directories are, by default, configured to generate TMS34010-compatible code. Note that TMS34010-compatible code also runs on the TMS34020, although it does not take advantage of TMS34020 graphics acceleration features not available on the TMS34010.

If you desire, you can configure the core and extended primitives to utilize the TMS34020's graphics acceleration features, but the library thus configured will not execute on the TMS34010.

The mechanism for configuring the library involves conditional assembly constants GSP_34010 and GSP_34020, defined in the include file `oem.inc`, which resides in the `\include` directory. By default, this file defines GSP_34010 and GSP_34020 as having the values 1 and 0, respectively, which enables TMS34010-compatible code. (This code also runs on the TMS34020.) To configure the library for TMS34020-specific code, edit the `oem.inc` file to define GSP_34010 and GSP_34020 as having the values 0 and 1, respectively, and recompile the library using the `makelib.bat` batch file described previously. The resulting object code will execute correctly on the TMS34020, but not on the TMS34010.

The `\oemprims` directories for the various supported graphics cards also contain `oem.h` and `oem.inc` files that define conditional assembly constants GSP_34010 and GSP_34020. As distributed, these files should already define the two constants correctly to accommodate the target graphics card, and you do not need to modify them. However, if you are porting the library to a proprietary graphics card, you should know that the role of these constants in the `\oemprims` directory is slightly different from that in the `\corprims` and `\extprims` directories. In the case of the `\oemprims` directory, the two constants are specific to the processor on the target graphics card. In other words, if the graphics card contains a TMS34010, the constants GSP_34010 and GSP_34020 in the `oem.h` and `oem.inc` files must be defined as 1 and 0, respectively. If the graphics card contains a TMS34020, the constants GSP_34010 and GSP_34020 must be defined as 0 and 1, respectively. Hardware-specific functions configured to execute on the TMS34010 will not run correctly on the TMS34020, and *vice versa*.

2.6 Conversion Between TIGA and Library Font Formats

The bit-mapped fonts distributed with the TMS340 Graphics Library reside in the `\fonts` directory. Each font archive file is identified by its `*.lib` extension; each archive file contains all the fonts in a particular typeface. The archive is built with the `gspar.exe` archive utility program described in the *TMS340 Family Code Generation Tools User's Guide*.

As described in Chapter 1, these fonts are identical to those distributed with the TIGA-340 Interface package, although the font file format differs from that of TIGA. Each TIGA font is distributed as a binary image file with a `*.fnt` file name extension. Each font in the TMS340 Graphics Library is distributed as a file in COFF format with an `*.obj` file name extension. The conversion between the TIGA and graphics library formats is straightforward with the `binsrc.exe` and `cof2bin.exe` conversion utilities that are distributed with the library and reside in the `\fonts` subdirectory. For more information on these utilities, refer to the `binsrc.doc` and `cof2bin.doc` documentation files in the `\fonts` subdirectory.

For example, to convert the *austin25* font from graphics library format to TIGA format, use the following MS-DOS command to extract the `austin25.obj` file from the `austin.lib` font archive with the `gspar.exe` utility:

```
gspar –x austin.lib austin25.obj
```

Then use this MS-DOS command to convert the `austin25.obj` file, in COFF format, to binary format with the `cof2bin.exe` utility:

```
cof2bin austin25.obj austin25.fnt
```

where the output file, `austin25.fnt`, is in the binary format used by TIGA.

To continue the example, convert the TIGA font file `austin25.fnt` to the graphics library format in two steps. First, use the `binsrc.exe` utility to convert the TIGA font file to a TMS340 assembly language source file with the following MS-DOS command:

```
binsrc –a austin25.fnt austin25.asm
```

If you inspect the resulting source file, `austin25.asm`, you will notice that the label assigned to the font data structure is *austin25*, which is the file name as well. By convention, the file name (and the label) matches the global name assigned to this font within the graphics library. To convert from assembly source to COFF format, invoke the TMS340 assembler in the usual manner:

```
gspa austin25.asm austin25.obj
```

The resulting output file, `austin25.obj`, is in the COFF format used for the fonts distributed with the graphics library.

Chapter 3

Graphics Library Overview

The TMS340 Graphics Library is a collection of software functions for drawing text and graphics on a bit-mapped display controlled by a TMS340 Family Graphics System Processor. The library represents a subset of the core and extended primitives available to applications running under the TIGA-340 Interface environment, as described in the *TIGA-340 Interface User's Guide*. The TMS340 Graphics Library package currently supports two software-compatible TMS340 graphics processors, the TMS34010 and TMS34020. Full source code is provided for all library functions. Also distributed with the library are a number of demonstration programs and a collection of bit-mapped fonts. The library can easily be ported to run on display systems spanning a broad range of display resolutions and pixel depths.

The graphics library is intended to be used with Release 4.00 or above of the TMS340 Family Code Generation Tools from Texas Instruments, as described in the *TMS340 Family Code Generation Tools User's Guide*. These software development tools include a C compiler, assembler, linker, archiver, and additional utilities.

The library, as distributed, is configured to run on several TMS34010- and TMS34020-based graphics boards, including the TMS34010 and TMS34020 Software Development Boards available from Texas Instruments. An up-to-date list of display hardware to which the library has already been ported is available in the documentation files on the magnetic media on which the library is distributed. Porting the library to additional TMS340 graphics processor-based displays is straightforward; the system implementation issues involved in porting or extending the library are addressed later in this chapter.

The TMS340 Graphics Library contains all of the extended primitives distributed with the TIGA-340 Interface package, but only a subset of the TIGA core primitives. This is due to the differences between the TIGA environment and that of the graphics library. TIGA provides a convenient interface for dividing processing tasks between the TMS340 graphics processor and a host processor. Graphics applications executing through TIGA can improve their performance by utilizing the processing power of the host and

the TMS340 graphics processor together in parallel. Refer to the *TIGA-340 Interface User's Guide* for details.

In contrast to TIGA, the TMS340 Graphics Library is designed to support the development both of applications that reside completely on the TMS340 graphics processor and of applications that rely on proprietary software interfaces for TMS340 graphics processor communications with other processors.

The environment of the TMS340 Graphics Library is simpler than TIGA's in two respects:

1) TIGA executes on two processors, the host and the TMS340 graphics processor, which operate in parallel. The graphics library, on the other hand, executes on the TMS340 graphics processor alone.

2) TIGA utilizes interrupts, whereas the graphics library does not. Where necessary, graphics library functions poll interrupt requests, but the interrupts are disabled.

The TMS340 Graphics Library offers the following benefits:

- ❑ Convenient access to TMS340 graphics processor capabilities and performance for evaluation by potential developers
- ❑ A simple, single-processor environment for learning about and prototyping with the TMS340 graphics processor
- ❑ Working examples of graphics functions written in assembly code for the TMS340 graphics processor
- ❑ An easily portable software package for shaking out new TMS340x0-based hardware designs
- ❑ Library routines that can be utilized as defined, or adapted to serve proprietary needs
- ❑ A library of graphics functions that run independently of the TIGA dual-processor environment
- ❑ A simplified environment for initial development and debugging of software that executes under the TIGA Graphics Manager

Regarding the last item above, developers of proprietary graphics extensions to the standard TIGA primitives may find the simpler environment of the TMS340 Graphics Library to be more convenient for the initial testing and debugging of customized code. Once a proprietary graphics function has been successfully tested within the library environment, it can be subjected to further testing within the more complex TIGA environment. Porting a function from the graphics library to TIGA is in most cases trivial because of the similarity of the two graphics environments.

3.1 Graphics Capabilities

The functions in the TMS340 Graphics Library are designed to execute over a wide range of display resolutions and pixel depths. The range of screen resolutions supported by the library spans the available raster display technology. The pixel sizes supported by the library are 1, 2, 4, 8, 16, and 32 bits.

The library achieves this level of display independence by exploiting the inherent graphics capabilities of the TMS34010 and TMS34020 graphics processors. Software executing on a TMS340 processor can configure display parameters such as display dimensions and pixel size by simply loading the parameters into dedicated hardware registers. The processor's graphics instructions automatically make the adjustments necessary to accommodate the parameters of the display.

By default, library functions that output graphics to the display simply overwrite destination pixels with source pixels. The library, however, supports several optional methods for combining source and destination pixel values to produce the final pixel values written to the screen:

- A variety of Boolean and arithmetic operations are supported for combining source and destination pixels.
- Designated bits within pixels can be masked off to protect the bits against modification during writes.
- Objects written to the screen can contain transparent regions through which the original background is visible.

The three capabilities above are not mutually exclusive. They can be configured independently by calls to library functions, and used in any combination when drawing to the display.

The library has been carefully designed to make the behavior of the graphics functions predictable in all cases. The library follows well-defined conventions for mapping pixels to x-y coordinates, for determining which pixels are contained within the boundaries of filled regions, and for selecting thin, but connected, sets of pixels to approximate lines and arcs.

All graphics output is automatically clipped to the boundaries of the display. A clipping window can be defined that restricts graphics output to a rectangular region of the display.

3.2 Core and Extended Primitives

The TMS340 Graphics Library is divided into two sub-libraries, the Core Primitives Library and the Extended Primitives Library. TIGA similarly divides the library functions into a set of core primitives that are permanently installed as part of the Graphics Manager, and a set of extended primitives that can be installed as an extension to the core primitives. The distinction between core and extended primitives is of less importance in the TMS340 Graphics Library environment than in TIGA but is maintained for the sake of uniformity with the TIGA environment and documentation.

Within the graphics library environment, the differences between core and extended primitives are primarily functional. The core primitives are, in general, dedicated to initializing, configuring, and interrogating the graphics environment but provide only rudimentary capabilities for drawing to the display. The extended primitives provide a broader range of text and graphics output capabilities, including the abilities to draw text in a variety of proportionally spaced fonts and to draw graphics objects such as lines, ellipses, arcs and polygons.

Table 3–1 is a comprehensive, alphabetical listing of the functions available in the TMS340 Graphics Library. The rightmost column identifies the function as belonging to either the core or extended primitives. The Core Primitives and Extended Primitives Libraries are described in Chapters 6 and 7 of this user's guide as separate, but related libraries.

Table 3–1. Summary of Library Functions

Function Name	Description	Type
bitblt	Transfer bit-aligned block	Ext
clear_frame_buffer	Clear frame buffer	Core
clear_page	Clear current drawing page	Core
clear_screen	Clear screen	Core
cpw	Compare point to clipping window	Core
cvxyl	Convert x-y position to linear address	Core
decode_rect	Decode rectangular image	Ext
delete_font	Delete font	Ext
draw_line	Draw line	Ext
draw_oval	Draw oval	Ext
draw_ovalarc	Draw oval arc	Ext
draw_piearc	Draw pie arc	Ext
draw_point	Draw point	Ext
draw_polyline	Draw polyline	Ext
draw_rect	Draw rectangle	Ext
encode_rect	Encode rectangular image	Ext
field_extract	Extract field from TMS340 graphics processor memory	Core
field_insert	Insert field into TMS340 graphics processor memory	Core
fill_convex	Fill convex polygon	Ext
fill_oval	Fill oval	Ext
fill_piearc	Fill pie arc	Ext
fill_polygon	Fill polygon	Ext
fill_rect	Fill rectangle	Ext
frame_oval	Fill oval frame	Ext
frame_rect	Fill rectangular frame	Ext
get_colors	Get colors	Core
get_config	Get hardware configuration information	Core
get_env	Get graphics environment information	Ext
get_fontinfo	Get font information	Core
get_modeinfo	Get graphics mode information	Core
get_nearest_color	Get nearest color	Core

Table 3–1. Summary of Library Functions (Continued)

Function Name	Description	Type
get_offscreen_memory	Get off-screen memory	Core
get_palet	Get entire palette	Core
get_palet_entry	Get single palette entry	Core
get_pixel	Get pixel	Ext
get_pmask	Get plane mask	Core
get_ppop	Get pixel processing operation code	Core
get_textattr	Get text attributes	Ext
get_text_xy	Get text x-y position	Core
get_transp	Get transparency flag	Core
get_vector	Get trap vector	Core
get_windowing	Get window clipping mode	Core
get_wksp	Get workspace information	Core
gsp2gsp	Transfer from one location to another within TMS340 graphics processor memory	Core
in_font	Verify characters in font	Ext
init_palet	Initialize palette	Core
init_text	Initialize text	Core
install_font	Install font	Ext
lmo	Find leftmost one	Core
move_pixel	Move pixel	Ext
page_busy	Get page busy status	Core
page_flip	Flip display and drawing pages	Core
patnfill_convex	Fill convex polygon with pattern	Ext
patnfill_oval	Fill oval with pattern	Ext
patnfill_piearc	Fill pie arc with pattern	Ext
patnfill_polygon	Fill polygon with pattern	Ext
patnfill_rect	Fill rectangle with pattern	Ext
patnframe_oval	Fill oval frame with pattern	Ext
patnframe_rect	Fill rectangular frame with pattern	Ext
patnpen_line	Draw line with pen and pattern	Ext
patnpen_ovalarc	Draw oval arc with pen and pattern	Ext
patnpen_piearc	Draw pie arc with pen and pattern	Ext
patnpen_point	Draw point with pen and pattern	Ext

Table 3–1. Summary of Library Functions (Continued)

Function Name	Description	Type
patnpen_polyline	Draw polyline with pen and pattern	Ext
peek_breg	Peek at B-file register	Core
pen_line	Draw line with pen	Ext
pen_ovalarc	Draw oval arc with pen	Ext
pen_piearc	Draw pie arc with pen	Ext
pen_point	Draw point with pen	Ext
pen_polyline	Draw polyline with pen	Ext
poke_breg	Poke value into B-file register	Core
put_pixel	Put pixel	Ext
rmo	Find rightmost one	Core
seed_fill	Seed fill	Ext
seed_patnfill	Seed fill with pattern	Ext
select_font	Select font	Ext
set_bcolor	Set background color	Core
set_clip_rect	Set clipping rectangle	Core
set_colors	Set foreground and background colors	Core
set_config	Set hardware configuration	Core
set_draw_origin	Set drawing origin	Ext
set_dstbm	Set destination bit map	Ext
set_fcolor	Set foreground color	Core
set_palet	Set multiple palette entries	Core
set_palet_entry	Set single palette entry	Core
set_patn	Set fill pattern	Ext
set_pensize	Set pen size	Ext
set_pmask	Set plane mask	Core
set_ppop	Set pixel processing operation code	Core
set_srcbm	Set source bit map	Ext
set_textattr	Set text attributes	Ext
set_text_xy	Set text x-y position	Core
set_transp	Set transparency mode	Core
set_vector	Set trap vector	Core
set_windowing	Set window clipping mode	Core
set_wksp	Set workspace information	Core

Table 3–1. Summary of Library Functions (Concluded)

Function Name	Description	Type
styled_line	Draw styled line	Ext
styled_oval	Draw styled oval	Ext
styled_ovalarc	Draw styled oval arc	Ext
styled_piearc	Draw styled pie arc	Ext
swap_bm	Swap source and destination bit maps	Ext
text_out	Output text	Core
text_outp	Output text at current x-y position	Core
text_width	Get width of text string	Ext
transp_off	Turn transparency off	Core
transp_on	Turn transparency on	Core
wait_scan	Wait for scan line	Core
zoom_rect	Zoom rectangle	Ext

3.3 Differences Between TIGA and TMS340 Graphics Library Routines

The list of functions in Table 3–1 is a subset of the functions available in the TIGA environment. Not included in the TMS340 Graphics Library are the TIGA functions for managing host-to-TMS340 graphics processor communications, interrupts, cursors, dynamic linking, and dynamic memory management. The TIGA memory management functions are distinguished by their names from the host processor's ANSI-standard C memory management functions. Because of the single-processor environment of the TMS340 Graphics Library, however, library-based software applications can simply call the standard memory management functions distributed with the C compiler for the TMS340 graphics processor.

The source code for the following six functions differs between the TIGA and the TMS340 Graphics Library implementations:

- *get_config*
- *get_modeinfo*
- *page_busy*
- *page_flip*
- *set_config*
- *wait_scan*

The *get_config* function retrieves system configuration information in the form of a CONFIG data structure. The TIGA version of the function includes the size of TIGA's communications buffer as part of the CONFIG structure. The buffer size is obtained from a global communications structure that is defined in TIGA's environment but not in the TMS340 Graphics Library's environment. The graphics library version of *get_config* sets the communications buffer size in the CONFIG structure to *null*.

The *get_modeinfo* function returns a block of information characterizing the designated graphics mode in the form of a MODEINFO data structure. The TIGA version implements this function entirely on the host processor, whereas the graphics library version runs entirely on the TMS340 graphics processor.

The *set_config* function configures the display hardware and initializes the graphics software environment. Typically, this function should be called before any of the other library functions are called. The TIGA and graphics library versions of *set_config* differ somewhat because of the differences between the TIGA and graphics library environments. For example, the TIGA version of *set_config* initializes the cursor management parameters; the graphics library does not include cursor management functions.

The *page_flip*, *page_busy,* and *wait_scan* routines in TIGA and the graphics library are functionally equivalent but differ in the way that they respond

to display interrupt requests. The TIGA implementations rely on an interrupt service routine invoked by the TMS340 graphics processor's display interrupt. The graphics library implementations of these functions poll the display interrupt request and assume that the display interrupt is disabled.

3.4 Graphics Library Environment

The global variables maintained within the TMS340 Graphics Library are a subset of those maintained within the TIGA Graphics Manager, which is the TMS340 graphics processor-resident portion of the TIGA software interface. These variables are accessible by the functions within the library. If developers add proprietary functions to the library, the new functions can access the library's global variables as well.

A list of global variables is presented in Table 3–2. The corresponding type definitions for the variables are given in Appendix A.

Table 3–2. Library Global Variables

Type	Global Name	Description
CONFIG	config	Current configuration
PALET	DEFAULT_PALET[16]	Default color palette
ENVIRONMENT	env	Graphics environment
ENVTEXT	envtext	Text environment
MODEINFO	*modeinfo	Graphics mode information
OFFSCREEN_AREA	*offscreen	List of off-screen buffers
PAGE	*page	List of video pages
PALET	palet[ ]	Palette currently in use
PATTERN	pattern	Current area-fill pattern
FONT	*sysfont	System font

TMS340 graphics processor-based applications linked with the TMS340 Graphics Library can access these globals directly or, if emulation of the TIGA applications environment is important, indirectly. Host-resident applications running through TIGA have no direct access to the global variables on the TMS340 graphics processor. TIGA provides indirect access to the globals in the TMS340 processor's environment through functions such as *get_config*, *get_fontinfo,* and *get_env.* The same functions provided by TIGA to indirectly access the TMS340 graphics processor environment are available in the TMS340 Graphics Library.

The entire graphics environment is initialized by the *set_config* function, which must be called before any of the other functions in the graphics library are called. The initial call to *set_config* should specify a nonzero value for the second argument; this causes the drawing environment to be initialized. Refer to the description of the *set_config* function in Chapter 6 for more information.

3.5 Bit-Mapped Fonts

A total of 108 bit-mapped fonts are distributed with the TMS340 Graphics Library for use with the text functions in the library. The font package distributed with the library consists of 20 different typefaces, each of which is available in a variety of font sizes.

Table 3–3 is a list of the fonts distributed with the library. Most of the typefaces are proportionally spaced, although a few have uniform horizontal spacing. The *global symbol* is the external name of the font, and the *xx* appended to each symbol is replaced with the font size. For instance, global symbol *arrows25* refers to Arrows font size 25.

Table 3–3. Summary of Available Fonts

Face Name	Global Symbol	Number of Font Sizes	Horizontal Spacing
Arrows	arrows*xx*	2	uniform
Austin	austin*xx*	6	proportional
Corpus Christi	corpus*xx*	5	uniform
Devonshire	devons*xx*	3	proportional
Fargo	fargo*xx*	3	proportional
Galveston	galves*xx*	6	proportional
Houston	houstn*xx*	6	proportional
Luckenbach	lucken*xx*	1	proportional
Math	math*xx*	6	proportional
San Antonio	sanant*xx*	3	proportional
System	sys*xx*	2	block
Tampa	tampa*xx*	4	proportional
TI Art Nouveau	ti_art*xx*	5	proportional
TI Bauhaus	ti_bau*xx*	9	proportional
TI Cloister	ti_clo*xx*	2	proportional
TI Dom Casual	ti_com*xx*	5	proportional
TI Helvetica	ti_hel*xx*	12	proportional
TI Park Avenue	ti_prk*xx*	8	proportional
TI Roman	ti_rom*xx*	12	proportional
TI Typewriter Elite	ti_typ*xx*	8	uniform

Two VGA-style *block* fonts are provided for emulating cell-mapped text generated by a CRT controller with a character ROM. These are the System fonts listed near the middle of Table 3–3.

The fonts in Table 3–3 are identical to the ones distributed with the TIGA-340 Interface package. In the case of TIGA, the fonts are stored as binary files with `.fnt` file-name extensions. The `.fnt` files can be downloaded to buffers in the TMS340 graphics processor's local memory at runtime. In the TMS340 Graphics Library, the fonts are distributed as COFF object files with `.obj` file-name extensions that can be linked with programs that execute on the TMS340 graphics processor.

The conversion between the TIGA `.fnt` and graphics library `.obj` formats is straightforward using the *binsrc* and *cof2bin* utilities discussed in Section 2.6.

3.6 Application Programming Issues

Some of the issues that may be of interest to you if you are planning to write application programs based on the TMS340 Graphics Library are discussed below. These issues include source code portability, stack growth, and library code size.

3.6.1 Specifying Complete Argument Lists

As a general rule—and in particular, to ensure portability to the TIGA applications environment—the application program should explicitly specify all arguments to library functions, even if some of those arguments are ignored.

For example, the *set_dstbm* (set destination bit map) library function accepts five arguments. If the first argument is 0, however, the function ignores the values of the remaining four arguments. The recommended practice in a case such as this is to specify all arguments, although dummy values can, of course, be used for those arguments that are ignored:

```
set_dstbm(0, 0, 0, 0, 0);
```

While a function call such as

```
set_dstbm(0);   /*wrong*/
```

that omits the last four arguments may execute correctly in some instances, this approach is discouraged for the sake of broader portability.

3.6.2 Library Globals

As described in Section 3.4, if emulation of the TIGA applications environment is seen as important, the global variables defined within the library should not be accessed directly by application programs. Both TIGA and the TMS340 Graphics Library provide the same functions for accessing the contents of these variables indirectly.

3.6.3 Portability of C Source Code

Some forethought may be required to ensure that C code written for the TMS340x0 is portable. This may be an issue, for instance, to an applications programmer who plans eventually to port C routines developed in the TMS340 Graphics Library environment to TIGA.

As an example, the TMS340 Family C Compiler defines an integer of type *int* as 32 bits. A TIGA application program, however, may be compiled to run on a PC by the Microsoft C Compiler, which defines an *int* as 16 bits. Fortunately, both the TMS340 and Microsoft compilers define types *short* and *long* as 16 and 32 bits, respectively. Specifying all integers as type *short*

or *long* greatly enhances portability between the two environments. This is the main reason that TIGA (and the TMS340 Graphics Library) specifies all integer function arguments, return values, and structure members as being of type *short* or *long*, rather than of type *int*.

3.6.4 Stack Growth

Probably the worst potential offenders in the library in regard to stack growth are the *fill_polygon* and *patnfill_polygon* functions. These functions allocate temporary storage on the system stack, and the amount of storage increases with the number of edges in the polygon. The actual amount of stack space allocated for each edge is 16 bytes. If the number of edges is quite large, the stack may overflow.

At the time of this writing, the size of the default stack allocated by the TMS340 Family C Compiler is 4000 bytes. This should be sufficient to handle polygons having in excess of 200 edges each.

3.6.5 Library Code Size

The TMS340 Family Linker is intelligent enough to link in only the object modules it requires from the library archive files. Some customers, however, ask how large the object code is for the entire library. Presumably, they plan to place the entire library in ROM.

The precise memory requirement varies from one system to another because of differences in the size of the system-dependent code and data. At the time of writing, the TMS340 Graphics Library occupies roughly 35 kilobytes of TMS340 graphics processor memory in machine code form. This figure includes all routines in the library and other data such as graphics mode initialization structures—but excludes the fonts. The breakdown for the current port of the library to the TMS34010 Software Development Board is as follows:

function code	= 25.6	kilobytes
other data	= 9.3	kilobytes
TOTAL	= 35.0	kilobytes

As you might expect, the 108 bit-mapped fonts take up a lot of storage:

bit-mapped fonts = 729.2 kilobytes

The worst memory hogs are the fonts with the largest point sizes. If memory space requirements are tight, you may need to select with care the fonts you use.

3.7 System Implementation Issues

The TMS340 Graphics Library can be easily customized for proprietary software applications and hardware systems based on the TMS340 graphics processor. This section explains how to port the library to proprietary hardware systems, add proprietary functions, or reconfigure the library for other purposes. These topics are covered:

1) Register Usage Conventions
2) Interrupts
3) System-Level Hardware Functions
4) Functions with System Dependencies
5) TMS34010 and TMS34020 Code Compatibility
6) Floating-Point Compatibility
7) Silicon Revision Number

3.7.1 Register Usage Conventions

To use proprietary assembly language functions in conjunction with the library functions, follow the library's conventions regarding usage of the TMS340 graphics processor's registers.

When an assembly language routine is called from a program compiled with the TMS340 Family C Compiler, certain registers are in known states. Here are descriptions of those known states:

- System Stack Pointer

  The SP points to the top of the system stack.

- Status Register

  The C environment always leaves the field-1 sign and extension parameters in the status register defined as follows:

  - FE1 = 0 (field 1 is zero-extended)
  - FS1 = 0 (field 1 is 32 bits in length)

  The field-0 parameters FE0 and FS0 are undefined.

- A-File Registers

  Register A14 points to the top of the C program stack.

- B-File Registers

  DPTCH—Contains screen pitch (difference in starting memory addresses of any two successive scan lines in display memory).

OFFSET—Contains memory address of pixel at top left corner of screen.

WSTART—Contains screen coordinates of pixel at top left corner of current clipping window.

WEND—Contains screen coordinates of pixel at bottom right corner of current clipping window.

COLOR0—Contains current background color (pixel-replicated to 32 bits).

COLOR1—Contains current foreground color (pixel-replicated to 32 bits).

- I/O Registers

  CONTROL—The PPOP field contains the current pixel processing code. The T field is a 1 if transparency is enabled, and 0 otherwise. The W field is always set to 3 (clip to window, no WV interrupt request). The PBH and PBV bits are always 0.

  CONVDP—Corresponds to screen pitch in DPTCH register.

  PMASK—Contains current plane mask (pixel-replicated to 16 bits for a TMS34010 and to 32 bits for a TMS34020).

  PSIZE—Contains current pixel size for screen.

The above assumptions apply to functions called from C programs. They do not apply to interrupt service routines, because such routines may interrupt a function that is using one of these registers for another purpose.

Prior to returning, a function called from a program compiled with the TMS340 Family C Compiler must restore the following registers to their original state at entry:

- 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. Register A14, the C program stack pointer, must be updated to point to the top of the current program stack. Refer to the description of the function-calling conventions in the *TMS340 Family Code Generation Tools User's Guide*.
- In general, all B-file registers must be restored. Certain B-file registers, however, may be altered by attribute control functions. For instance, the *set_colors* function alters the contents of B8 (COLOR0) and B9 (COLOR1), *set_clip_rect* alters B5 (WSTART) and B6 (WEND), and *page_flip* alters B4 (OFFSET).
- In general, I/O registers such as CONTROL, DPYCTL, CONVDP, and INTENB should be restored before returning to the calling program.

The contents of certain I/O registers, however, may be altered by attribute control functions. For instance, the *set_ppop* function alters the PPOP field in CONTROL, the *transp_on* and *transp_off* functions alter the state of the T bit in CONTROL, and *set_pmask* alters the contents of PMASK. As a rule, these registers are not modified by graphics output functions.

Take care when you write graphics routines that modify only bits 5–14 of the TMS340 graphics processor's CONTROL register. These 10 bits are identical in the TMS34010 and TMS34020, and software that accesses only these bits can safely ignore the differences between the two processors. The same cannot be said of the other 6 CONTROL bits, which control disabling of the processor's instruction cache, the TMS34010's DRAM refreshing, and the TMS34020's transparency mode. TI recommends that when you write to a part of the register, you first set the appropriate field size in the status register to only the portion of the register that is actually to be modified. This avoids disturbing the other CONTROL bits. TI discourages reading the entire CONTROL register, modifying a selected field, and writing back the entire register. While the latter method may appear to operate correctly in some environments, the resulting code will not be as portable or robust as code that uses the recommended method.

Refer to the *TMS340 Family Code Generation Tools User's Guide* for a description of the rules imposed by the TMS340 Family C Compiler on function calls.

3.7.2 Interrupts

The assembly language routines within the library use the TMS340 graphics processor's A14 register as general-purpose, temporary storage. 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 program stack. An interrupt service routine written in C must allocate its own program stack. This can be done in one of several ways. For instance, the ISR can temporarily allocate extra space on the system stack, and utilize this space as program stack. Alternately, a temporary program stack for interrupts can be allocated statically as a global array. The latter method is suitable only if the ISR code is not required to be reentrant.

The library routines themselves do not use interrupts. Several library functions make use of the TMS340 graphics processor's WV (window violation) interrupt request, but they assume that the WV interrupt is disabled.

Similarly, the library's *page_flip*, *page_busy,* and *wait_scan* functions poll the DI (display interrupt) request but assume that the display interrupt is dis-

abled. (In other words, the DIE bit in the INTENB register should be 0 if the library implementations of these functions are used.) An operating environment or application program that includes a display interrupt service routine may have difficulty coexisting with these three functions as currently implemented in the library.

The TIGA implementations of the *page_flip*, *page_busy,* and *wait_scan* functions are functionally equivalent to those in the TMS340 Graphics Library but rely on interrupts rather than polling. This is because the TIGA versions of these functions must share the TMS340 graphics processor's display interrupt with other features such as real-time cursor management. Refer to the *TIGA-340 Interface User's Guide* for more information on the TIGA versions of these functions.

3.7.3 System-Level Hardware Functions

Several TMS340 graphics processor hardware functions are most appropriately controlled by systems-level software such as an operating system or control program, if one is present. The graphics library as distributed, however, assumes that no such software is present. Based on this assumption, when the *set_config* function initializes the drawing environment, it also initializes the following hardware functions:

1) The IE (interrupt enable) bit in the status register is set to 0.
2) The INTENB (interrupt enable) register is set to 0.
3) The CD (cache disable) bit in the CONTROL register is set to 0.
4) The DRAM refresh rate and refresh mode are initialized. This involves loading the RM and RR fields in the TMS34010's CONTROL register, or the RR field in the TMS34020's CONFIG register.

If the library is ported to an environment in which these hardware functions are controlled outside of the graphics library, the code for initializing the parameters above should be deleted from the *set_config* routine.

3.7.4 Functions with System Dependencies

Most of the functions in the TMS340 Graphics Library are independent of system-specific features such as pixel size, frame buffer dimensions, and color palette hardware. This is true of all functions in the Extended Primitives Library. Several functions in the Core Primitives Library, however, perform system-dependent operations and must be ported from one TMS340 hardware configuration to another. The system-dependent library functions are listed below according to the types of hardware dependencies they have.

First, the following function performs all the hardware-specific initialization of the video timing registers, screen refresh, DRAM refresh, pixel size, screen dimensions, and so on:

- *set_config*

These six functions depend on the hardware palette configuration:

- *get_nearest_color*
- *get_palet*
- *get_palet_entry*
- *init_palet*
- *set_palet*
- *set_palet_entry*

The implementation of these two functions depends on whether the TMS340 graphics processor's interrupt vectors are mapped into RAM or ROM:

- *set_vector*
- *get_vector*

The following functions may use the *bulk initialization* capability of some video RAMs:

- *clear_frame_buffer*
- *clear_page*

Bulk initialization is a method of rapidly clearing a portion of a display memory that is composed of video RAMs that support both memory-to-serial-register and serial-register-to-memory cycles. First, a single row is loaded from the memory array within each video RAM to the serial register; this is accomplished with a single memory-to-register cycle. Next, the contents of the serial register are rapidly copied to a series of rows; this is accomplished with a sequence of register-to-memory transfers.

The next function is typically implemented with a FILL instruction in a TMS34010-based system but may use the faster VFILL instruction if implemented in a TMS34020-based system with video RAMs that support block write cycles:

- *clear_screen*

The following functions need to be recompiled for the target processor (TMS34010 or TMS34020), but the source code does not require porting:

- *page_flip*

- *wait_scan*

Finally, these routines are included for convenience with the system-dependent functions, even though they are themselves system-independent:

- *get_config*
- *page_busy*

3.7.5 TMS34010 and TMS34020 Code Compatibility

Conditional assembly statements embedded in the assembly code for certain graphics functions control whether the functions are assembled to execute TMS34010 or TMS34020 code. The details of how to configure the library at assembly time are described in Section 2.5.

By default, the library as distributed is configured to generate TMS34010-compatible code. This code is upward-compatible with the TMS34020. That is, it will run correctly on the TMS34020 as well as on the TMS34010, although it may not take full advantage of certain graphics acceleration features unique to the TMS34020. Guidelines for writing upward-compatible TMS340 graphics processor code are given in the user's guides for the TMS34010 and TMS34020.

If you prefer, however, the library can easily be configured to generate TMS34020-specific code. The current library contains implementations of a number of graphics functions that can be configured to take advantage of graphics acceleration features available only on the TMS34020. These features are unavailable on the TMS34010, and the code that uses these features does not execute correctly on the TMS34010.

If you write your own processor-dependent functions, you have an alternative to the conditional assembly method currently used to statically configure the library. The software can perform a runtime check to dynamically determine whether it is running on a TMS34010 or TMS34020 and can execute the appropriate TMS34010- or TMS34020-specific code. This approach is typically used in instances in which the code that performs the runtime check is not speed-critical. Moving a runtime check inside the inner loop of a speed-critical assembly-coded graphics function, for instance, would probably be inappropriate.

If you are writing a proprietary function as an extension to the existing graphics library, you can perform a runtime processor check by reading the *device_rev* field of the global structure *config*. The same information is available to application programs in the *device_rev* field of the CONFIG structure retrieved by the *get_config* function.

3.7.6 Floating-Point Compatibility

Versions 4.0 and above of the TMS340 Family C Compiler can be configured to generate either TMS340-compatible or IEEE-compatible floating-point values, as described in the *TMS340 Family Code Generation Tools User's Guide*. The graphics library does not utilize any of the floating-point math routines distributed with the C compiler. The applications programmer can configure proprietary code to use either of the two floating-point formats without concern for the effect this choice will have on the graphics library functions.

The IEEE-compatible format is defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic. The TMS340-compatible format is similar to the IEEE, except that it uses an explicit leading 1 in the mantissa in place of the implicit leading 1 used in the IEEE format. For more information, refer to the *TMS340 Family Code Generation Tools User's Guide*.

3.7.7 Silicon Revision Number

The TMS340x0 REV assembly-language instruction generates a silicon revision number that identifies the processor as a TMS34010 or TMS34020 and also indicates the silicon revision. One of the initialization operations performed by the library's *set_config* function is to load the revision number into the *device_rev* field of the global structure *config* mentioned earlier. (Refer to the description of the CONFIG data structure in Appendix A.)

The earliest TMS34010 devices do not recognize the REV instruction; they treat it as an illegal opcode. To permit the TMS340 Graphics Library to run without mishap on these early devices, the library includes an illegal opcode interrupt service routine that emulates execution of the REV instruction.

If your graphics card contains a TMS34020, or a later version of the TMS34010 that recognizes REV as a valid instruction, you may choose to delete the illegal opcode ISR from the library. Please refer to subsection 2.3.2 for details.

Chapter 4

Graphics Operations

The TMS340 Graphics Library supports the drawing of a variety of two-dimensional geometric objects such as points, lines, polygons, ellipses, arcs, pie-slice wedges, and polygons.

Geometric objects can be rendered in a variety of styles. Filled primitives such as polygons and ellipses can be filled with either a solid color or a two-dimensional area-fill pattern. Vector primitives that produce 1-pixel-thick lines and arcs can be drawn either in a single color or with a one-dimensional line-style pattern. A rectangular drawing pen (or brush) is available for producing thicker lines and arcs; the area swept out by the pen is filled with either a solid color or an area-fill pattern.

The library follows a set of strict conventions in order to make the behavior of the drawing functions (library functions that produce graphics output) predictable in all cases. These conventions cover the following:

- The naming of the functions
- The mapping of x–y coordinates onto the screen (a display surface addressed as a two-dimensional array of pixels)
- Defining the paths followed by vector primitives such as lines and arcs
- Defining the pixels covered by area-fill primitives such as polygons and ellipses

The library supports a variety of methods for combining source and destination pixel values during drawing operations. Pixels are combined according to how the user configures the library's plane mask, transparency attribute, and pixel-processing operation code.

All graphics output is automatically clipped either to the screen or to a rectangular clipping window located within the screen limits.

4.1 Graphics-Function Naming Conventions

A set of conventions has been adopted for naming graphics functions in the Extended Primitives Library that draw geometric objects such as lines and ellipses. Each object can be rendered in a variety of styles, and the rendering style also is reflected in the function name. The name assigned to the function is a concatenation of a modifier (such as *rect* for rectangle) denoting a geometric type and another modifier (such as *fill*) designating a rendering style. For example, the *fill_rect* function fills a rectangle with a solid color.

Table 4–1 is a list of the geometric types supported by the library. The left column specifies the function-name modifier corresponding to each type.

Table 4–1. Geometric Types

Function Name	Geometric Type
line	A straight line
oval	An ellipse in standard position (major and minor axes aligned with the x–y coordinate axes)
ovalarc	An arc from an ellipse in standard position
point	A single point
polygon	A filled region bounded by a series of connected straight edges
polyline	A series of connected straight lines
piearc	A pie-slice-shaped wedge bounded by an arc (from an ellipse in standard position) and two straight edges (connecting the ends of the arc to the center of the ellipse)
rect	A rectangle with vertical and horizontal sides
seed	A pixel of a particular color designating a connected region of pixels of the same color

Table 4–2 is a list of the graphics rendering styles supported by the library. The left column specifies the function-name modifier corresponding to each style.

Table 4–2. Rendering Styles

Function Name	Rendering Style
draw	Draws a line, arc, or outline a single pixel thick with the current foreground color.
styled	Similar to “draw” except that the line, arc, or outline is drawn with a repeating 32-bit line-style pattern that is rendered in the current foreground and background colors. Alternately, background pixels in the pattern are skipped.
pen	Traces a line or curve with a rectangular drawing pen, and fills the area swept out by the pen with the current foreground color.
patnpen	Similar to “pen” except that the area swept out by the pen is filled with a 16-by-16 area-fill pattern in the current foreground and background colors.
fill	Fills the interior of an object with the current foreground color.
patnfill	Similar to “fill” except that the object is filled with a 16-by-16 area-fill pattern in the current foreground and background colors.
frame	Fills a frame with the current foreground color. The area enclosed by the frame is not modified.
patnframe	Similar to “frame” except that the frame is filled with a 16-by-16 area-fill pattern in the current foreground and background colors.

Not all combinations of geometric type and rendering style are available in the library. Table 4–3 is a checklist indicating which combinations are supported.

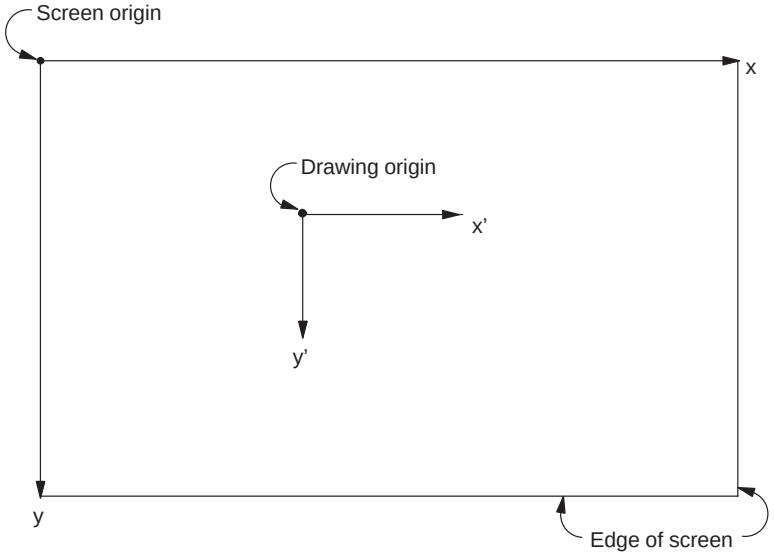
Table 4–3. Checklist of Available Geometric Types and Rendering Styles

	Rendering Style							
Geometric Type	**draw**	**styled**	**pen**	**patnpen**	**fill**	**patnfill**	**frame**	**patnframe**
line	√	√	√	√				
oval	√	√			√	√	√	√
ovalarc	√	√	√	√				
piearc	√	√	√	√	√	√		
point	√		√	√				
polygon					√	√		
polyline	√		√	√				
rect	√				√	√	√	√
seed					√	√		

4.2 Coordinate Systems

Figure 4–1 shows the conventions used by the library to map x–y coordinates onto the screen.

Figure 4–1. Screen Coordinates and Drawing Coordinates



The *screen coordinate system* maps the pixels on the display surface to x and y coordinates. By convention, the screen origin is located in the top left corner of the screen. The x axis is horizontal, and x increases from left to right. The y axis is vertical, and y increases from top to bottom.

A *drawing coordinate system* is also defined. All drawing operations (both graphics and text output) take place relative to the drawing origin. Unlike the screen origin, which remains fixed, the drawing origin can be moved relative to the screen. The directions of the x and y axes match those of the screen coordinate system.

The drawing origin is aligned with the screen origin immediately after initialization of the graphics environment by the *set_config* function. The drawing origin may be displaced in x and y from the screen origin by means of a call to the *set_draw_origin* function. All subsequent drawing operations are specified relative to the new drawing origin. While Figure 4–1 shows the drawing origin lying within the boundaries of the screen, the origin may also be moved to a position above, below or to the side of the screen. Only objects that are drawn on the screen and within the clipping window (to be described) will be visible.

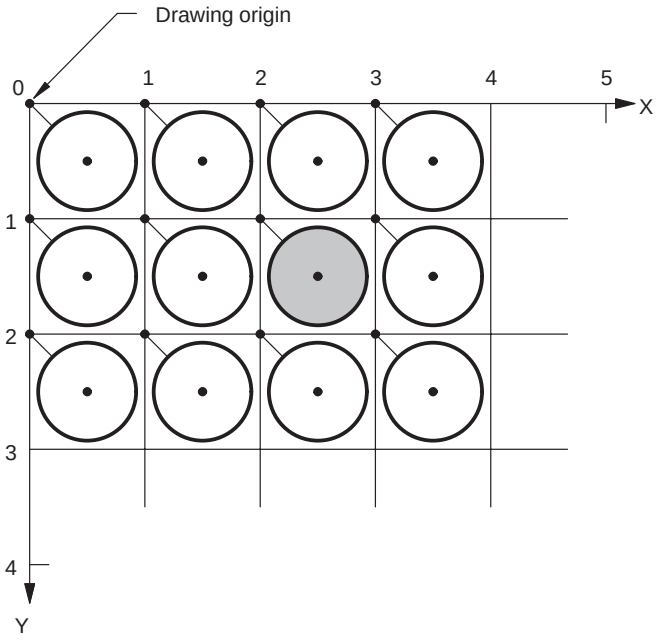
Figure 4–2 is a close-up of several pixels in the vicinity of the drawing origin that illustrates the relationship of the pixels on the screen to the coordinate grid lines. Each vertical or horizontal grid line corresponds to an integer x or y coordinate value. Centered within each square of the grid is a pixel, drawn as a circle.

The *draw* and *styled* functions within the library (refer to Table 4–2) identify a pixel by the integer x–y coordinates at the top left corner of its grid square. For instance, the pixel designated by the function call

```
draw_point(2, 1);
```

is, in fact, centered at (2.5, 1.5) and is darkened in Figure 4–2.

Figure 4–2. Mapping of Pixels to Coordinate Grid



The graphics library represents x–y coordinates as 16-bit signed integers. Valid coordinate values are limited to the range –16384 to +16383. Restricting the values to this range provides one guard bit to protect against overflow during 16-bit arithmetic operations.

4.3 Area-Filling Conventions

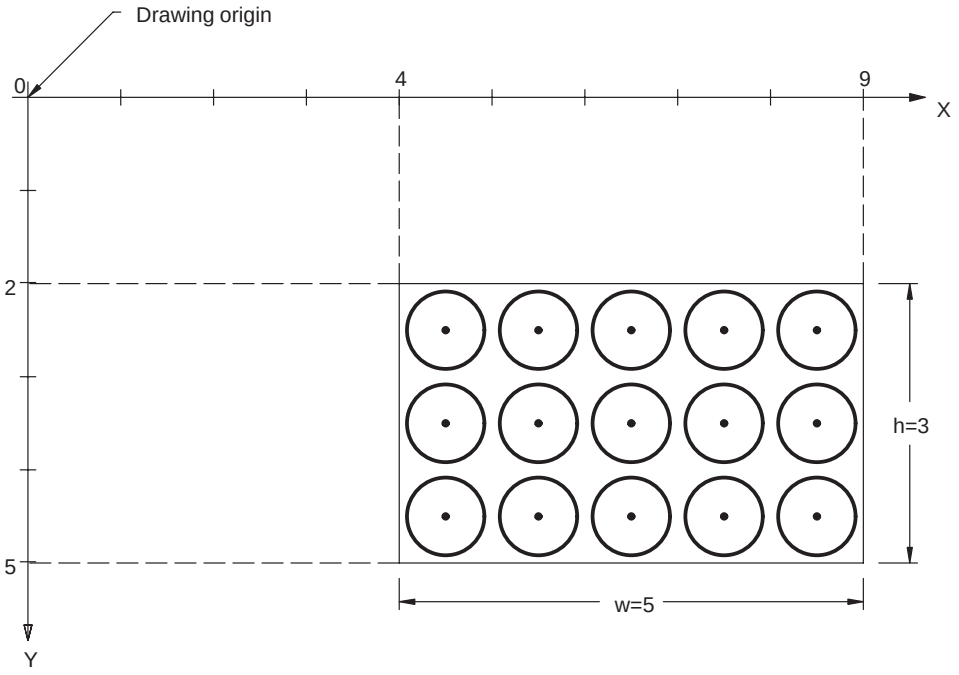
The *fill*, *frame,* and *pen* functions within the library designate a pixel as being part of a filled region if the center of the pixel falls within the boundary of that region.

Figure 4–3 shows an example of a filled region — a rectangle of width 5 and height 3. The top left corner of the rectangle is located at coordinates (4, 2). The function call to fill this particular rectangle is

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

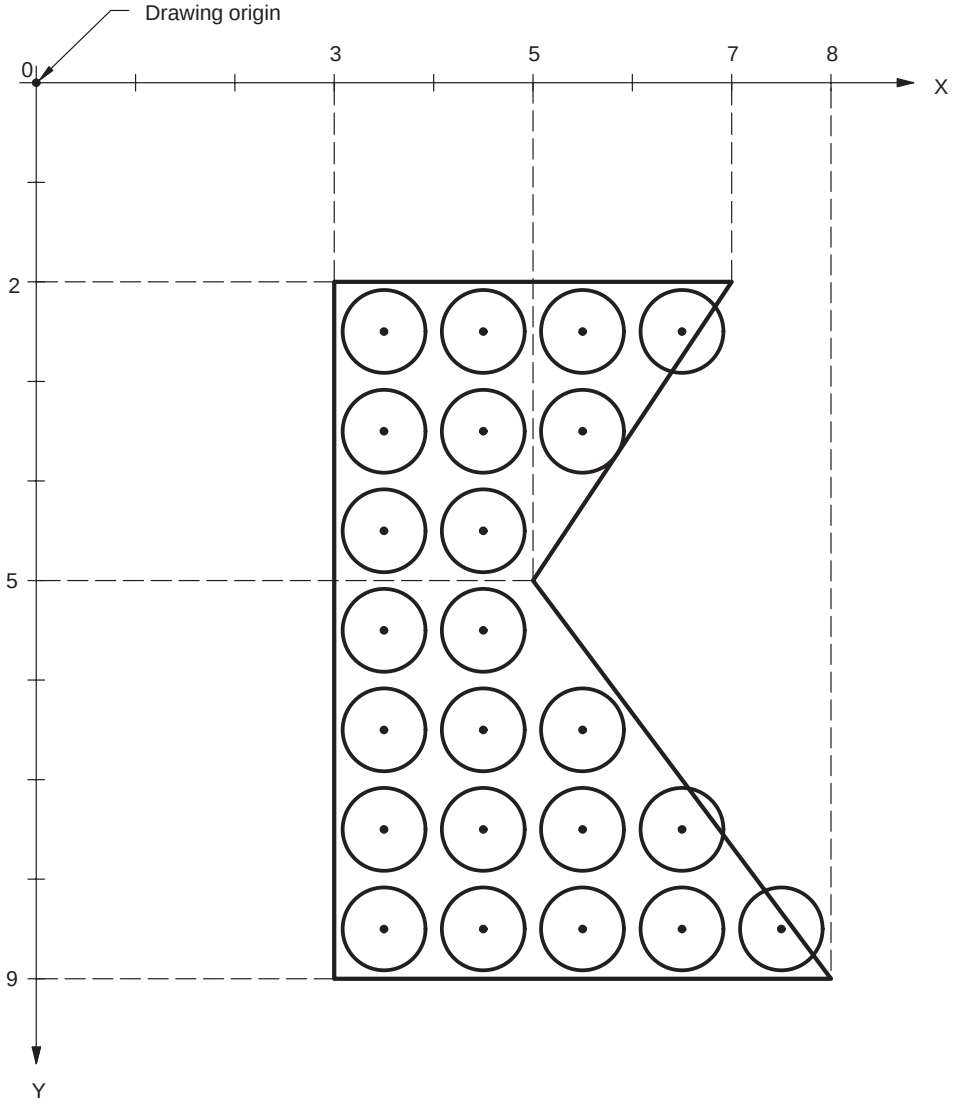
The pixels selected to fill the rectangle are indicated in the figure.

Figure 4–3. A Filled Rectangle



As a second example, a filled polygon is shown in Figure 4–4. The five straight edges of the polygon separate the interior of the polygon, which is filled, from the exterior. (This figure was drawn using the *fill_polygon* function.)

Figure 4–4. A Filled Polygon



By convention, a pixel is considered to be part of the interior if its center falls within the boundary of the polygon. If the center falls precisely on a boundary, the pixel is inside if and only if the polygon interior is immediately to its right (x-increasing direction). Pixels with centers along a horizontal edge are a special case and are inside if and only if the polygon interior is immediately below (y-increasing direction).

The names of graphics functions that follow the area-filling conventions described in this section include the modifiers *fill*, *pen,* or *frame*.

4.4 Vector-Drawing Conventions

Mathematically ideal points, lines, and arcs are defined to be infinitely thin. Since these figures contain no area, they would be 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. Vector-drawing conventions apply to all library functions whose names include the modifiers *draw* or *styled*.

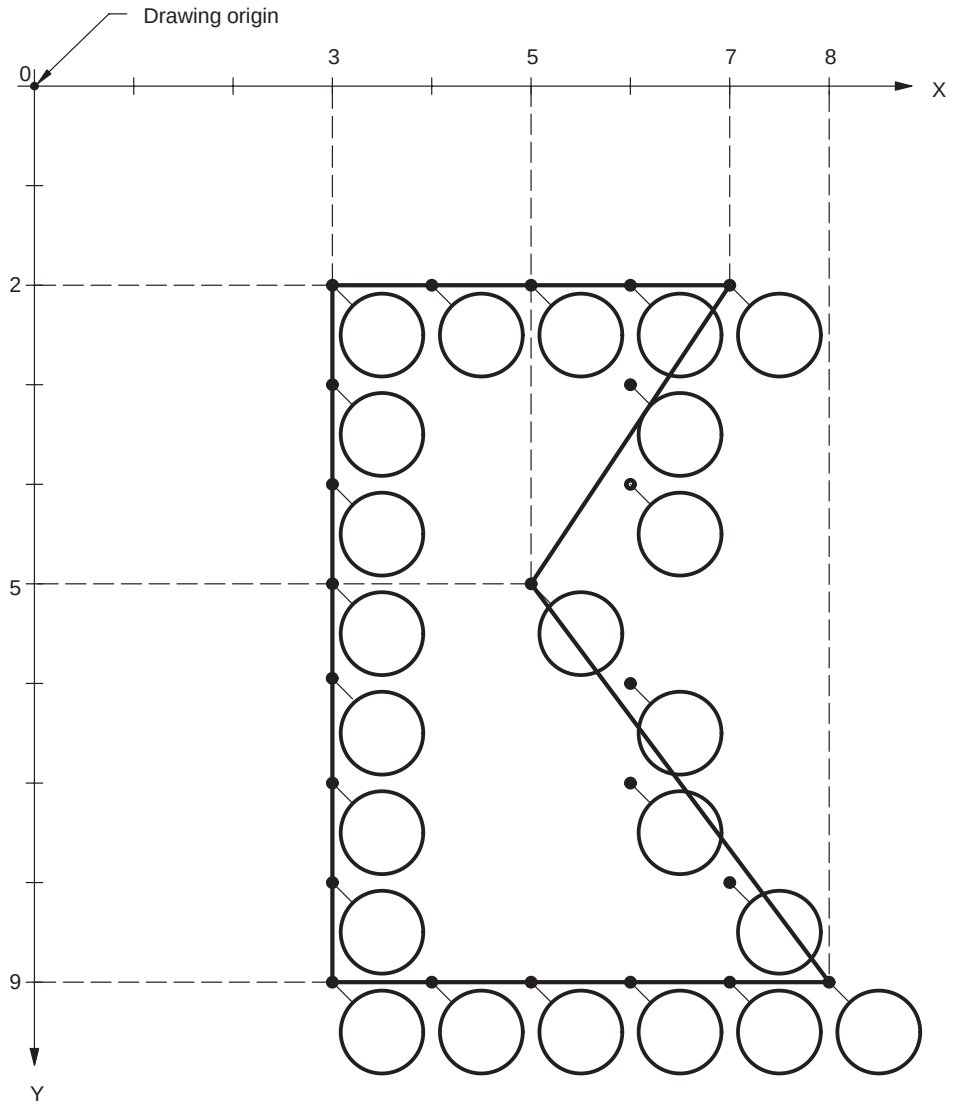
The vector-drawing conventions associate the point specified by the integer coordinate pair (x, y) with the pixel that lies just to the lower right of this point; that is, the pixel whose center lies at coordinates (x+1/2, y+1/2). For example, the function call

```
draw_point(2, 1);
```

draws the pixel centered at (2.5, 1.5), as shown in Figure 4–2.

As a second example, the polygon from Figure 4–4 is shown again in Figure 4–5, but this time it is outlined rather than filled. (This figure was drawn using the *draw_polyline* function.) The integer coordinate points selected to represent the edges 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.

Figure 4–5. An Outlined Polygon



A line or arc drawn using the vector-drawing conventions consists of a thin, but connected set of pixels selected to follow the ideal line or arc as closely as possible. Each pixel is horizontally, vertically, or diagonally adjacent to its neighbor on either side, with no holes or gaps in between. The resulting line or arc is only a single pixel in thickness.

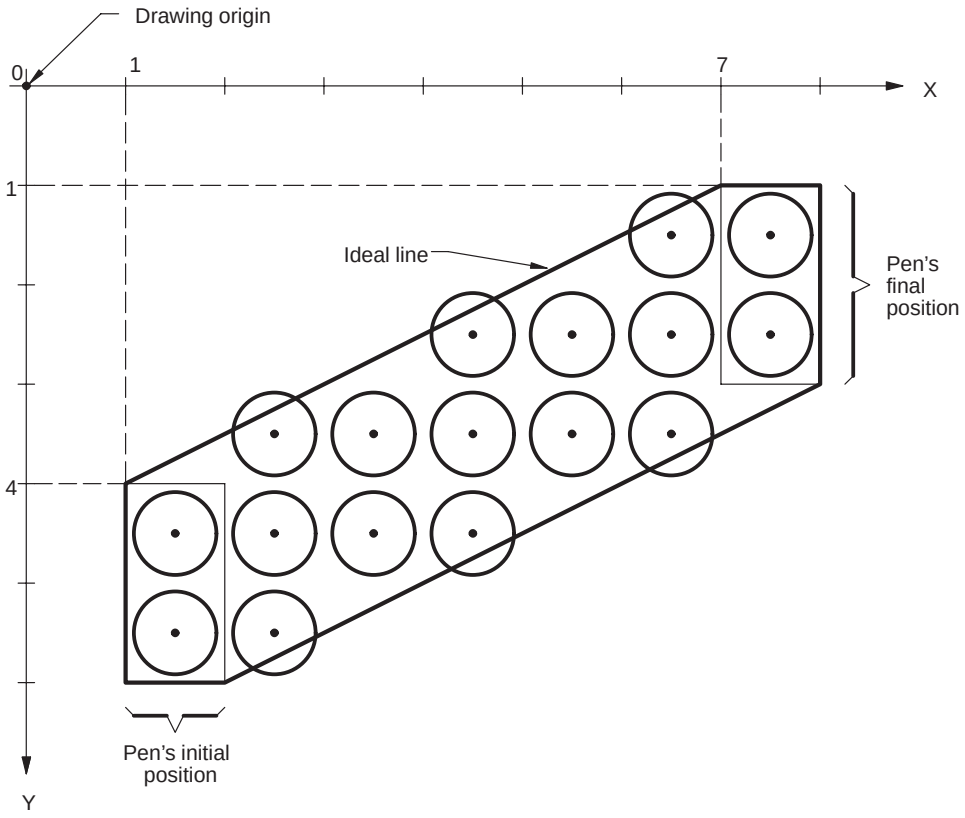
4.5 Rectangular Drawing Pen

The graphics functions that follow the vector-drawing conventions above can draw only lines and arcs that are a single pixel in thickness. To draw lines and arcs of arbitrary thickness, a logical *pen* (or brush) is defined. Library functions that use the pen include the modifier *pen* as part of their names.

The drawing pen is rectangular, and its position is defined by the integer coordinates at its top left corner. When a pen of integer width w and height h draws a point at (x, y), the 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 either with a solid color or with an area-fill pattern, depending on the function called.

Figure 4–6 shows a line from (1, 4) to (7, 1) drawn by a pen of width 1 and height 2. The pen is initially positioned at the bottom left of the figure, with its top left corner at (1, 4). As the pen moves along the line, the pen is always located with its top left corner touching the ideal line. The area swept out by the pen as it traverses the line from start to end is filled according to the area-filling conventions described previously. The pixels interior to the line are indicated in the figure.

Figure 4–6. A Line Drawn by a Pen



When the pen's width and height are both 1, a line or arc drawn by the pen is similar in appearance to one drawn using the vector-drawing conventions discussed previously. The pen, however, conforms to the area-filling conventions, and a pen function can track the perimeter of a filled figure more faithfully than the corresponding vector-drawing function can.

For instance, consider an ellipse defined by some width w, height h, and top-left-corner coordinates (x, y). The ellipse is filled by the function call

```
fill_oval(w, h, x, y);
```

If the filled ellipse is outlined by calling *draw_oval*, which is a vector-drawing function, with the same arguments, the outline may not conform to the edge of the filled area, and gaps may appear between the filled area and the outline. Calling the *pen_oval* function with the same arguments, however, draws an outline that follows the edge of the filled area precisely, remaining flush to the ellipse at all points along the perimeter.

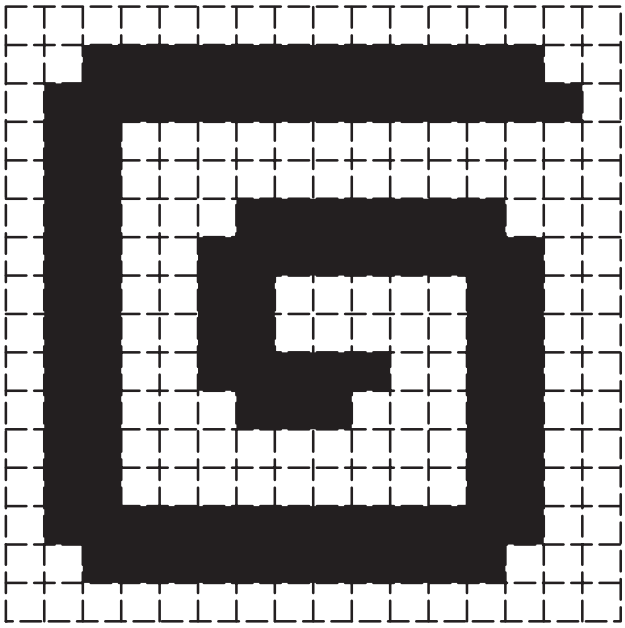
4.6 Area-Fill Patterns

Graphics functions that include the modifier *patn* as part of their names fill geometric figures with a two-dimensional pattern rather than a solid color. Currently, the only *area-fill patterns* supported are two-color patterns that are 16 pixels wide by 16 pixels high.

The tiling of patterns to the screen is currently fixed relative to the top left corner of the screen. In other words, changing the drawing origin causes no shift in the mapping of the pattern to the screen, although the geometric objects filled with the pattern are themselves positioned relative to the drawing origin. The screen-relative x and y coordinate values at the top left corner of each instance of the pattern are multiples of 16.

Before an area on the screen is filled with a particular pattern, the pattern must be installed by calling the *set_patn* function. The pattern is specified as a 16-by-16 bit map, as shown in Figure 4–7, and is stored in memory as an array of 256 contiguous bits. The bits within a pattern bit map are listed in left-to-right order within a row, and the rows are listed in top-to-bottom order. For instance, the top row in the figure contains bits 0 (left) to 15 (right); bit 255 is located in the bottom right corner. The shaded squares in Figure 4–7 represent to 1s in the source bit map, and white squares represent to 0s. When a pattern is drawn to the screen, screen pixels corresponding to 1s in the bit map are replaced by the foreground color, and 0s by the background color.

Figure 4–7. A 16-by-16 Area-Fill Pattern



4.7 Line-Style Patterns

Graphics functions that include the modifier *styled* as part of their names draw lines and arcs using a *line-style pattern*. A line-style pattern is a 1-dimensional pattern of two colors. The pattern controls the color of each successive pixel output to the screen as a line or arc is drawn.

The line-style pattern is specified as a 32-bit mask containing a repeating pattern of 1s and 0s. The pattern bits are consumed in the order 0,1,...,31, where 0 is the LSB. If the line is more than 32 pixels long, the pattern is repeated modulo 32 as the line is drawn. A bit value of 1 in the pattern mask means that the corresponding pixel is drawn in the foreground color, while a 0 means that the pixel is drawn in the background color. As an option, background pixels can be skipped over rather than drawn.

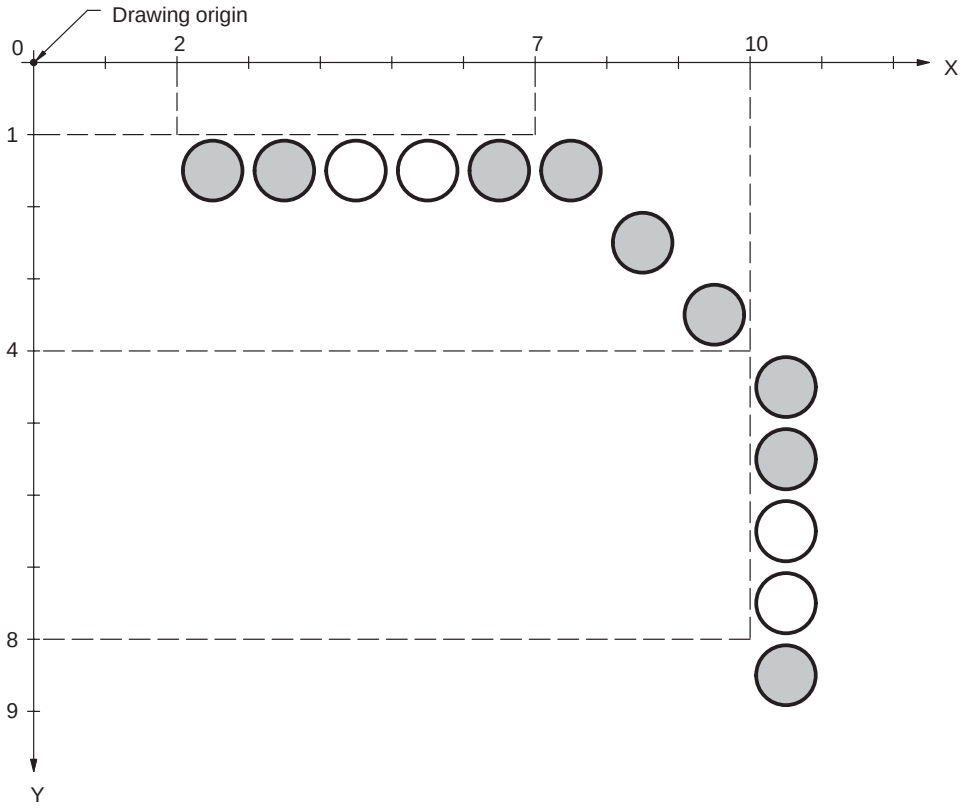
When a line-style pattern function such as *styled_line* is called, either a new pattern mask is specified, or an old one is reused. The latter option supports the drawing of continuous patterns across a series of connecting lines. After the *styled_line* function has been used to draw a line *n* pixels in length, the original pattern has been rotated left (*n*–1) modulo 32 bits. The rotated pattern is always saved by the function before returning. The saved pattern is ready to be used as the pattern for a second line that continues from the end point of the first line. The last pixel plotted in the first line is identical to the first pixel in the second line.

For example, three connected styled lines are shown in Figure 4–8. Darkened pixels correspond to 1s in the line-style mask, and white pixels correspond to 0s. The lines in the Figure 4–8 are drawn by the following three function calls:

```
styled_line(2, 1, 7, 1, 1, 0xF3F3F3F3);
styled_line(7, 1, 10, 4, 3, 0);
styled_line(10, 4, 10, 8, 3, 0);
```

The first call loads the line-style mask 0xF3F3F3F3, and draws a line from (2, 1) to (7, 1). The last two calls reuse the mask loaded by the first call and draw lines from (7, 1) to (10, 4) to (10, 8).

Figure 4–8. Three Connected Styled Lines



4.8 Operations on Pixels

Drawing (or graphics output) operations consist of replacing one or more pixels on the screen with new pixel values. By default, a specified source pixel simply replaces a designated destination pixel. The graphics library, however, provides several optional methods for processing the source and destination pixel values to determine the final pixel value written to the screen.

- *Transparency* is a pixel attribute that, when enabled, permits objects written onto the screen to have transparent regions through which the original background pixels are preserved.
- The *plane mask* specifies which bits within pixels can be modified during pixel operations.
- Boolean and arithmetic *pixel-processing operations* specify how source and destination pixel values are combined.

These three methods for processing pixels can be used independently or in conjunction with each other. Transparency, plane masking, and pixel processing are orthogonal in the sense that they can be used in any combination, and each is controlled independently of the other two. These attributes affect all drawing operations, including those that involve text, geometric objects, and pixel arrays.

Immediately following initialization of the drawing environment by the *set_config* function, the following defaults are in effect:

- Transparency is disabled (all pixels are opaque).
- The plane mask is 0 (all bits within pixels can be modified).
- The pixel-processing operation is *replace* (the source pixel value simply replaces the destination pixel).

Transparency, plane masking, and pixel processing are described individually below. Refer to the user's guides for the TMS34010 and TMS34020 for additional information.

4.8.1 Transparency

Pixel transparency is useful in applications involving text, area-fill patterns, and pixel arrays in which only the shapes, and not the extraneous pixels surrounding them, are to be drawn to the screen. When a rectangular pixel array containing a shape is written to the screen, the pixel transparency attribute can be enabled to avoid modifying destination pixels in the rectangular region surrounding the shape. In effect, the source pixels surrounding the shape are treated as though they are transparent rather than opaque.

The library's default transparency mode is enabled and disabled by calls to the *transp_on* and *transp_off* functions. In TMS34020-based systems, additional transparency modes may be selected by means of the *set_transp* function. Only the default mode is available in TMS34010-based systems. Refer to the *TMS34020 User's Guide* for information on the additional modes.

When transparency is enabled in the default mode, a pixel that has a value of 0 is considered to be transparent, and it will not overwrite a destination pixel. The check for a 0-valued pixel is applied not to the original source pixel value, but to the pixel value resulting from pixel processing and plane masking. In the case of pixel processing operations such as AND, MIN, and *replace*, a source pixel value of 0 ensures that the result of the operation will be a transparent pixel, regardless of the destination pixel value.

4.8.2 Plane Mask

The plane mask specifies which bits within a pixel are protected from modification, and affects all operations on pixels. The plane mask has the same number of bits as a pixel in the display memory. A value of 1 in a particular plane mask bit means that the corresponding bit in a pixel is protected from modification. Pixel bits corresponding to 0s in the plane mask can be modified.

The plane mask allows the bits within the pixels on the screen to be manipulated as bit planes (or color planes) that can be modified independently of other planes. A useful way to think of planes is as laminations or layers parallel to the display surface. The number of planes is the same as the number of bits in a pixel.

For example, at 4 bits per pixel, three contiguous planes can be dedicated to 8-color graphics, while the fourth is used to overlay text in a single color. The plane mask permits the text layer to be manipulated independently of the graphics layers, and *vice versa*.

During a write to a pixel in memory, the 1s in the plane mask designate which bits in the pixel are write-protected; only pixel bits corresponding to 0s in the plane mask are modified. During a pixel read, 1s designate which bits within a pixel are always read as 0, regardless of their values in memory; only pixel bits corresponding to 0s in the plane mask are read as they appear in memory.

The plane mask can be modified by means of a call to the library's *set_pmask* function.

4.8.3 Pixel-Processing Operations

During drawing operations, source and destination pixels are combined according to a specified Boolean or arithmetic operation and written back to

the destination pixel. The library supports 16 Boolean pixel-processing operations (or “raster ops”) and 6 arithmetic operations. The Booleans are performed in bitwise fashion on operand pixels, while the arithmetic operations treat pixels as unsigned integers.

A 5-bit PPOP code specifies one of the 22 pixel-processing operations, as shown in Table 4–4 and Table 4–5. Legal PPOP codes are in the range 0 to 21. As shown in the two tables, codes for Boolean operations are in the range 0 to 15, and codes for arithmetic operations are in the range 16 to 21.

Table 4–4. Boolean Pixel-Processing Operation Codes

PPOP Code	Description
0	replace destination with source
1	source AND destination
2	source AND NOT destination
3	set destination to all 0s
4	source OR NOT destination
5	source EQU destination
6	NOT destination
7	source NOR destination
8	source OR destination
9	destination (no change)
10	source XOR destination
11	NOT source AND destination
12	set destination to all 1s
13	NOT source OR destination
14	source NAND destination
15	NOT source

Table 4–5. Arithmetic Pixel-Processing Operation Codes

PPOP Code	Description
16	source plus destination (with overflow)
17	source plus destination (with saturation)
18	destination minus source (with overflow)
19	destination minus source (with saturation)
20	MAX(source, destination)
21	MIN(source, destination)

The result of an *arithmetic* pixel-processing operation is undefined at screen pixel sizes of 1 and 2 bits on the TMS34010, and at a pixel size of 1 bit on the TMS34020.

The PPOP code can be altered with a call to the *set_ppop* function.

4.9 Clipping Window

The graphics output produced by the library's drawing functions is always confined to the interior of a rectangular clipping window that occupies all or a portion of the screen. All library drawing functions automatically inhibit attempted writes to pixels outside this window.

The width, height, and position of the clipping window can be modified by a call to the *set_clip_rect* function. The function call

```
set_clip_rect(w, h, x, y);
```

defines the window to be a rectangle of width w and height h whose top left corner lies at coordinates (x, y). The x–y coordinates are specified relative to the drawing origin in effect at the time the function is called. The four sides of the clipping window are parallel to the x and y axes. If a clipping rectangle is specified that lies partially outside the screen boundaries, the *set_clip_rect* function automatically trims the window to the limits of the screen.

The default clipping window covers the entire screen. This default is in effect immediately following initialization of the drawing environment by the *set_config* function.

4.10 Pixel-Size Independence

The TMS34010 can support pixel sizes of 1, 2, 4, 8, and 16 bits, and the TMS34020 can support pixel sizes of 1, 2, 4, 8, 16, and 32 bits. Any particular TMS340 graphics processor-based display hardware system, however, may support only a subset of the pixel sizes that the processor itself can handle. Possible hardware limitations include the amount of video RAM in the system and the pixel sizes supported by the color palette device.

With the exception of the handful of system-dependent functions described in subsection 3.7.4, the graphics library functions are written to be independent of the pixel size. The library achieves pixel-size independence by taking advantage of special graphics hardware internal to the TMS34010 and TMS34020 processor chips. Changing the pixel size in software is not much more difficult than loading the processor's PSIZE (pixel size) register with a new value.

Application programs based on the graphics library are potentially able to execute on display systems that support a variety of pixel sizes. Ideally, an application program should be flexible enough to take advantage of the large number of colors available in systems with large pixel sizes, yet also run satisfactorily in systems that are limited to small pixel sizes. In practice, this ideal may be difficult to achieve.

For instance, an application written to run on a 1-bit-per-pixel display should be able to run with little modification at 2, 4, 8, 16, or 32 bits per pixel. This is achieved, however, by restricting the application's choice of colors to black and white, regardless of the number of colors supported by the display hardware.

At the other end of the spectrum, consider an application that is written to control a true color display with 8 bits of red, green, and blue intensity per pixel. The application writer may be able to stretch the program to reasonably accommodate pixel sizes of 16 or even 8 bits per pixel, although at some loss in image quality. This can be done by using certain well-known halftoning or ordered-dithering algorithms to simulate a larger palette of colors. The application is unlikely, however, to run satisfactorily on a 1-bit-per-pixel display.

To summarize, the graphics library's high degree of pixel-size independence represents a powerful and useful feature. This does not automatically guarantee that all applications that call the library will not themselves contain inherent color dependencies.

Chapter 5

Bit-Mapped Text

The TMS340 Family Graphics Library supports the display of text in a variety of styles and sizes. At the low end, block fonts emulate the cell-mapped text produced by a character-ROM display. For desktop publishing applications, proportionally spaced WYSIWYG (what you see is what you get) text allows you to preview a page on the screen as it will appear when typeset.

Table 5–1 lists the text-related functions available in both the Core and Extended Primitives Libraries. Refer to the individual descriptions of these functions in Chapters 6 and 7 for details.

Table 5–1. Text-Related Functions

Function	Description	Type
delete_font	Remove a font from the font table	Ext
get_fontinfo	Return font physical information	Core
get_textattr	Return text rendering attributes	Ext
get_text_xy	Get text x-y position	Core
init_text	Initialize text drawing environment	Core
install_font	Install font into font table	Ext
select_font	Select an installed font for use	Ext
set_textattr	Set text rendering attributes	Ext
set_text_xy	Set text x-y position	Core
text_out	Render an ASCII string	Core
text_outp	Output text at current x-y position	Core
text_width	Return the width of an ASCII string	Ext

A font is a complete assortment of characters of a particular size and style (or typeface). The library currently supports fonts represented in bit-mapped form, although other representations (stroke and outline font formats, for example) may be supported in the future.

A bit-mapped representation of a font encodes the shape of each character in a bit map—a two-dimensional array of bits representing a rectangular image. The 1s in the bit map represent the body of the character, while the 0s represent the background. The character shape is drawn to the screen by expanding each bit to the pixel depth of the screen: 1s are expanded to the foreground color, and 0s to the background color.

5.1 Bit-Mapped Font Parameters

Figure 5–1 illustrates the parameters that characterize a bit-mapped character shape. These parameters are defined as follows:

Base Line — The *base line* is an invisible reference line corresponding to the bottom of the characters, not including the descenders.

Ascent — The *ascent* is measured as the number of vertical pixels from the base line to the top of the highest character (or more precisely, the top of the font rectangle, defined below). For example, in Figure 5–1, the ascent is 16 pixels.

Descent — The *descent* is measured as the number of vertical pixels from the base line to the bottom of the lowest descender. For example, in Figure 5–1, the descent is six pixels.

Leading — The *leading* is the number of vertical pixels between the descent line of one row of characters and the ascent line of the row just beneath it. For example, in Figure 5–1, the leading is five pixels. The term *leading* derives from the time that typesetters used strips of lead to separate rows of characters in their printing presses.

Character Origin — The *character origin* is the point in the character whose coordinates designate the position of the character when it is drawn on the screen. The position of the origin relative to the body of the character depends on the state of the library's text alignment attribute. In the default state, the origin lies at the top left corner of the character. Alternately, as shown in Figure 5–1, the origin can be located at the intersection of the base line with the left edge of the character, excluding any portion of the character which kerns to the left of the origin (as in the case of the descender of the character *j* in the figure). The base line origin is useful when multiple fonts are mixed in a single line of text, in which case the base lines for all characters should coincide.

Character Height — The *character height* is the sum of the ascent, the descent, and the leading. For example, in Figure 5–1, the character height is 16+6+5=27 pixels. Character

height is constant for all characters within a particular font but can vary between fonts.

Character Width

The *character width* is the distance from the character origin of the current character to the origin of the next character to its right. This width typically spans both the character image and the space separating the character image from the next character. The character width can vary from one character to the next within a font. For example, in Figure 5–1, the widths of the characters *A* and *j* are 18 and 6, respectively.

Character Rectangle

The *character rectangle* is a rectangle enclosing the character image. This image corresponds to the portion of the font data structure containing the bit map for the character shape. The sides of the rectangle are defined by the image width and the font height, as defined below. For example, in Figure 5–1, the character rectangle for the letter *A* is 16 pixels wide by 22 pixels high.

Font Height

The *font height* is the the sum of the ascent and descent parameters for the font.

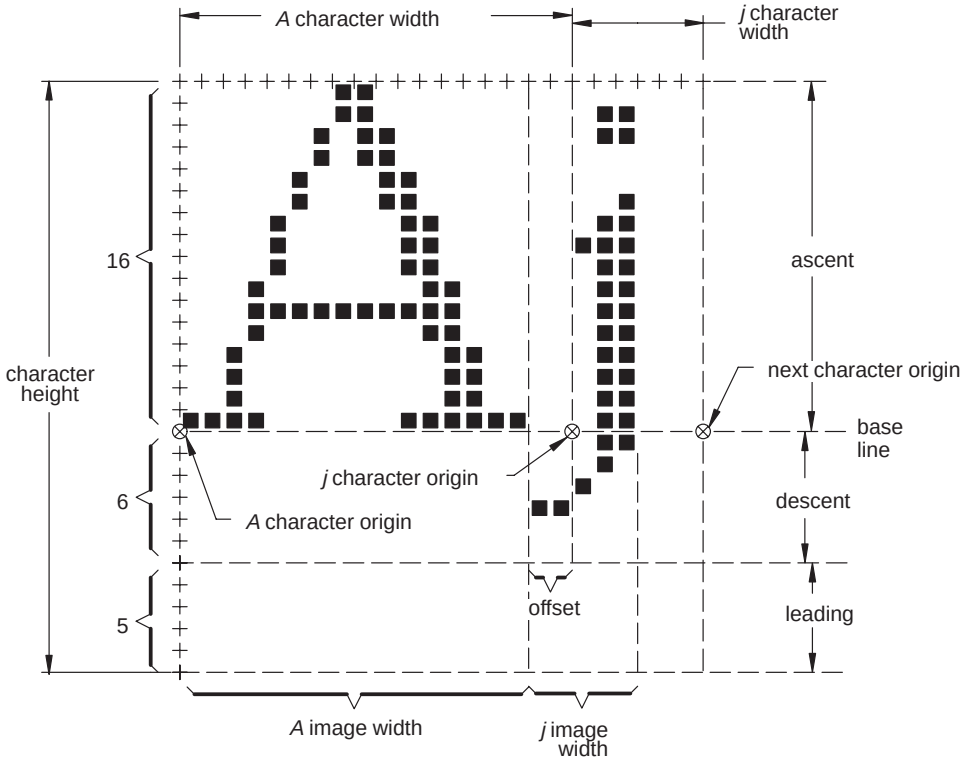
Character Offset

The *character offset* is the horizontal displacement from the character origin to the left edge of the character image. If the offset is negative, the character image extends to the left of the character origin. For example, in Figure 5–1, the descender of the lower-case *j* has an offset of –2. In the case of an especially narrow character, such as a lower-case *i* or *l*, a positive offset may be required to position the left edge of the character image to the right of the origin.

Image Width

The *image width* is the width of the bit map within the font data structure that contains the shape of the character. This width may not include the blank space separating the character from the characters to its left or right when it is displayed. In general, the image width varies from character to character within a font. For example, in Figure 5–1, the image widths of the characters *A* and *j* are 16 and 5, respectively.

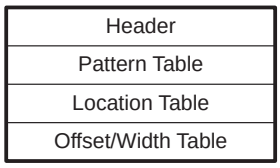
Figure 5–1. Bit-Mapped Font Attributes



5.2 Font Data Structure

The data structure for bit-mapped fonts that is used within both TIGA and the TMS340 Graphics Library is shown in Figure 5–2. The *header* portion is fixed in size and specifies font parameters such as ascent, descent, and so on. The other three parts—the *pattern*, *location,* and *offset/width* tables—vary in size from one font to the next. The pattern table is a bit map containing the shapes of the characters in the font. Each entry in the location table is an offset indicating where in the bit map the shape of a particular character is located. The offset/width table gives the character width, as defined above, for each character and also the character offset from the origin to the left edge of the character image. In general, the larger a particular font appears on the screen, the larger the data structure must be to represent it.

Figure 5–2. Data Structure for Bit-Mapped Fonts



5.2.1 Font Header Information

The header information is organized according to the FONT structure defined in the following C *typedef* declaration:

```
typedef struct
{
    unsigned short magic;    /* bit–mapped font code 0x8040 */
    long length;             /* length of font data in bytes */
    char facename[30];       /* ASCII string name of font */
    short deflt;             /* default for missing character */
    short first;             /* first ASCII code in font */
    short last;              /* last ASCII code in font */
    short maxwide;           /* maximum character width */
    short maxkern;           /* maximum kerning amount */
    short charwide;          /* block font character width */
    short avgwide;           /* average character width */
    short charhigh;          /* character height */
    short ascent;            /* ascent of highest character */
    short descent;           /* longest descender */
    short leading;           /* separation between text rows */
    long rowpitch;           /* bit pitch of pattern table */
    long oPatnTbl;           /* offset to pattern table */
    long oLocTbl;            /* offset to location table */
    long oOwTbl;             /* offset to offset/width table */
} FONT;
```

The fields of the FONT struct (font structure header) are defined as follows:

1) *magic*

   This field contains the value 0x8040, a code that designates the FONT structure for bit-mapped fonts above. If alternate data structures for stroke or outline fonts are supported in the future, these will be distinguished by alternate *magic* codes.

2) *length*

   The *length* of the entire font specified in 8-bit bytes. The length includes the entire data structure from the start of the *magic* field to the end of the offset/width table. The *length* parameter provides a convenient means for a program to determine how much memory to allocate for a font without having to analyze the internal details of the font data structure.

3) *facename*

   A 30-character string consisting of a font name of up to 29 characters, and a terminating *null* character. Some examples: “TI Roman”, “TI Helvetica”.

4) *deflt*

   The ASCII code of the default character to be used in place of a character missing from the font. When a missing character is encountered in an ASCII string, the default character is printed in its place. The default character must be implemented in the font. Typical choices for a default character include a space (ASCII code 32), period (46), and question mark (63). A value of 0 for the *deflt* field is a special case indicating that nothing is to be printed in place of the missing character; it is simply ignored.

   A *missing character* is any character that is not implemented in the font. By definition, all characters with ASCII codes in the ranges [1...*first*–1] and [*last*+1...255] are missing. (Note that ASCII code 0, or *null*, is reserved for use as a string terminator.) If a particular character in the range [*first*...*last*] is missing from the font, the offset/width table entry for the character is –1.

5) *first*

   The ASCII code of the first character implemented in the font. For example, ASCII character codes 0 through 31 may represent control functions that are nonprinting. If the first implemented character in a font is a space, with an ASCII code of 32, then the *first* field is set to 32.

6) *last*

   ASCII code of last character implemented in font.

7) *maxwide*

The maximum character width. This is the width of the widest character in the font.

8) *maxkern*

The maximum amount by which any character in the font kerns, expressed a positive horizontal distance measured in pixels. The descender of a character such as a lower-case *j* may extend or *kern* beneath the character to its left. The amount of kerning is measured as the offset from the character origin to the left edge of the character image. For example, if the maximum amount any character in the font kerns to the left of the origin is 3, the *maxkern* value is specified as +3.

9) *charwide*

The fixed character width in the case of a block font. For a proportionally spaced font, this field is set to 0, in which case the width for each individual character appears as an entry in the offset/width table.

10) *avgwide*

Average width of all characters implemented in the font. This value is the sum of all the character widths divided by the number of characters in the font. This parameter is useful for selecting a best-fit font at a particular target display resolution.

11) *charhigh*

The font height. This is the sum of the *ascent* and *descent* fields, and is a constant across all characters within a particular font.

12) *ascent*

The distance in pixels from the base line to top of the highest character, specified as a positive number.

13) *descent*

The distance in pixels from base line to bottom of lowest descender, specified as a positive number.

14) *leading*

The vertical spacing in pixels from the bottom of one line of text to the top of the next line of text, specified as a positive number.

15) *rowpitch*

The pitch per row of the pattern table. This is the difference in bit addresses from the start of one row in the pattern table bit map to the start of the next row. The TMS340 graphics processor's addresses point to bit boundaries in memory, and each row must start on an even 16-bit word boundary; hence, the *rowpitch* value is always a multiple of 16.

16) *oPatnTbl*

The pattern table offset. This is the difference in bit addresses from the start of the FONT structure (*magic* field) to the start of the pattern table. This field is expressed as a positive value that is an even multiple of 16 (the word size).

17) *oLocTbl*

The location table offset. This is the difference in bit addresses from the start of the FONT structure (*magic* field) to the start of the location table. This field is expressed as a positive value that is an even multiple of 16 (the word size).

18) *oOwTbl*

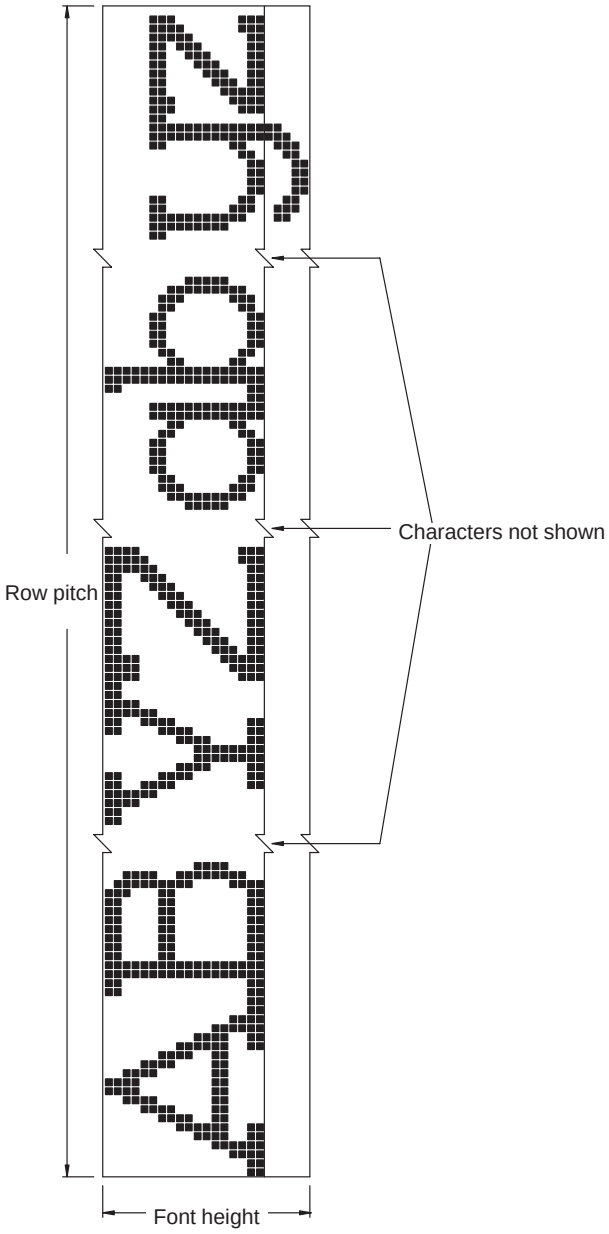
The offset/width table offset. This is the difference in bit addresses from the start of the FONT structure (*magic* field) to the start of the offset/width table. This field is expressed as a positive value that is an even multiple of 16 (the word size).

5.2.2 Font Pattern Table

The font pattern table is a two-dimensional bit map organized as shown in Figure 5–3. The table contains the character images for all characters implemented in the font, concatenated in order from left to right. The width of the table (number of bits per row) is the sum of the individual character widths and must be less than or equal to the pitch specified in the *rowpitch* field of the FONT structure. The number of rows is equal to the value contained in the *charhigh* field. The total number of bits in the bit map is obtained by multiplying *rowpitch* by *charhigh*. The base address of the table is the address of the bit located in the top left corner of the bit map. The top row of the bit map contains the top row of each character shape, stored in

left-to-right order; the second row from the top contains the second row of each character shape, and so on.

Figure 5–3. Bit-Mapped Font Representation



5.2.3 Location Table

The location table specifies the locations of the images for the individual characters in the pattern table. Each location table entry is 16 bits. One entry is provided for each character code in the range [*first...last*]. The table contains one additional entry, and the total number of entries is *(last – first* + 2).

The table entry for each character is the bit displacement from the base address of the pattern table (the address of the leftmost bit in the top row of the bit map in Figure 5–3) to the top left corner of the corresponding character image. The image width for a particular image is just the difference between the location table entries for that character and for the character that immediately follows it. The location table contains entries for all character codes from *first* to *last*, and an additional entry that is used to calculate the image width of the last character. The final location table entry is the offset of the first bit past the right edge of the top row in Figure 5–3.

If a particular ASCII character *n* in the range [*first...last*] is missing from the font, the image width is 0. In other words, location table entries *n–first* and *n–first*+1 contain the same offset value.

5.2.4 Offset/Width Table

The offset/width table contains the character offset and character width for all characters in the range [*first...last*] that are implemented in the font. (Refer to the definitions of the terms *character offset* and *character width* earlier in this section.) Each offset/width table entry is 16 bits. One entry is provided for each character code in the range [*first...last*]. The table also contains one final entry that is always set to –1, and the total number of entries is (*last – first* + 2).

The table entry for each character implemented in the font is an 8-bit character offset concatenated with an 8-bit character width. The offset is in the 8 MSBs of the word, and the width is in the 8 LSBs. If a particular ASCII character in the range [*first...last*] is missing from the font, the corresponding 16-bit entry is set to –1.

5.3 Proportionally Spaced Versus Block Fonts

Two varieties of fonts are distinguished by the value of the *charwide* field in FONT structure. A proportionally spaced font is identified by a *charwide* value of 0, while a nonzero *charwide* value identifies a block font. The system font, which is permanently installed in the font table as font number 0, is always a block font. The installable fonts may be either proportionally spaced or block fonts.

In the case of a proportionally spaced font, the character width is permitted to vary from one character to the next. The character image may cover only a portion of the character width. In other words, the character image does not necessarily overwrite the spaces separating successive characters in a string displayed on the screen. To replace an old line of text on the screen with a new line, the old line typically must be erased completely. If this is not done, portions of the old characters may be visible between the new characters. Also, the space (ASCII code 32) character causes the character pointer to move to the right on the screen but may not cause any pixels to actually be modified. Using space characters from a proportionally spaced font to erase a line of text is generally an ineffective technique.

In the case of a block font, on the other hand, the character width is uniform across all characters implemented in the font. The character image completely spans the character width, even in the case of a space character. Writing a string of characters, which may include spaces, to the screen completely overwrites an old line of characters lying beneath it.

This discussion assumes that the pixel-processing *replace* operation is in effect and that transparency is disabled. Different effects can be achieved by altering pixel processing and transparency, as described in the user's guides for the TMS34010 and TMS34020. The *replace* operation with transparency enabled may be particularly useful in applications requiring proportionally spaced text.

5.4 Font Table

The system font, permanently installed in the library's font table as font number 0, is always a block font. Additional fonts can be installed in the table and can be any combination of proportionally spaced and block fonts. The installable fonts are assigned table indices 1, 2, and so on by the library as they are installed, and the fonts are thereafter identified by these indices during text operations.

The font table is simply an array of pointers to the data structures for the installed fonts. The maximum number of entries available in the font table is fixed for a particular system but may vary from one system to another. In all systems, the font table will be large enough to contain at least 16 installed fonts (in addition to the permanently installed system font). An attempt to install an additional font in a table that is already full will return an error code. Refer to the description of the *install_font* function in Chapter 7 for details.

5.5 Text Attributes

The library provides application programs with direct control over three text attributes:

1) *Text Alignment*

   The position of the character origin (see definition in Section 5.1) for each character is located at the base line or top edge of the character. The default is the top edge.

2) *Additional Intercharacter Spacing*

   An amount by which the default character width (see definition) specified within the font data structure is increased. The default is 0.

3) *Intercharacter Gaps*

   The gaps between horizontally adjacent characters, which can be automatically filled with the background color. When this attribute is enabled, one line of proportionally spaced text can be cleanly written directly on top of another without first erasing the text underneath. When the attribute is disabled, only the rectangular area immediately surrounding each character image (see definition of image width) is filled with the background color. By default, the filling of intercharacter gaps is disabled.

Only proportionally spaced fonts are affected by the state of these attributes. In the case of a block font, the text alignment is always to the top left corner of each character, the intercharacter spacing is fixed at the *charwide* value defined in the font structure, and intercharacter gaps are always filled.

5.6 Available Fonts

The TMS340 Graphics Library includes a bit-mapped font database consisting of 20 typefaces available in a variety of sizes. The size of a font is given in terms of its height in pixels. This height is specified as the sum of its ascent and descent parameters. The available fonts are summarized in Table 5–2.

Several of the fonts in Table 5–2 are labeled as *monospaced* (represented as an *M* in the rightmost column) rather than *proportionally spaced*. A monospaced font is characterized by uniform character width across the font but is otherwise to be distinguished from the block fonts described previously. The monospaced fonts in Table 5–2 use the same font data structure as the proportionally spaced fonts. In particular, the *charwide* field is 0, and the structure includes an offset/width table.

Table 5–2. Font Database Summary

Font Name	Font Size in Pixels														Type†
Arrows							25		31						M
Austin		11		15			20		25			38		50	P
Corpus Christi		15		16			26		29			49			M
Devonshire							23		28			41			P
Fargo							22		26			38			P
Galveston			12	15			21	22	28			42			P
Houston				14	17		20		26			38		50	P
Luckenbach	07														P
Math				16	19		24		32			44		64	P
San Antonio							22		28			40			P
System					16		24								B
Tampa					18		22		30			42			P
TI Art Nouveau							22		28			41		54	82	P
TI Bauhaus		11		14	17	19	22	24	28			43		56		P
TI Cloister									27			40				P
TI Dom Casual							23	25	30			42	46			P
TI Helvetica		11		15	18	20	22	24	28	32	36	42		54	82	P
TI Park Avenue				15	18	21	23	25	28			43		54		P
TI Typewriter Elite		11		14	16	18	20	22	26			38				M
TI Roman		11		14	16	18	20	22	26	30	33	38		52	78	P
	05	09	10	12	14	16	18	20	24	28	32	36	40	48	72	

Point size equivalents at 640 × 480 screen resolution

† P = Proportional spacing M = Mono spacing B = Block font

5.6.1 Installable Font Names

The application program must be linked with the fonts that are used by the application. Within the program, each font is referred to by an external name that uniquely identifies it. The external names for the available fonts are presented in Table 5–3. These names refer to the fonts from a C program. To refer to the fonts from a TMS340 assembly language program, precede each font name with an underscore character.

Table 5–3. Installable Font Names

Font Name	External Name
Arrows font sizes 25 and 31:	arrows25, arrows31
Austin font sizes 11 through 50:	austin11, austin15, austin20, austin25, austin38, austin50
Corpus Christi font sizes 15 through 49:	corpus15, corpus16, corpus26, corpus29, corpus49
Devonshire font sizes 23 through 41:	devons23, devons28, devons41
Fargo font sizes 22 through 38:	fargo22, fargo26, fargo38
Galveston font sizes 12 through 42:	galves12, galves15, galves21, galves22, galves28, galves42
Houston font sizes 14 through 50:	houstn14, houstn17, houstn20, houstn26, houstn38, houstn50
Luckenbach font size 7:	lucken07
Math font sizes 16 through 64:	math16, math19, math24, math32, math44, math64
San Antonio font sizes 22 through 40:	sanant22, sanant28, sanant40
System font sizes 16 and 24	sys16, sys24
Tampa font sizes 18 through 42:	tampa18, tampa22, tampa30, tampa42
TI Art Nouveau font sizes 22 through 82:	ti_art22, ti_art28, ti_art41, ti_art54, ti_art82
TI Bauhaus font sizes 11 through 56:	ti_bau11, ti_bau14, ti_bau17, ti_bau19, ti_bau22, ti_bau24, ti_bau28, ti_bau43, ti_bau56
TI Cloister font sizes 27 and 40:	ti_clo27, ti_clo40
TI Dom Casual font sizes 23 through 46:	ti_dom23, ti_dom25, ti_dom30, ti_dom42, ti_dom46
TI Helvetica font sizes 11 through 82:	ti_hel11, ti_hel15, ti_hel18, ti_hel20, ti_hel22, ti_hel24, ti_hel28, ti_hel32, ti_hel36, ti_hel42, ti_hel54, ti_hel82
TI Park Avenue font sizes 15 through 54:	ti_prk15, ti_prk18, ti_prk21, ti_prk23, ti_prk25, ti_prk28, ti_prk43, ti_prk54
TI Roman font sizes 11 through 78:	ti_rom11, ti_rom14, ti_rom16, ti_rom18, ti_rom20, ti_rom22, ti_rom26, ti_rom30, ti_rom33, ti_rom38, ti_rom52, ti_rom78
TI Typewriter Elite font sizes 11 through 38:	ti_typ11, ti_typ14, ti_typ16, ti_typ18, ti_typ20, ti_typ22, ti_typ26, ti_typ38

The System font sizes 16 and 24 appearing near the middle of Table 5–2 are the only two block fonts. One of these is typically designated as the *system font* (the permanently installed font number 0) in a particular graphics mode. These fonts can also be installed in the font table in the same manner as the other fonts in Table 5–2.

Example The following example demonstrates how to access an external FONT structure from a C program. The global name of the TI Helvetica size 22 font, *ti_hel22*, is declared as an external structure of type FONT. (The reference must be resolved by linking the font with the program.) The font pointer is passed to the font table via the *install_font* function, and the *select_font* function is used to select the font. Finally, the *text_out* function is used to print the string “hello, world” to the screen in the selected font.

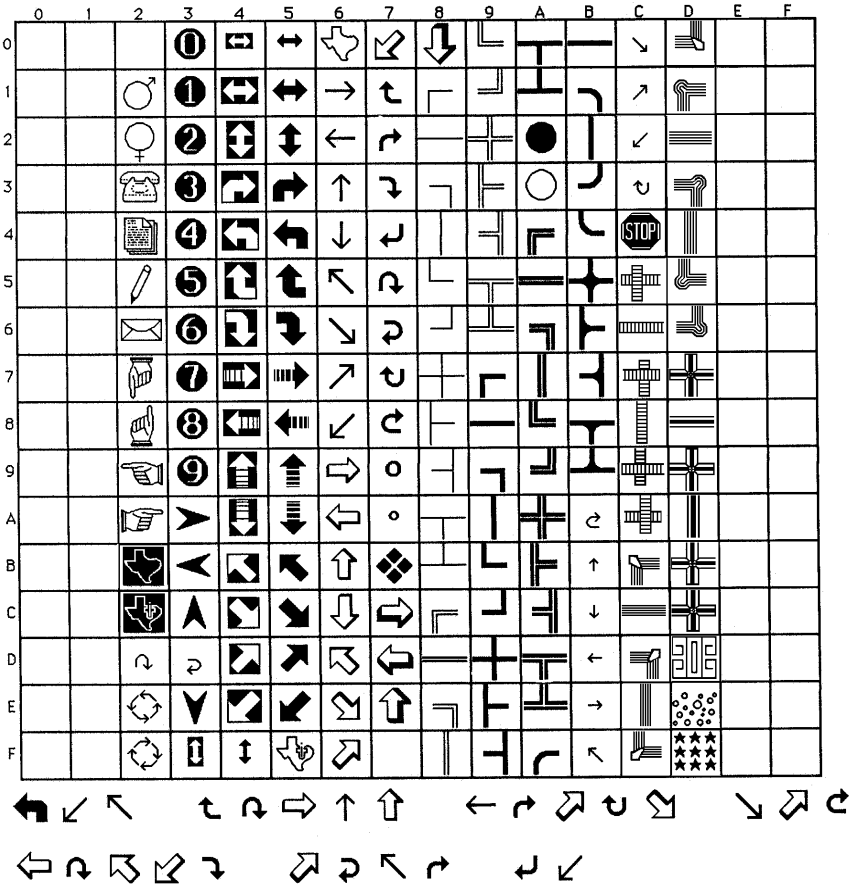
```
#include <gsptypes.h>   /* defines FONT structure */

extern FONT ti_hel22;

main()
{
    short n;

    set_config(0, !0);
    clear_screen(–1);

    n = install_font(&ti_hel22);
    select_font(n);
    text_out(10, 10, ”hello, world”);
}
```

5.6.2 Alphabetical Listing of Fonts

Each of the fonts included with the library is described briefly in the remainder of this section. Each typeface is presented separately, along with the list of available font sizes, spacing, and recommendations regarding the use of the face. Illustrations of each font are also presented at approximately true scale to indicate the relative dimensions of the various font sizes available for each typeface. The actual physical size of a font will vary, depending on the dimensions of the display device.

Spacing Monospace
Derivation Original character set, no typesetter's equivalent
Description Graphic accents, arrows, and symbols suitable for use in memos, transparencies, posters, flyers, and newsletters.
Sizes 25 and 31 pixels

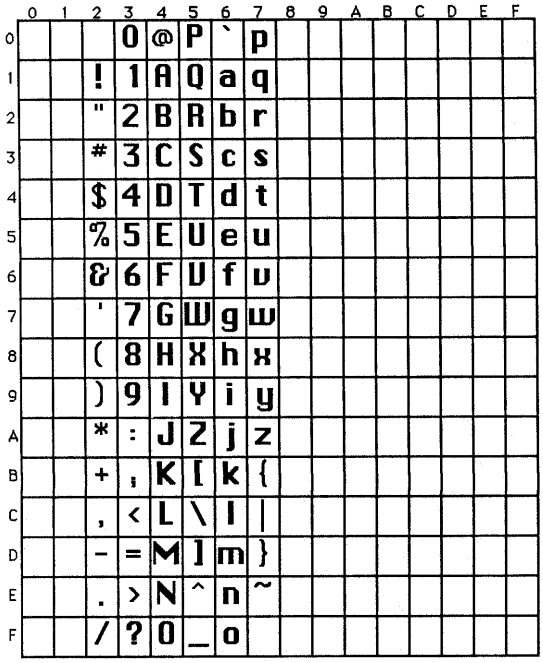
Example



austin *Austin Fonts*

Spacing	Proportional
Derivation	Original typeface, no typesetter's equivalent
Description	An upright, bold-weight, sans-serif typeface. Suited to many purposes. Smaller sizes serve well for general usage as body text or headings, while larger sizes are ideal for headlines and titles.
Sizes	11, 15, 20, 25, 38, and 50 pixels

Example



Taste in printing determines the form typography takes. The
of a congruous typeface, the quality and suitability for its pur
paper to be used, the care and labor,

Taste in printing determines the form typograp
takes. The selection of a congruous typeface,
quality and suitability for its purpose,

Taste in printing determines the f
typography takes. The selection o

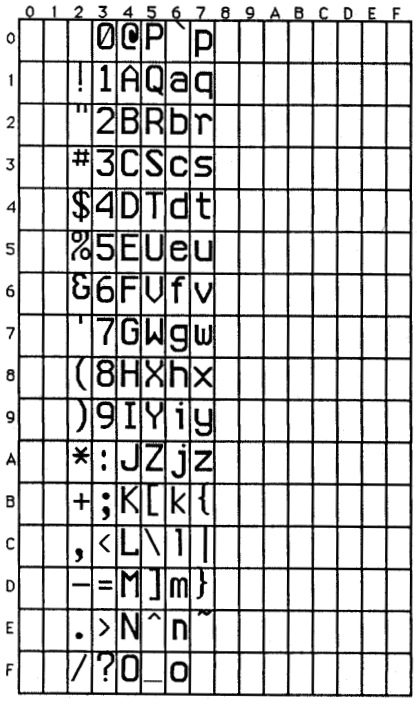
Taste in printing determin
the form typography takes
The selection of a

Taste in printing
determines the form
typography takes.

Taste in
printing
determines th

Spacing Monospace
Derivation Original character set, no typesetter's equivalent
Description Designed as a terminal display font. 16-pixel size renders a standard 80-column display at 640 × 480 resolution. 29-pixel renders a 40-column display at the same resolution. Light- to bold-weight, depending on size.
Sizes 15, 16, 26, 29, 49 pixels

Example




```
Taste in printing determines the form typogra
takes. The selection of a congruous typeface,
quality and suitability for
```

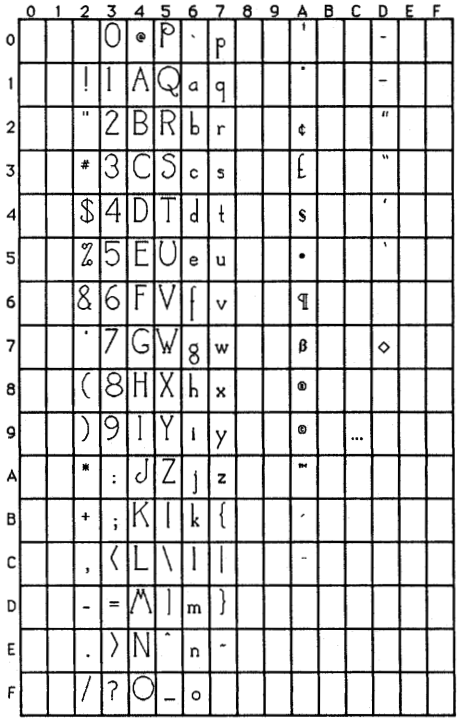
```
Taste in printing determines the form
typography takes. The selection of a co
typeface, the quality and
```

Taste in printing determ
the form typography take
The selection

Taste in printing
determines the form
typography takes. Th

Taste in
printing
determines the
form

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description A light-weight, stylized serif typeface. Elongated ascenders and descenders distinguish this font. Suitable for invitations, newsletters, flyers, or anything requiring a formal appearance.
Sizes 23, 28, and 41 pixels
Example



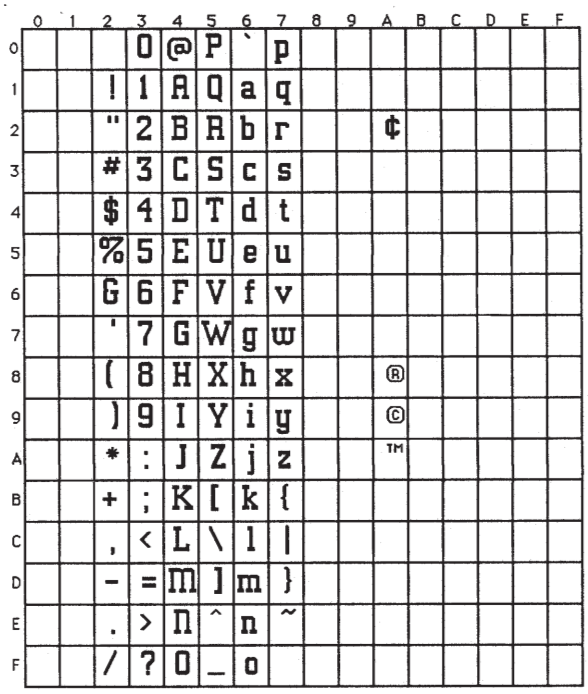
Taste in printing determines the form typography takes. selection of a congruous typeface, the quality and suitab its purpose, the paper to be used,

Taste in printing determines the form typo

takes. The selection of a congruous typeface quality and suitability for

Taste in printing determines the typography takes. The selection congruous

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description An upright, medium-weight serif face. Small sizes suited for diagrams and labels. Larger sizes are well suited to headlines and posters.
Sizes 22, 26, and 38 pixels
Example



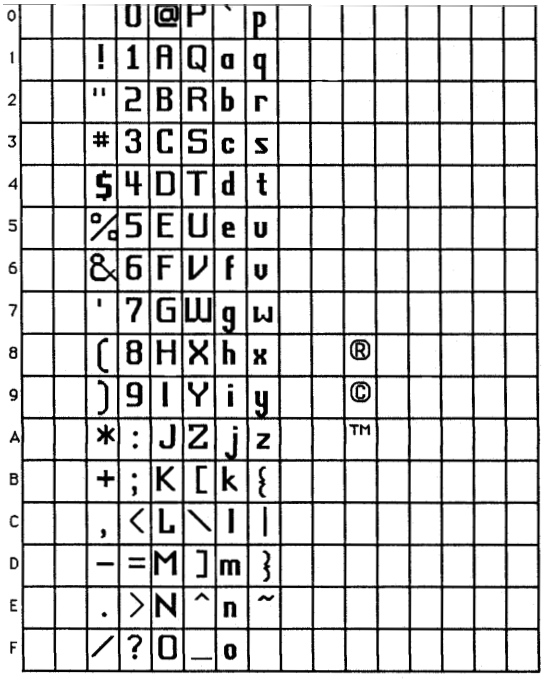
Taste in printing determines the form typography takes. The selection of a congruous typeface, the

Taste in printing determines form typography takes. The selection of a

Taste in printing determines the form typography takes.

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description An upright, bold-weight serif face. Suited to many purposes. Smaller sizes serve well for general usage as body text or headings, while larger sizes are ideal for headlines and titles.
Sizes 12, 15, 21, 22, 28, and 42 pixels

Example



Taste in printing determines the form typography takes. T selection of a congruous typeface, the quality and suitability purpose, the paper to be used, the care

Taste in printing determines the form typography take selection of a congruous typeface, the quality and sui for its purpose, the paper to be

Taste in printing determines the for

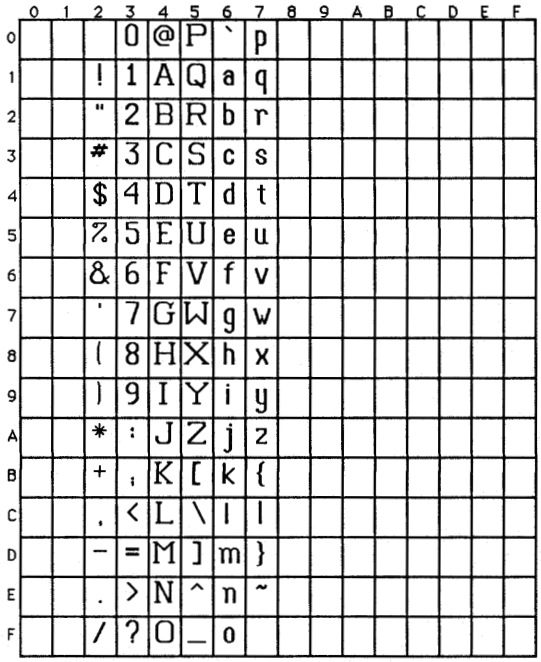
typography takes. The selection of congruous typeface, the

Taste in printing determines form typography takes. The selection of a

Taste in printing determine the form typography takes. The selection of a

Taste in printing determines the form typography takes.

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description An upright, light-to-medium-weight serif typeface. Suited to many purposes. Smaller sizes serve well for general usage as body text or headings while larger sizes are ideal for headlines and titles.
Sizes 14, 17, 20, 26, 38, and 50 pixels
Example



Taste in printing determines the form typography takes. The sele
congruous typeface, the quality and suitability for its purpose, th
be used, the care

Taste in printing determines the form typography
The selection of a congruous typeface, the quality
suitability for its purpose, the paper

Taste in printing determines the form typog

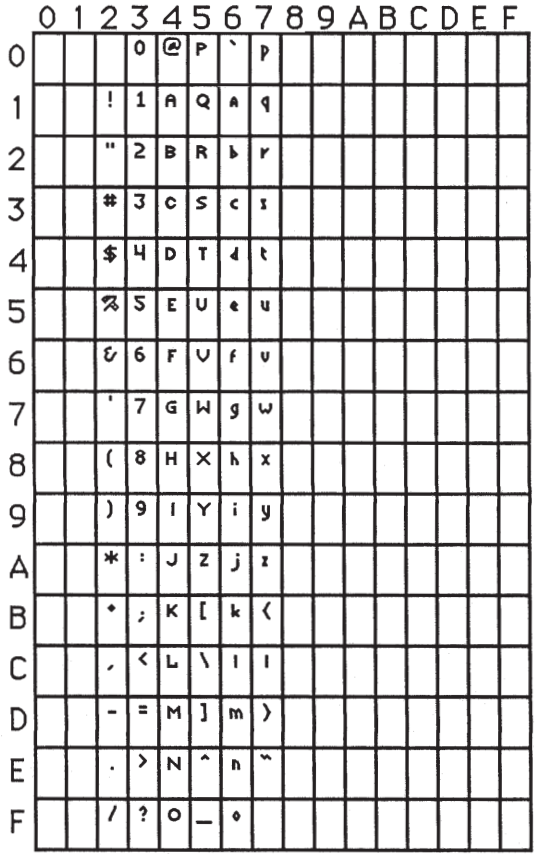
takes. The selection of a congruous typeface
quality and suitability for its

Taste in printing determines the
form typography takes. The selec
of a congruous

Taste in printing determ
the form typography tak
The

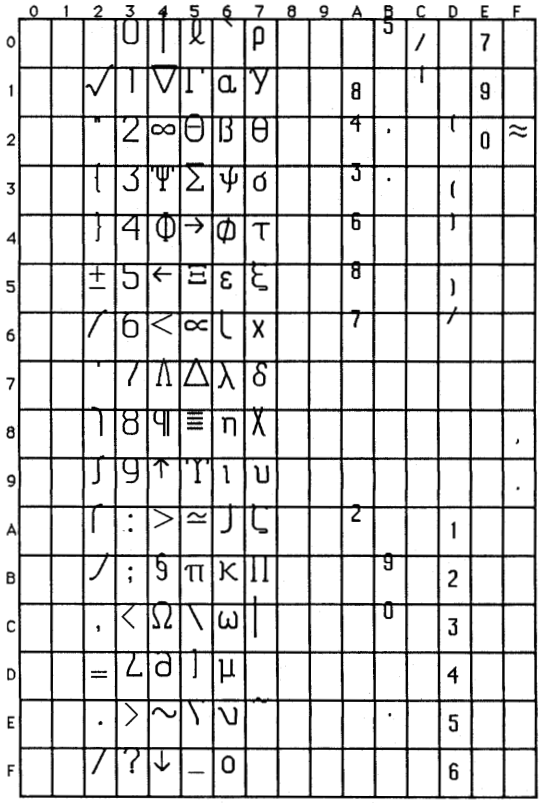
Taste in printing
determines the fo

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description Designed as the smallest legible font at 640 × 480 resolution. Useful for diagrams or any other task requiring very small text.
Sizes 7 pixels
Example



Taste in printing determines the form typography takes. The selection of a congruous type and suitability for its purpose, the paper to be used, the care

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description Math and Greek symbols, including subscripts and superscripts. Light- to medium-weight, depending on size.
Sizes 16, 19, 24, 32, 44, and 64 pixels
Example



→αστε ιν ρθιντινλ φετεθμινεσ τηε λοθμ τυρολθα τακεσ. →ηε σεωεψτιον ολ α ψονλθξοξσ τυρελαψε, ·
ϒξαιωιτυ ανφ σξιταβιωιτυ

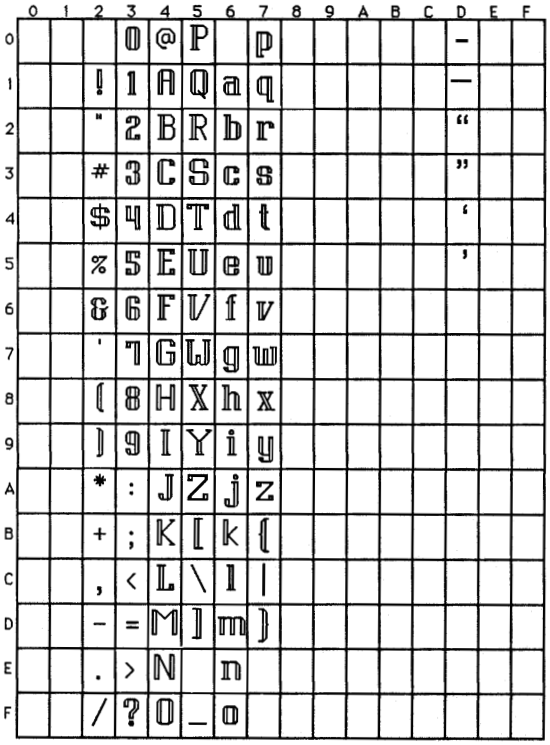
→αστε ιν ρθιντινλ φετεθμινεσ τηε λοθμ
τυρολθαρηυ τακεσ. →ηε σεωεψτιον ολ α ψον
τυρελαψε, τηε ϒξαιωιτυ ανφ

→αστε ιν ρθιντινλ φετεθμινεσ τηε l τυρολθαρηυ τακεσ. →ηε σεωεψτιον ol ψονλθξοξσ τυρειαψε, τηε

→αστε ιν ρθιντινλ φετεθμινεσ τηε loθμ τυρολθαρηυ τακεσ. →ηε

→αστε ιν ρθιντινλ φετεθμινεσ τr

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description A serif typeface with hollow (commonly called *in-line*) uprights. Distinctive and semiformal in appearance, ideal for memos, newsletters, flyers, and headings.
Sizes 22, 28, and 40 pixels
Example



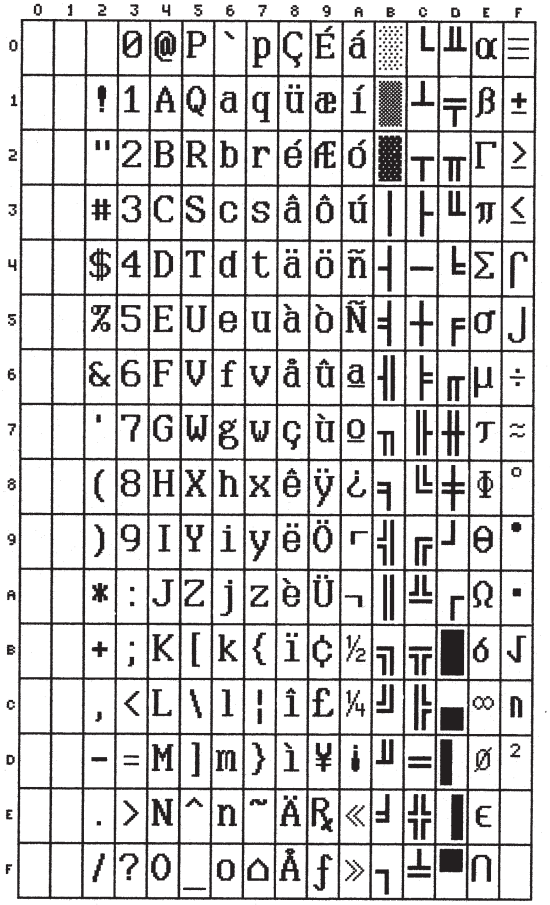
Taste in printing determines the typography takes. The selection o congruous

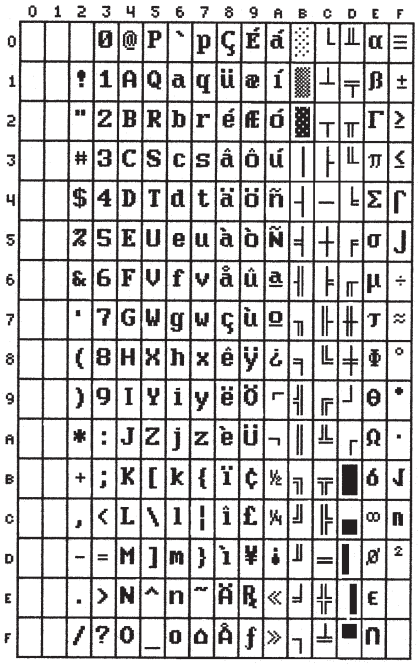
Taste in printing determir

the form typography takes selection

Taste in printing determines the form

Spacing Monospaced (block font)
Derivation Original character set, no typesetter's equivalent
Description Designed to emulate character-ROM fonts displayed by text terminals. The smaller size is suitable for low- to medium-resolution displays. The larger size is suitable for high-resolution displays of 1024-by-768 and above. The characters defined within this font are compatible with the IBM EGA/VGA extended character set.
Sizes 16 and 24 pixels
Example

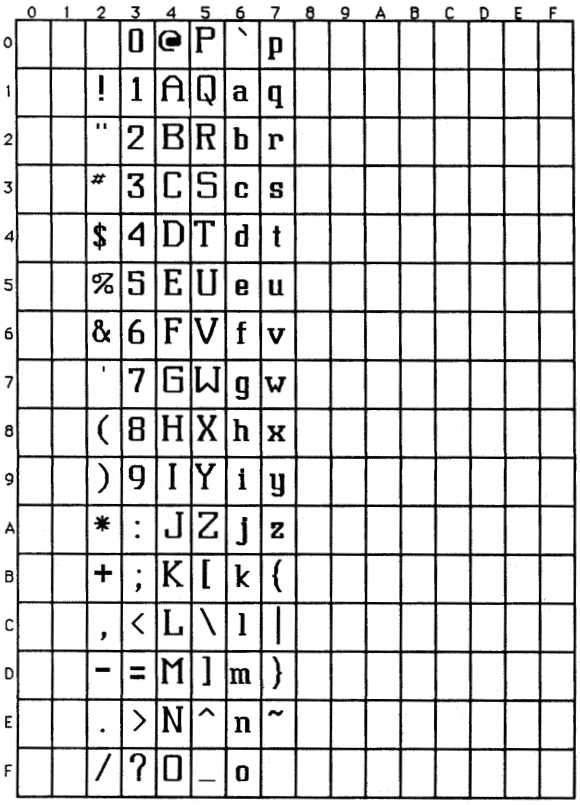




```
Taste in printing determines the form typography t
The selection of a congruous typeface, the quality
suitability for its purpose, the paper to be used,
care and labor,
```

Taste in printing determines the
The selection of a congruous type
suitability for its purpose, the
care and labor,

Spacing Proportional
Derivation Original character set, no typesetter's equivalent
Description A bold- to medium-weight serif typeface. Small sizes suited for diagrams and labels. Larger sizes are well suited to headlines and posters.
Sizes 18, 22, 30, and 42 pixels
Example



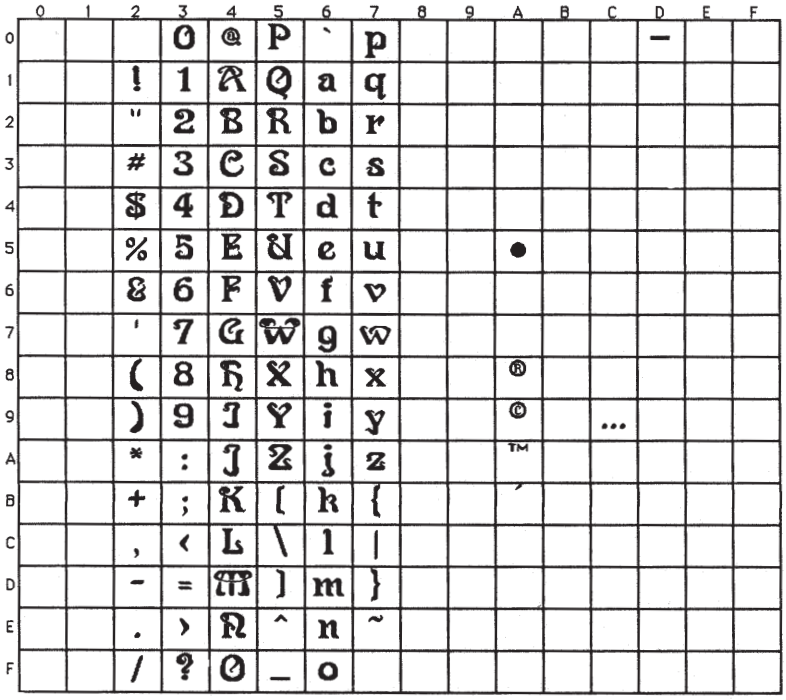
Taste in printing determines the form typography takes. The selection of a con typeface, the quality and suitability for : purpose,

Taste in printing determines the form typography takes. The selection of a congruous typeface, the quality and

Taste in printing determ the form typography tak The selection of a congr

Taste in printing determines the for typography takes.

Spacing Proportional
Derivation Art Nouveau
Description A bold-weight, stylized serif typeface. Very ornate; perfect for flyers, posters, and newsletters.
Sizes 22, 28, 41, 54, and 82 pixels
Example



Taste in printing determines form typography takes. The selection of a congruous typ the quality and

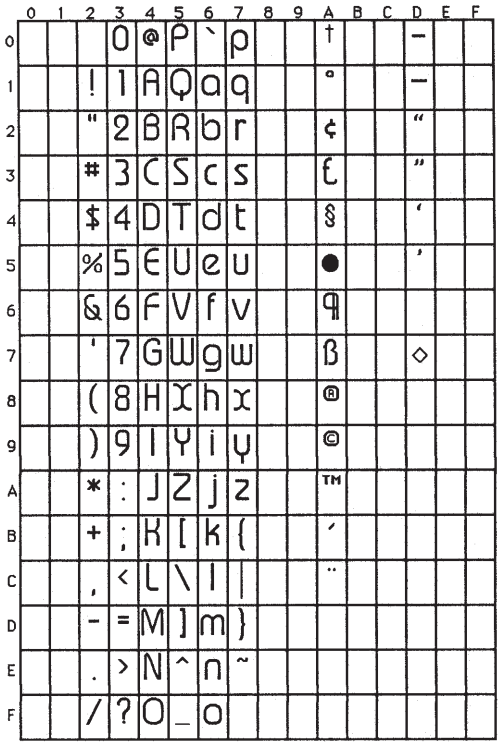
Taste in printing

determines the form
typography takes. The
selection of a congruou

Taste in printin
determines the
form typograph
takes.

Taste in
printing
determines
the form

Spacing Proportional
Derivation Bauhaus Medium
Description A medium-weight sans-serif typeface. General purpose font suited to all uses. Commonly seen on business cards, letterheads, magazines, and other publications.
Sizes 11, 14, 17, 19, 22, 24, 28, 43, 56 pixels
Example



Taste in printing determines the form typography takes selection of a congruous typeface, the quality and suitc its purpose, the paper to be used, the care

Taste in printing determines the form typography The selection of a congruous typeface, the quality suitability for its purpose, the paper to

Taste in printing determines the form typogr
takes. The selection of a congruous typefac
quality and suitability for

Taste in printing determines the form
typography takes. The selection of a
congruous typeface, the quality and suitc

Taste in printing determines the f
typography takes. The selection c
congruous typeface, the

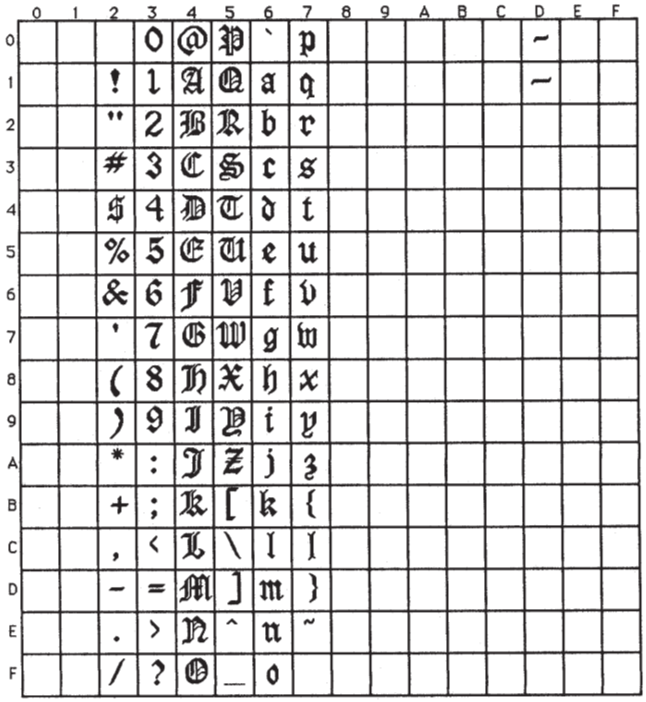
Taste in printing determines th
form typography takes. The
selection of a

Taste in printing determin
the form typography takes
The selection of a

Taste in printing
determines the fo


Taste in
printing
determines th
form

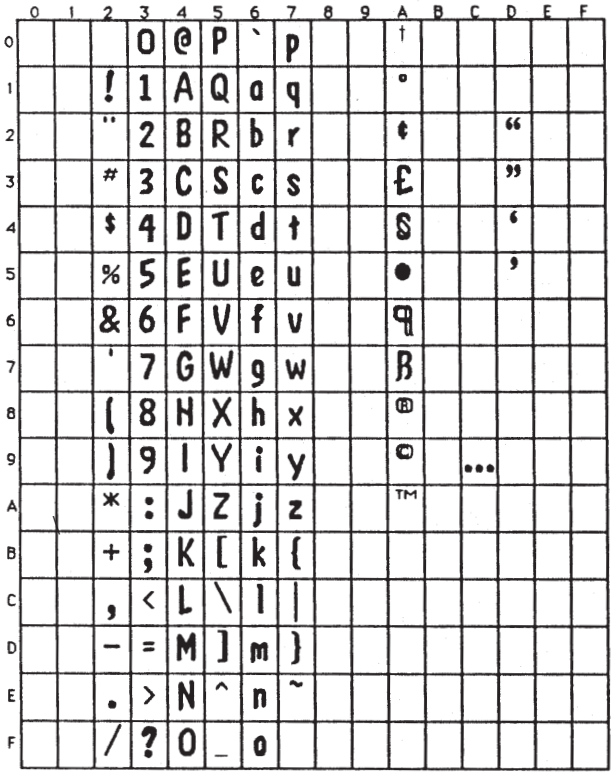
Spacing Proportional
Derivation Cloister Black
Description A highly stylized, bold-weight Olde English typeface. Best suited for invitations, posters, and flyers. Very decorative.
Sizes 27 and 40 pixels



Taste in printing determ
the form typography takes.
selection

Taste in printing

Spacing Proportional
Derivation Dom Casual
Description A bold-weight semi-cursive typeface. Distinctive and informal. Ideal for newsletters, posters, and flyers.
Sizes 23, 25, 30, 42, and 46 pixels



Taste in printing determines the form typography takes. The selection of a congruous typeface, the quality and

Taste in printing determines the f

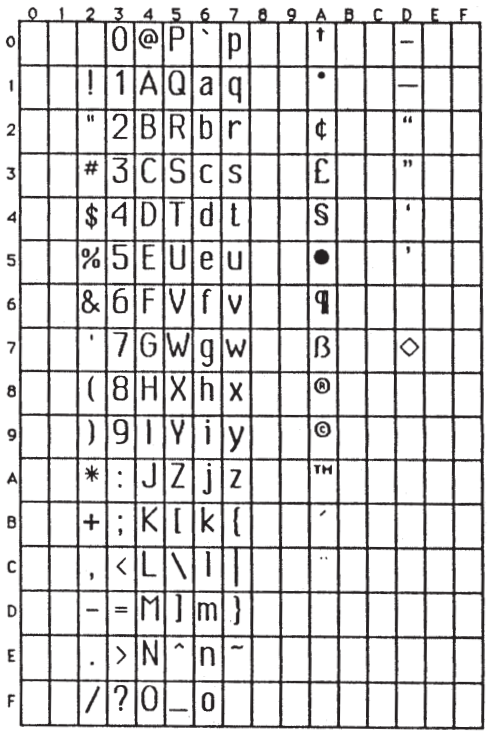
typography takes. The selection of congruous typeface, the

Taste in printing determines form typography takes. The selection of a congruous

Taste in printing determines the form typography takes.

Taste in printing determines the for

Spacing Proportional
Derivation Helvetica
Description A light-weight sans-serif typeface. Patterned after one of the most widely used typefaces in the United States. Appropriate for use in all business-related applications, particularly correspondence and newsletters.
Sizes 11, 15, 18, 20, 22, 24, 28, 32, 36, 42, 54, and 82 pixels
Example



Taste in printing determines the form typography takes. The selec
congruous typeface, the quality and suitability for its purpose, the
used, the care

Taste in printing determines the form typography takes. The se
congruous typeface, the quality and suitability for its purpose,
be used, the care

Taste in printing determines the form typog takes. The selection of a congruous typeface, quality and suitability for its purpose,

Taste in printing determines the form typography takes. The selection of a typeface, the quality and suitability

Taste in printing determines the typography takes. The selection of congruous typeface, the quality

Taste in printing determines the fo typography takes. The selection of congruous typeface, the

Taste in printing determines form typography takes. The selection of a congruous

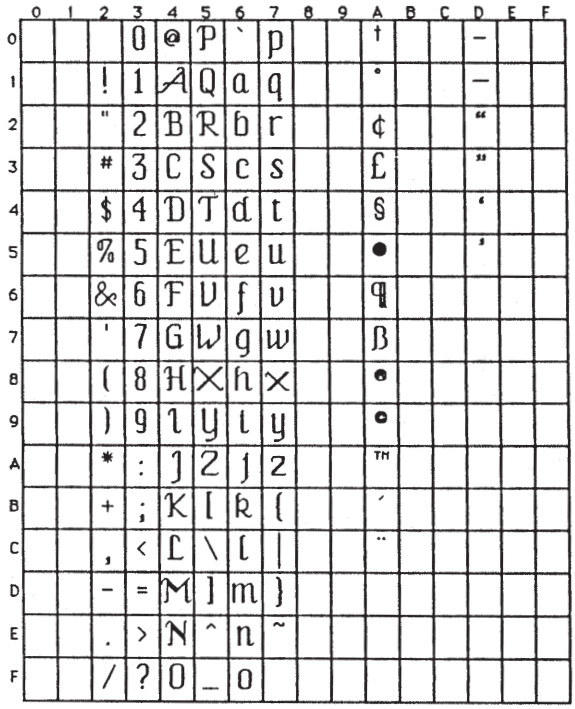
Taste in printing determines form typography takes. The

Taste in printing determines form typography takes. The selection of a

Taste in printing determines the forn typography takes.

Taste in printin determines the form

Spacing Proportional
Derivation Park Avenue/Zapf Chancery
Description A medium-weight, ornate cursive typeface. Suited to many purposes. Commonly seen on wedding invitations but appropriate wherever a formal font is desired.
Sizes 15, 18, 21, 23, 25, 28, 43, and 54 pixels
Example



Taste in printing determines the form typography selection of a congruous typeface, the quality and suitability for its purpose, the paper to

Taste in printing determines the form typography takes. The selection of a con typeface, the quality and suitability

Taste in printing determines the form
typography takes. The selection of a con
typeface, the quality and suitability

Taste in printing determines the forr
typography takes. The selection of a
congruous typeface, the

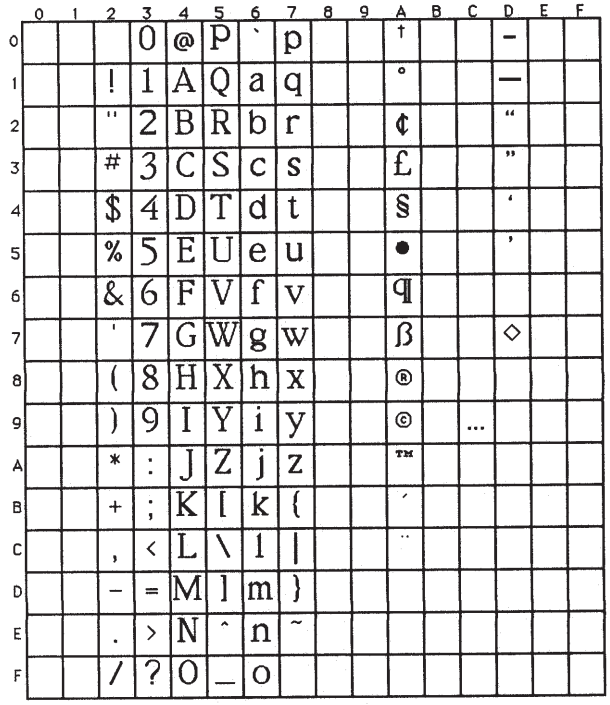
Taste in printing determines the fo
typography takes. The selection of
congruous typeface, the

Taste in printing determines tf
form typography takes. The
selection of a congruous

Taste in printing
determines the for

Taste in printin determines the form

Spacing Proportional
Derivation Times-Roman
Description A light- to medium-weight serif typeface. Patterned after the most widely used typeface in the United States and most English speaking countries. Appropriate for use in all business-related applications, particularly correspondence and newsletters.
Sizes 11, 14, 16, 18, 20, 22, 26, 30, 33, 38, 52, and 78 pixels
Example



Taste in printing determines the form typography tak
selection of a congruous typeface, the quality and sui
its purpose, the paper to be used, the care

Taste in printing determines the form typogı
takes. The selection of a congruous typeface,
quality and suitability for its purpose, the papı
used, the care

Taste in printing determines the form
typography takes. The selection of a cor
typeface, the quality and suitability for it
purpose,

Taste in printing determines the form
typography takes. The selection of a cor
typeface, the quality and suitability

Taste in printing determines the
typography takes. The selection o
congruous typeface, the quality ai

Taste in printing determines
form typography takes. The
selection of a congruous typel
the

Taste in printing determi
the form typography take
The selection of a congrı

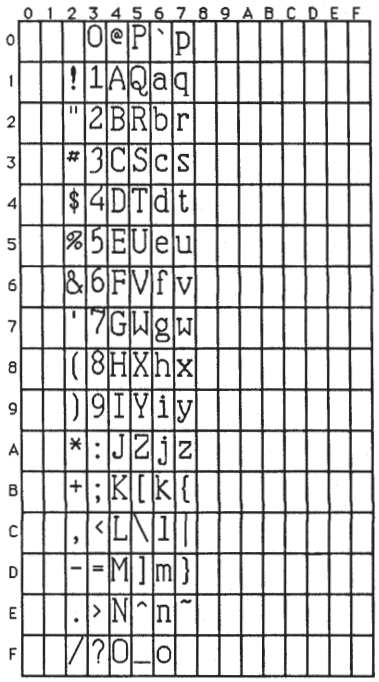
Taste in printing determi
the form typography tak
The selection of a

Taste in printing determ
the form typography tal
The selection of a

Taste in printing
determines the for
typography takes.

Taste in printing determines th form

Spacing	Monospace
Derivation	Typewriter Elite
Description	A light-weight serif typeface. Small sizes suited to correspondence and newsletters. Larger sizes perfect for labels and headlines.
Sizes	11, 14, 16, 18, 20, 22, 26, and 38 pixels
Example	



```
Taste in printing determines the form typography takes. The selection of a typeface, the quality and
```

Taste in printing determines the form typography takes. The selection of a typeface, the quality and

Taste in printing determines the typography takes. The selection of congruous typeface, the

Taste in printing determines the
typography takes. The selection
congruous typeface,

Taste in printing determines
form typography takes. The
selection of a

Taste in printing determin
the form typography takes.
selection of a

Taste in printing
determines the form
typography takes. The

Taste in printin
determines the
form

Chapter 6

Core Primitives

This chapter describes the functions in the Core Primitives Library. The Extended Primitives Library is described in the next chapter.

Remember to call the *set_config* function (a member of the Core Primitives Library) to initialize the drawing environment before you call any of the other functions in the Core and Extended Primitives Libraries.

The table below summarizes the 50 functions in the Core Primitives Library. The remainder of this chapter is an alphabetical, detailed description of the syntax, usage, and operation of each function. These descriptions are augmented by complete example programs that can be compiled and run exactly as written.

Function Name	Description
clear_frame_buffer	Clear frame buffer
clear_page	Clear current drawing page
clear_screen	Clear screen
cpw	Compare point to clipping window
cvxyl	Convert x-y position to linear address
field_extract	Extract field from TMS340 graphics processor memory
field_insert	Insert field into TMS340 graphics processor memory
get_colors	Get colors
get_config	Get hardware configuration information
get_fontinfo	Get font information
get_modeinfo	Get graphics mode information
get_nearest_color	Get nearest color
get_offscreen_memory	Get off-screen memory
get_palet	Get entire palette
get_palet_entry	Get single palette entry
get_pmask	Get plane mask

Function Name	Description
get_ppop	Get pixel processing operation code
get_text_xy	Get text x-y position
get_transp	Get transparency flag
get_vector	Get trap vector
get_windowing	Get window clipping mode
get_wksp	Get workspace information
gsp2gsp	Transfer from one location to another within TMS340 graphics processor memory
init_palet	Initialize palette
init_text	Initialize text
lmo	Find leftmost one
page_busy	Get page busy status
page_flip	Flip display and drawing pages
peek_breg	Peek at B-file register
poke_breg	Poke value into B-file register
rmo	Find rightmost one
set_bcolor	Set background color
set_clip_rect	Set clipping rectangle
set_colors	Set foreground and background colors
set_config	Set hardware configuration
set_fcolor	Set foreground color
set_palet	Set multiple palette entries
set_palet_entry	Set single palette entry
set_pmask	Set plane mask
set_ppop	Set pixel processing operation code
set_text_xy	Set text x-y position
set_transp	Set transparency mode
set_vector	Set trap vector
set_windowing	Set window clipping mode
set_wksp	Set workspace information
text_out	Output text
text_outp	Output text at current x-y position
transp_off	Turn transparency off
transp_on	Turn transparency on
wait_scan	Wait for scan line

Syntax

```
void clear_frame_buffer(color)
unsigned long color;         /* pixel value */
```

Description

The *clear_frame_buffer* function rapidly clears the entire display memory by setting it to the specified color. If the display memory contains multiple display pages (for example, for double-buffered animation) *all* pages are cleared.

Argument *color* is a pixel value. Given a screen pixel depth of N bits, the pixel value contained in the N LSBs of the argument is replicated throughout the display memory. In other words, the pixel value is replicated 32/N times within each 32-bit longword in the display memory. Pixel size N is restricted to the values 1, 2, 4, 8, 16, and 32 bits.

If the value of argument color is specified as –1, the function clears the display memory to the current background color. (In order to clear the frame buffer to all 1s when the pixel size is 32 bits, set the background color to 0xFFFFFFFF and call the *clear_frame_buffer* function with an argument of *–1*.)

This function can rapidly clear the screen in hardware configurations that support *bulk initialization* of the display memory. Bulk initialization is supported by video RAMs that can perform serial-register-to-memory cycles. The serial register in each video RAM is loaded with initialization data and copied internally to a series of rows in the memory array. Whether the function utilizes bulk initialization or some other functionally equivalent method of clearing the screen varies from one implementation to the next.

Off-screen areas of the display memory may also be affected by this function; data stored in such areas may be lost as a result. The *clear_screen* function is similar in operation but does not affect data contained in off-screen areas.

If the graphics display system reserves an area of the display memory to store palette information (as is the case in configurations that use the TMS34070 color palette chip), this area is left intact by the function.

Example Use the *clear_frame_buffer* function to clear the display memory to the default background color. Use the *text_out* function to print a couple of words to the screen.

```
main()
{
    set_config(0, !0);
    clear_frame_buffer(-1);
    text_out(10, 10, "Hello world.");
}
```

Syntax

```
void clear_page(color)
unsigned long color;          /* pixel value */
```

Description

The *clear_page* function rapidly clears the entire drawing page by setting it to the specified pixel value. If the display memory contains multiple display pages (for example, for double-buffered animation), only the current drawing page is cleared.

Given a screen pixel depth of N bits, the pixel value contained in the N LSBs of argument *color* is replicated throughout the drawing page. In other words, the pixel value is replicated 32/N times within each 32-bit longword in the page. Pixel size N is restricted to the values 1, 2, 4, 8, 16, and 32 bits.

If the value of argument *color* is specified as *–1*, the function clears the page to the current background color. (In order to clear the drawing page to all 1s when the pixel size is 32 bits, set the background color to 0xFFFFFFFF and call the *clear_page* function with an argument of *–1*.)

This function can rapidly clear the screen in hardware configurations that support *bulk initialization* of the display memory. Bulk initialization is supported by video RAMs that can perform serial-register-to-memory cycles. The serial register in each video RAM is loaded with initialization data and copied internally to a series of rows in the memory array. Whether the function utilizes bulk initialization or some other functionally equivalent method of clearing the screen varies from one implementation to the next.

The *clear_page* function can affect off-screen as well as on-screen areas of the display memory. Data stored in off-screen areas may be lost as a result. The *clear_screen* function is similar in operation but does not affect data contained in off-screen areas. The *clear_page* function may clear the screen more rapidly than the *clear_screen* function, depending on the implementation.

If the graphics display system reserves an area of the display memory to store palette information (as is the case in configurations that use the TMS34070 color palette chip), this area is left intact by the function.

Example Use the *clear_page* function to clear alternating drawing pages in an application requiring double-buffered animation. The graphics mode selected by the *set_config* function is assumed to support more than one video page. The *text_out* function is used to make the letters *abc* rotate in a clockwise direction around the digits *123*.

```
#define  GMODE    0          /* double-buffered graphics mode */
#define  RADIUS   64         /* radius of rotating text */

main()
{
    long x, y;
    short disppage, drawpage;

    set_config(GMODE, !0);
    drawpage = 0;
    disppage = 1;
    x = RADIUS << 16;
    y = 0;
    for ( ; ; ) {
        page_flip(disppage ^= 1, drawpage ^= 1);
        x -= y >> 5;
        y += x >> 5;
        while (page_busy())
            ;
        clear_page(-1);
        text_out(RADIUS, RADIUS, "123");
        text_out(RADIUS+(x>>16), RADIUS+(y>>16), "abc");
    }
}
```

Syntax

```
void clear_screen(color)
unsigned long color;         /* pixel value */
```

Description

The *clear_screen* function clears the entire current drawing page by setting it to the specified pixel value. If the display memory contains multiple display pages (for example, for double-buffered animation), only the current drawing page is cleared.

Given a screen pixel depth of N bits, the pixel value contained in the N LSBs of argument *color* is replicated throughout the visible screen. In other words, the pixel value is replicated 32/N times within each 32-bit longword in the area of display memory corresponding to the visible screen. Pixel size N is restricted to the values 1, 2, 4, 8, 16, and 32 bits.

If the value of argument *color* is specified as *–1*, the function clears the page to the current background color. (In order to clear the screen to all 1s when the pixel size is 32 bits, set the background color to 0xFFFFFFFF and call the *clear_screen* function with an argument of *–1*.)

The *clear_screen* function does not affect data contained in off-screen areas of the display memory. The *clear_page* function is similar in operation but may affect data contained in off-screen areas; data stored in such areas may be lost as a result. The *clear_page* function may clear the screen more rapidly than the *clear_screen* function, depending on the implementation.

Example

Use the *clear_screen* function to clear the screen to the default background color prior to printing the text “Hello world.” on the screen.

```
main()
{
    set_config(0, !0);
    clear_screen(–1);
    text_out(10, 10, ”Hello world.”);
}
```

Syntax

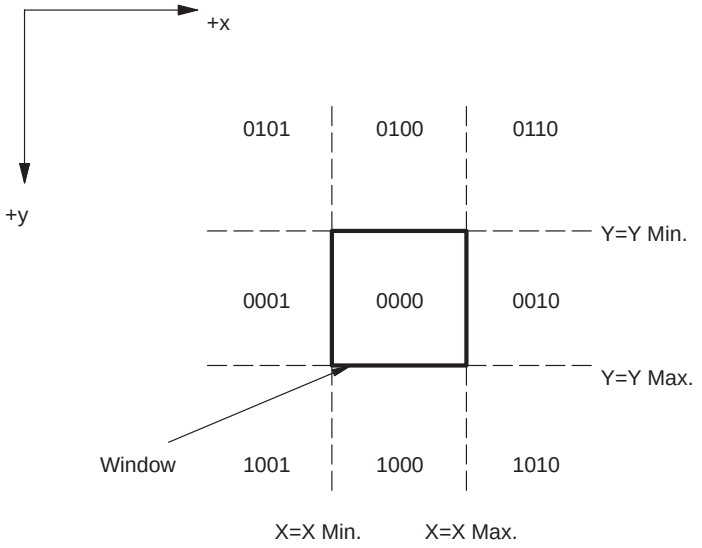
```
short cpw(x, y)
short x, y;                    /* pixel coordinates */
```

Description The *cpw* function returns a 4-bit outcode indicating the specified pixel's position relative to the current clipping window. The outcode indicates whether the pixel is located above or below, to the left or right of, or inside the window.

Arguments *x* and *y* are the coordinates of the pixel, specified relative to the current drawing origin.

The clipping window is rectangular. As shown in Figure 6–1, the area surrounding the clipping window is partitioned into 8 regions.

Figure 6–1. Outcodes for Line Endpoints



Each of the 8 regions is identified by a unique 4-bit outcode. The outcode values for the 8 regions and for the window itself are encoded as follows:

$01XX_2$	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
0000_2	if the point lies within the window

The outcode is right-justified in the 4 LSBs of the return value and zero-extended.

Refer to the user's guide for the TMS34010 or TMS34020 for a detailed description of the outcodes.

Example Use the *cpw* function to animate a moving object that bounces off the sides of the clipping window. When a check of the object's x–y coordinates indicates that it has strayed outside the window, the sign of the object's x or y component of velocity, as appropriate, is reversed. The moving object is an asterisk rendered in the system font. The asterisk is erased by overwriting it with a blank. Note that the system font is a block font; overwriting an asterisk with a blank from a proportionally spaced font might not have the same effect.

```
#define  WCLIP          130      /* width of clipping window */
#define  HCLIP          100      /* height of clipping window */

main()
{
    short x, y, vx, vy;

    set_config(0, !0);
    clear_screen(–1);
    set_clip_rect(WCLIP, HCLIP, 0, 0);
    vx = 2;
    vy = 1;
    for (x = y = 0; ; x += vx, y += vy) {
         text_out(x, y, ”*”);
         if (cpw(x, 0))
             vx = –vx;
         if (cpw(0, y))
             vy = –vy;
         wait_scan(HCLIP);
         text_out(x, y, ” ”);
    }
}
```

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

PTR cvxyl(x, y)
short x, y;                  /* x–y coordinates */
```

Description The *cvxyl* function returns the 32-bit address of a pixel in the TMS340 graphics processor's memory, given the x and y coordinates of the pixel on the screen.

Arguments *x* and *y* are the coordinates of the specified pixel, defined relative to the current drawing origin. If the coordinates correspond to an off-screen location, the calling program is responsible for ensuring that the coordinates correspond to a valid pixel location.

Example Use the *cvxyl* function to determine the base addresses of the all the video pages available in graphics mode 0. The *page_flip* function is used repeatedly to flip to a new page before the *cvxyl* function is called. The *text_out* function is used to print out the 32-bit memory address of each page.

```
#include  <gsptypes.h>

main()
{
    short x, y, n, digit;
    long p;
    char *s, c[80];
    CONFIG cfg;
    FONTINFO fntinf;

    set_config(0, 1);
    clear_screen(–1);
    get_config(&cfg);
    get_fontinfo(–1, &fntinf);
    x = y = 10;
    text_out(x, y, "video page addresses:");
    for (n = 0; n < cfg.mode.num_pages; n++) {
         page_flip(0, n);
         s = &c[strlen(strcpy(c,  "  0x00000000"))];
         for (p = cvxyl(0, 0); p; p /= 16) {
              digit = p & 15;
              *––s = (digit < 10) ? (digit + '0') : (digit + 'A' – 10);
         }
         y += fntinf.charhigh;
         page_flip(0, 0);
         text_out(x, y, c);
    }
}
```

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

unsigned long field_extract(gptr, fs)
PTR gptr;                    /* GSP memory pointer */
unsigned short fs;           /* field size */
```

Description

The *field_extract* function returns the contents of a field located in memory.

Argument *gptr* is a pointer containing the 32-bit address of a field in the TMS340 graphics processor's memory. Argument *fs* specifies the length of the field and is restricted to values in the range 1 to 32 bits.

The function definition places no restrictions on the alignment of the address; the field is permitted to begin at any bit address. Given an *fs* value of N and a *gptr* value of A, the specified field consists of contiguous bits A through A+N–1 in memory.

The contents of the field are placed in the N LSBs of the return value and zero-extended.

Example

Use the *field_extract* function to examine a field from an I/O register located in the TMS340 graphics processor's memory. Retrieve the contents of the PPOP field, a 5-bit field that begins in bit 10 of the CONTROL register.

```
#define  CONTROL   0xC00000B0 /* address of GSP CONTROL reg. */
#define  XOR             10   /* PPOP = XOR */

main()
{
    unsigned long ppop;
    static char c[] = "PPOP = ????";

    set_config(0, !0);
    clear_screen(–1);
    set_ppop(XOR);                          /* load PPOP field */
    ppop = field_extract(CONTROL+10, 5);    /* read it back */
    ltoa(ppop, &c[7]);
    text_out(10, 10, c);
}
```

field_insert *Insert Field into GSP Memory*

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

void field_insert(gptr, fs, val)
PTR gptr;                    /* GSP memory pointer */
short fs;                    /* field size */
unsigned long val;           /* data to be inserted */
```

Description

The *field_insert* function writes a specified value to a field located in the TMS340 graphics processor's memory.

Argument *gptr* is a pointer containing the 32-bit address of a field in the TMS340 graphics processor's memory. Argument *fs* specifies the length of the field and is restricted to values in the range 1 to 32 bits. Argument *val* specifies the value to be written.

The function definition places no restrictions on the alignment of the address; the field is permitted to begin at any bit address. Given an *fs* value of N, and a *gptr* value of A, the specified field consists of contiguous bits A through A+N–1 in memory. The N LSBs of argument *val* are copied into the specified field in memory; the remaining bits of the argument are ignored by the function.

Example

Use the *field_insert* function to load a value into a field in an I/O register located in the TMS340 graphics processor's memory. The PPOP field is a 5-bit field that begins in bit 10 of the CONTROL register. Use the *get_ppop* function to read back the PPOP field, and use the *text_out* function to print its value.

```
#define  CONTROL   0xC00000B0   /* I/O register address */
#define  NOT_OR    13           /* NOT src OR dst –-> dst */

main()
{
    static char c[] = "PPOP = ????";

    set_config(0, !0);
    clear_screen(–1);
    field_insert(CONTROL+10, 5, NOT_OR);     /* load PPOP field */
    ltoa(get_ppop(), &c[7]);                 /* read it back */
    text_out(10, 10, c);
}
```

Syntax

```
void get_colors(fcolor, bcolor)
unsigned long *fcolor;     /* pointer to foreground color */
unsinged long *bcolor;     /* pointer to background color */
```

Description

The *get_colors* function retrieves the pixel values corresponding to the current foreground and background colors.

Arguments *fcolor* and *bcolor* are pointers to long integers into which the function loads the foreground and background colors, respectively. Each pixel value is right-justified within its destination longword and zero-extended.

Example

Use the *get_colors* function to retrieve the default foreground and background pixel values assigned by the *set_config* function. Use the *text_out* function to print the values on the screen.

```
#include <gsptypes.h>              /* defines FONTINFO structure */

static FONTINFO fontinfo;

main()
{
    unsigned long fcolor, bcolor;
    short x, y;
    static char c1[40] = "white = ", c2[40] = "black = ";

    set_config(0, !0);
    clear_screen(-1);
    get_fontinfo(0, &fontinfo);
    get_colors(&fcolor, &bcolor);  /* retrieve colors */
    ltoa(fcolor, &c1[8]);
    x = y = 10;
    text_out(x, y, c1);
    ltoa(bcolor, &c2[8]);
    y += fontinfo.charhigh;
    text_out(x, y, c2);
}
```

Syntax

```
typedef   struct
{
    long   disp_pitch;
    short  disp_vres;
    short  disp_hres
    short  screen_wide
    short  screen_high;
    short  disp_psize;
    long   pixel_mask;
    short  palet_gun_depth;
    long   palet_size;
    short  palet_inset;
    short  num_pages;
    short  num_offscrn_areas;
    long   wksp_addr;
    long   wksp_pitch;
} MODEINFO;

typedef   struct
{
    short  version_number;
    long   comm_buff_size;
    long   sys_flags;
    long   device_rev;
    short  num_modes;
    short  current_mode;
    long   program_mem_start;
    long   program_mem_end;
    long   display_mem_start;
    long   display_mem_end;
    long   stack_size;
    long   shared_mem_size;
    HPTR   shared_host_addr;
    PTR    shared_gsp_addr;
    MODEINFO mode;
} CONFIG;

void get_config(config)
CONFIG *config;     /* hardware configuration info */
```

Description

The *get_config* function retrieves a list of parameters that describe the characteristics of both the hardware configuration and the current graphics mode.

Argument *config* is a pointer to a structure of type CONFIG, into which the function copies parameter values describing the configuration of the display hardware. The last field in the CONFIG structure is a structure of type MODEINFO, which contains parameters describing the currently selected graphics mode.

The fields of the CONFIG structure are defined as follows:

version_number	TIGA revision number, assigned by Texas Instruments Incorporated.
comm_buff_size	Size, in bytes, of the TIGA communications buffer. The contents of this field are undefined in TMS340 Graphics Library applications.
sys_flags	Bits 0–7 indicate which of up to 8 TMS34082 Floating-Point Coprocessors are present in the system. Bits 8–15 are reserved.
device_rev	This is the TMS340 graphics processor silicon revision number, as generated by the processor's REV instruction.
num_modes	Number of graphics modes for boards that allow the switching between different display setups.
current_mode	Mode number corresponding to the current graphics mode.
program_mem_start	Start address of program memory.
program_mem_end	End address of program memory.
display_mem_start	Start address of display memory.
display_mem_end	End address of display memory.
stack_size	Size in bytes of the block of memory allocated for both the system stack and program stack. The two stacks grow toward each other from opposite ends of the block.
shared_mem_size	Size in bytes of dual-ported memory that is shared between the host processor and the TMS340 graphics processor.
shared_host_addr	If *shared_mem_size* is nonzero, this is the start address in host memory of the shared memory; otherwise, it is undefined.
shared_gsp_addr	If *shared_mem_size* is nonzero, this is the start address in TMS340 graphics processor memory of the shared memory; otherwise, it is undefined.

The fields of the MODEINFO structure are defined as follows:

disp_pitch	The display pitch is the difference in memory addresses of two vertically adjacent pixels on the screen.
disp_vres	The display vertical resolution is the number of scan lines on the screen.
disp_hres	The display horizontal resolution is the number of pixels per scan line on the screen.

screen_wide	The width of the active screen in millimeters. In systems in which this dimension is unknown, set to 0.
screen_high	The height of the active screen in millimeters. In systems in which this dimension is unknown, set to 0.
disp_psize	Pixel size (in bits).
pixel_mask	Contains a mask of the active bits within a pixel. Pixel sizes are restricted to powers of 2 in the range 1 to 32. Not all bits within each pixel are necessarily used, however. For example, if the pixel size is 8 bits, but only the video RAM sockets corresponding to the 6 LSBs of each pixel are actually populated, *pixel_mask* is set to 0x3F.
palet_gun_depth	Number of bits per gun in the color palette.
palet_size	Number of entries in the color palette.
palet_inset	For most systems, this field is set to 0. For TMS34070-based boards, which store the palette in the frame buffer, this field contains the length in bits of the palette data that precedes the pixel data for each scan line.
num_pages	Number of video pages (or frame buffers) in display memory. Some systems provide multiple pages to support flickerless animation.
num_offscrn_areas	This is the number of off-screen memory blocks available. This field specifies the number of two-dimensional pixel arrays available in off-screen portions of the display memory. (See description of *get_offscreen_memory* function.)
wksp_addr	Starting linear address in memory of the off-screen workspace area, which is 1 bit per pixel but has the same horizontal and vertical dimensions as the screen.
wksp_pitch	Pitch of off-screen workspace area. If 0, then no off-screen workspace is currently allocated.

Note that the structures described above may change in subsequent revisions. To minimize the impact of such changes, write your application programs to refer to the elements of the structure symbolically by their field names, rather than as offsets from the start of the structure. The include files provided with the library will be updated in future revisions to track any such changes in data structure definitions.

Example Use the *get_config* function to retrieve the pixel size for the current graphics mode. Use the *text_out* function to print the pixel size on the screen.

```
#include <gsptypes.h>               /* defines CONFIG structure */

main()
{
    CONFIG cfg;
    unsigned long psize;
    static char c[] = ”pixel size = ????”;

    set_config(0, !0);
    clear_screen(–1);
    get_config(&cfg);
    psize = cfg.mode.disp_psize;    /* pixel size in bits */
    ltoa(psize, &c[13]);
    text_out(10, 10, c);
}
```

Syntax

```
typedef struct
{
    char facename[30];
    short deflt;
    short first;
    short last;
    short maxwide;
    short avgwide;
    short maxkern;
    short charwide;
    short charhigh;
    short ascent;
    short descent;
    short leading;
    FONT *fontptr;
    short id;
} FONTINFO;

short get_fontinfo(id, pfontinfo)
short id;               /* font identifier */
FONTINFO *pfontinfo;    /* font information */
```

Description

The *get_fontinfo* function copies a structure whose elements describe the characteristics of the designated font. The font must have been previously installed in the font table.

Argument *id* is an index that identifies the font. The system font is always designated as font 0; that is, it is identified by an *id* value of 0. The system font is installed in the font table during initialization of the drawing environment by the *set_config* function. Additional fonts may be installed in the font table by means of calls to the *install_font* function. The *install_font* function returns an identifier value that is subsequently used to refer to the font. The currently selected font is designated by an *id* value of *–1*.

Argument *pfontinfo* is a pointer to a structure of type FONTINFO, into which the function copies parameter values that characterize the font designated by argument *id*.

The function returns a nonzero value if the structure is successfully copied; otherwise, 0 is returned.

The fields of the FONTINFO structure are defined as follows:

facename	String containing font name.
deflt	ASCII code of default character.
first	ASCII code of first character implemented in font.
last	ASCII code of last character implemented in font.
maxwide	Maximum character width.
avgwide	Average width of characters.

maxkern	Maximum character kerning amount.
charwide	Width of characters (0 in the case of a proportionally spaced font).
charhigh	Character height (sum of ascent, descent, and leading).
ascent	Ascent (distance in pixels from base line to top of highest character).
descent	Descent (distance in pixels from base line to bottom of lowest descender).
leading	Leading (vertical spacing in pixels from bottom of one line of text to top of next line of text).
fontptr	Address of font in TMS340 graphics processor's memory.
id	Font identifier (font table index).

Note that the structure described above may change in subsequent revisions. To minimize the impact of such changes, write your application programs to refer to the elements of the structure symbolically by their field names, rather than as offsets from the start of the structure. The include files provided with the library will be updated in future revisions to track any such changes in data structure definitions.

Example Use the *get_fontinfo* function to retrieve the face name, character width, and character height of the system font. Use the *text_out* function to print the three font parameters on the screen.

```
#include <gsptypes.h>     /* defines FONTINFO structure */

main()
{
    FONTINFO fntinf;
    short x, y;
    char c[80];

    set_config(0, !0);
    clear_screen(–1);
    get_fontinfo(0, &fntinf);
    x = y = 10;
    text_out(x, y, fntinf.facename);
    y += fntinf.charhigh;
    strcpy(c, ”character width = ”);
    ltoa(fntinf.charwide, &c[18]);
    text_out(x, y, c);
    y += fntinf.charhigh;
    strcpy(c, ”character height = ”);
    ltoa(fntinf.charhigh, &c[19]);
    text_out(x, y, c);
}
```

Syntax

```
short get_modeinfo(index, modeinfo)
short index;                /* graphics mode index */
MODEINFO *modeinfo;         /* graphics mode information */
```

Description

The *get_modeinfo* function copies a structure whose elements describe the characteristics of the designated graphics mode.

Argument *index* is a number that identifies one of the graphics modes supported by the display hardware configuration. The *index* values are assigned to the available graphics modes by the display hardware vendor. Each configuration supports one or more graphics modes, which are numbered in ascending order beginning with 0.

Argument *modeinfo* is a pointer to a structure of type MODEINFO, into which the function copies parameter values that characterize the graphics mode designated by argument *index*.

The function returns a nonzero value if the mode information is successfully retrieved. If an invalid index is specified, the function returns 0.

The number of graphics modes supported by a particular display configuration is specified in the *num_modes* field of the CONFIG structure returned by the *get_config* function. Given that the number of supported modes is some number N, the modes are assigned indices from 0 to N–1.

The *get_modeinfo* function has no effect on the current graphics mode setting. The display is configured in a particular graphics mode by means of a call to the *set_config* function.

The fields of the MODEINFO structure are defined as follows:

disp_pitch	The display pitch is the difference in memory addresses of two vertically adjacent pixels on the screen.
disp_vres	The display vertical resolution is the number of scan lines on the screen.
disp_hres	The display horizontal resolution is the number of pixels per scan line on the screen.
screen_wide	The width of the active screen in millimeters. In systems in which this dimension is unknown, set to 0.
screen_high	The height of the active screen in millimeters. In systems in which this dimension is unknown, set to 0.
disp_psize	Pixel size (in bits).
pixel_mask	Contains a mask of the active bits within a pixel. Pixel sizes are restricted to powers of 2 in the range 1 to 32. Not all bits within each pixel are necessarily used, however. For example, if the pixel size is 8 bits, but only the video RAM sockets corresponding to the

	6 LSBs of each pixel are actually populated, the pixel_mask is set to 0x3F.
palet_gun_depth	Number of bits per gun in the color palette.
palet_size	Number of entries in the color palette.
palet_inset	For most systems, this field is set to 0. For TMS34070-based boards, which store the palette in the frame buffer, this field contains the length in bits of the palette data that precedes the pixel data for each scan line.
num_pages	Number of display pages in display memory. Some systems provide multiple pages to support flickerless animation.
num_offscrn_areas	This is the number of off-screen memory blocks available. This field specifies the number of two-dimensional pixel arrays available in off-screen portions of the display memory. (See description of *get_offscreen_memory* function).
wksp_addr	Starting linear address in memory of the off-screen workspace area, which is 1 bit per pixel but has the same horizontal and vertical dimensions as the screen.
wksp_pitch	Pitch of off-screen workspace area. If 0, then no off-screen workspace is currently allocated.

Note that the structures described above may change in subsequent revisions. To minimize the impact of such changes, write your application programs to refer to the elements of the structure symbolically by their field names, rather than as offsets from the start of the structure. The include files provided with the library will be updated in future revisions to track any such changes in data structure definitions.

Example Use the *get_modeinfo* function to retrieve a list of the screen resolutions corresponding to the graphics modes supported by the display hardware configuration. Use the *text_out* function to print the list on the screen.

```
#include <gsptypes.h>     /* MODEINFO, CONFIG and FONTINFO */

main()
{
    MODEINFO modinf;
    CONFIG cfg;
    FONTINFO fntinf;
    char c[80];
    short x, y, mode, i;

    set_config(0, !0);
    clear_screen(-1);
    get_config(&cfg);
    get_fontinfo(0, &fntinf);
    x = y = 10;
    for (mode = 0; get_modeinfo(mode, &modinf); mode++) {
         i = strlen(strcpy(c, "mode "));
         i += ltoa(mode, &c[i]);
         strcpy(&c[i], ":  ");
         i = strlen(c);
         i += ltoa(modinf.disp_hres, &c[i]);
         strcpy(&c[i], "-by-");
         i = strlen(c);
         ltoa(modinf.disp_vres, &c[i]);
         text_out(x, y, c);
         y += fntinf.charhigh;
    }
}
```

Syntax

```
unsigned long get_nearest_color(r, g, b, i)
unsigned char r, g, b; /* red, green and blue components */
unsigned char i;       /* gray-scale intensity */
```

Description The *get_nearest_color* function searches the current palette and returns the pixel value whose color is closest to that specified by the input arguments.

If the current graphics mode supports a color display, arguments *r*, *g,* and *b* represent the 8-bit red, green, and blue components of the target color. Each component value corresponds to an intensity value in the range 0 to 255, where 255 is the brightest intensity and 0 is the darkest.

In the case of a gray-scale display, argument *i* represents a gray-scale intensity in the range 0 to 255.

The pixel value returned by the function is right-justified and zero-extended.

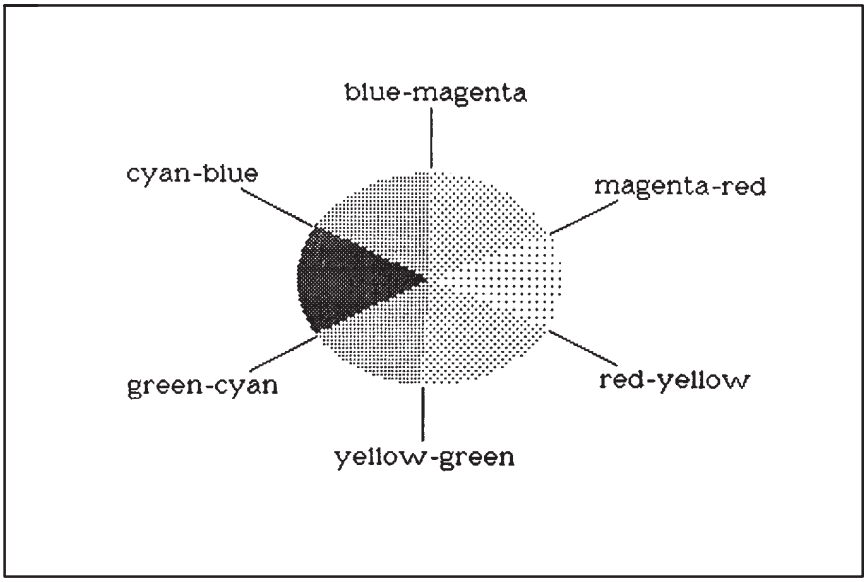
In the case of a gray-scale palette, the return value is the palette index value whose intensity is closest to that specified in argument *i*.

In the case of a color palette, the function performs a more complex calculation. The function calculates the magnitude of the differences between the *r*, *g,* and *b* argument values and the red, green, and blue components, respectively, of each individual color available in the palette. Each of the three differences (red, green, and blue) is multiplied by an individual weighting factor, and the sum of the weighted differences is treated as the effective distance of the color palette entry from the color specified by arguments *r*, *g,* and *b*. The palette entry corresponding to the smallest weighted sum is selected as being nearest to the specified color. The function returns the palette index value corresponding to the selected color.

Example Use the *get_nearest_color* function to determine the pixel values around the perimeter of a color wheel. Use the *fill_piearc* function from the Extended Primitives Library to render the wheel. The wheel is partitioned into the following six regions of color transition:

1) red to yellow
2) yellow to green
3) green to cyan
4) cyan to blue
5) blue to magenta
6) magenta to red

Each region spans a 60-degree arc of the wheel.




```
color_wheel(t, r, g, b, i)
short t;
unsigned char r, g, b, i;
{
    long val;

    val = get_nearest_color(r, g, b, i);
    set_fcolor(val);
    fill_piearc(140, 110, 10, 10, t, 1);
}

main()
{
    short t;
    unsigned char r, g, b;

    set_config(0, !0);
    clear_screen(-1);
    for (t = 0, r = 255, g = b = 15; t < 60; t++, g += 4)
         color_wheel(t, r, g, b, g);          /* red to yellow */
    for ( ; t < 120; t++, r -= 4)
         color_wheel(t, r, g, b, r);          /* yellow to green */
    for ( ; t < 180; t++, b += 4)
         color_wheel(t, r, g, b, b);          /* green to cyan */
    for ( ; t < 240; t++, g -= 4)
         color_wheel(t, r, g, b, g);          /* cyan to blue */
    for ( ; t < 300; t++, r += 4)
         color_wheel(t, r, g, b, r);          /* blue to magenta */
    for ( ; t < 360; t++, b -= 4)
         color_wheel(t, r, g, b, b);          /* magenta to red */
}
```

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

typedef struct {
    PTR addr;
    unsigned short xext, yext;
} OFFSCREEN_AREA;

void get_offscreen_memory(num_blocks, offscreen)
short num_blocks;         /* number of off–screen buffers
*/
OFFSCREEN_AREA *offscreen;/* list of off–screen buffers */
```

Description

The *get_offscreen_memory* function returns a list of off-screen buffers located in the TMS340 graphics processor's display memory.

Argument *num_blocks* specifies the number of off-screen buffer areas to be listed. Argument *offscreen* is an array to contain the list of off-screen buffers. Each element of the offscreen array is a structure of type OFFSCREEN_AREA.

The fields of the OFFSCREEN_AREA structure are defined as follows:

addr	base address of off-screen buffer
xext	x extent (width in pixels) of off-screen buffer
yext	y extent (height in pixels) of off-screen buffer

An off-screen buffer is a two-dimensional array of pixels, the rows of which may not be contiguous in memory. The pixel size is the same as that of the screen, and each off-screen buffer has the same pitch as the screen. The pitch is the difference in memory addresses between two vertically adjacent pixels in the buffer.

If an off-screen buffer is used as the off-screen workspace (see the description of the *set_wksp and get_wksp* functions), this buffer is always the first buffer listed in the *offscreen* array.

Let N represent the number of off-screen buffers available in a particular graphics mode. If argument *num_blocks* is greater than N, only the first N elements of the offscreen array will be loaded with valid data. If argument *num_blocks* is less than N, only the first *num_blocks* elements of the *offscreen* array will be loaded with valid data. The number of off-screen areas available in the current mode is specified in the *num_offscrn_areas* field of the CONFIG structure returned by the *get_config* function.

After the display memory (usually video RAM) has been partitioned into one or more video pages (or frame buffers), some number of rectangular, noncontiguous blocks of display memory are typically left over. These blocks may not be suitable for general use (for example, for storing a program) but may be of use to some applications as temporary storage for graphical information such as the areas behind pull-down menus on the screen.

Example Use the *get_offscreen_memory* function to list the first (up to) 5 off-screen buffers available in the current graphics mode. Use the *text_out* function to print the x and y extents of each buffer on the screen.

```
#include <gsptypes.h>       /* OFFSCREEN_AREA, CONFIG, FONTINFO */
#define   MAXBUFS     5     /* max. number of buffers needed */

main()
{
    OFFSCREEN_AREA offscrn[MAXBUFS];
    CONFIG cfg;
    FONTINFO fntinf;
    short x, y, i, k, nbufs;
    char c[80];

    set_config(0, !0);
    clear_screen(-1);
    get_config(&cfg);
    get_fontinfo(-1, &fntinf);
    if ((nbufs = cfg.mode.num_offscrn_areas) > MAXBUFS)
       nbufs = MAXBUFS;
    get_offscreen_memory(nbufs, offscrn);
    if (!nbufs)
       text_out(10, 10, "No off-screen buffers available.");
    else
       for (i = 0, x = y = 10; i < nbufs; i++) {
           k = strlen(strcpy(c, "Buffer "));
           k += ltoa(i, &c[k]);
           k += strlen(strcpy(&c[k], ":  xext = "));
           k += ltoa(offscrn[i].xext, &c[k]);
           k += strlen(strcpy(&c[k], ", yext = "));
           ltoa(offscrn[i].yext, &c[k]);
           text_out(x, y, c);
           y += fntinf.charhigh;
       }
}
```

Syntax

```
typedef struct { unsigned char r, g, b, i; } PALET;

void get_palet(palet_size, palet)
short palet_size;   /* number of palette entries */
PALET *palet;       /* list of palette entries */
```

Description

The *get_palet* function copies multiple palette entries into an array.

Argument *palet_size* is the number of palette entries to load into the target array.

Argument *palet* is an array of type PALET. Each array element is a structure containing *r*, *g*, *b,* and *i* fields. Each field specifies an 8-bit red, green, blue, or gray-scale intensity value in the range 0 to 255, where 255 is the brightest intensity and 0 is the darkest. In the case of a graphics mode for a color display, the red, green, and blue component intensities are loaded into the *r*, *g,* and *b* fields, respectively, while the *i* field is set to 0. In the case of a gray-scale mode, the intensities are loaded into the i fields, and the *r*, *g,* and *b* fields are set to 0.

If argument *palet_size* specifies some number N that is less than the number of entries in the palette, only the first N palet entries are loaded into the array. If the number N of palette entries is less than the number specified in *palet_size*, only the first N elements of the array are modified. The number of palette entries in the current graphics mode is specified in the *palet_size* field of the CONFIG structure returned by the *get_config* function.

The 8-bit *r*, *g*, *b,* and *i* values retrieved for each palette entry represent the color components or gray-scale intensity actually output by the physical display device. For example, assume that the *r*, *g*, *b,* and *i* values of a particular palette entry are set by the *set_palet* or *set_palet_entry* functions to the following values: *r* = 0xFF, *g* = 0xFF, *b* = 0xFF, and *i* = 0. If the display hardware supports only 4 bits of red, green, and blue intensity per gun, the values read by a call to *get_palet* or *get_palet_entry* are: *r* = 0xF0, *g* = 0xF0, *b* = 0xF0, and *i* = 0.

Example Use the *get_palet* function to get the *r*, *g*, *b,* and *i* components of the first 8 colors in the default palette. Use the *text_out* function to print the values on the screen.

```
#include <gsptypes.h>  /* PALET, CONFIG and FONTINFO */
#define  MAXSIZE   8   /* max. number of LUT entries to print */

main()
{
    PALET lut[16];
    CONFIG cfg;
    FONTINFO fntinf;
    short k, n, size, x, y;
    char *s, c[80];

    set_config(0, !0);
    clear_screen(-1);
    get_config(&cfg);
    if ((size = cfg.mode.palet_size) > MAXSIZE)
         size = MAXSIZE;
    get_palet(size, lut);   /* get up to 8 palette entries */
    get_fontinfo(-1, &fntinf);
    x = y = 10;
    for (k = 0; k < size; k++, y += fntinf.charhigh) {
         n = strlen(strcpy(c, "color "));
         n += ltoa(k, &c[n]);
         n += strlen(strcpy(&c[n], ": r="));
         n += ltoa(lut[k].r, &c[n]);
         n += strlen(strcpy(&c[n], ", g="));
         n += ltoa(lut[k].g, &c[n]);
         n += strlen(strcpy(&c[n], ", b="));
         n += ltoa(lut[k].b, &c[n]);
         n += strlen(strcpy(&c[n], ", i="));
         n += ltoa(lut[k].i, &c[n]);
         text_out(x, y, c);
    }
}
```

Syntax

```
short get_palet_entry(index, r, g, b, i)
long index;              /* index to palette entry */
unsigned char *r, *g, *b; /*red, green and blue components*/
unsigned char *i;        /* gray-scale intensity */
```

Description

The *get_palet_entry* routine returns the red, green, blue, and gray-scale intensity components of a specified entry in the palette.

The palette entry is specified by argument *index*, which is an index into the color look-up table, or palette. If the palette contains N entries, valid indices are in the range 0 to N–1. The number of palette entries in the current graphics mode is specified in the *palet_size* field of the CONFIG structure returned by the *get_config* function.

Arguments *r*, *g*, *b,* and *i* are pointers to the red, green, blue, and gray-scale intensity values, respectively, that are retrieved by the function. Each intensity is represented as an 8-bit value in the range 0 to 255, where 255 is the brightest intensity and 0 is the darkest. In the case of a graphics mode for a color display, the red, green, and blue component intensities are loaded into the *r*, *g,* and *b* fields, respectively, while the *i* field is set to 0. In the case of a gray-scale mode, the intensity is loaded into the *i* field, and the *r*, *g,* and *b* fields are set to 0.

If argument *index* is in the valid range, the function returns a nonzero value, indicating that the components of the palette entry have been successfully retrieved. If argument *index* is invalid, the return value is 0, indicating that no palette entry corresponds to the specified index.

Example Use the *get_palet_entry* function to get the *r*, *g*, *b*, and *i* components of the first 8 colors in the default palette. Use the *text_out* function to print the values on the screen.

```
#include <gsptypes.h>  /* CONFIG and FONTINFO struct's */
#define  MAXSIZE   8   /* max. number of LUT entries to print */

main()
{
    CONFIG cfg;
    FONTINFO fntinf;
    long k, size;
    unsigned char r, g, b, i;
    short n, x, y;
    char *s, c[80];

    set_config(0, !0);
    clear_screen(-1);
    get_config(&cfg);
    if ((size = cfg.mode.palet_size) > MAXSIZE)
        size = MAXSIZE;
    get_fontinfo(-1, &fntinf);
    x = y = 10;
    for (k = 0; k < size; k++, y += fntinf.charhigh) {
        get_palet_entry(k, &r, &g, &b, &i);
        n = strlen(strcpy(c, "color "));
        n += ltoa(k, &c[n]);
        n += strlen(strcpy(&c[n], ": r="));
        n += ltoa(r, &c[n]);
        n += strlen(strcpy(&c[n], ", g="));
        n += ltoa(g, &c[n]);
        n += strlen(strcpy(&c[n], ", b="));
        n += ltoa(b, &c[n]);
        n += strlen(strcpy(&c[n], ", i="));
        n += ltoa(i, &c[n]);
        text_out(x, y, c);
    }
}
```

Syntax `unsigned long get_pmask()`

Description The *get_pmask* function returns the value of the plane mask. The size of the plane mask in bits is the same as the pixel size.

Given a pixel size of N bits, the plane mask is right-justified in the N LSBs of the return value and zero-extended. The screen pixel size in the current graphics mode is specified in the *disp_psize* field of the CONFIG structure returned by the *get_config* function.

The plane mask designates which bits within a pixel are protected against writes and affects all operations on pixels. During writes, the 1s in the plane mask designate bits in the destination pixel that are protected against modification, while the 0s in the plane mask designate bits that can be altered. During reads, the 1s in the plane mask designate bits in the source pixel that are read as 0s, while the 0s in the plane mask designate bits that can be read from the source pixel as is.

The plane mask is set to its default value of 0 during initialization of the drawing environment by the *set_config* function. The plane mask can be altered with a call to the *set_pmask* function.

The plane mask corresponds to the contents of the TMS340 graphics processor's PMASK register. The effect of the plane mask in conjunction with the pixel-processing operation and the transparency mode is described in the user's guides for the TMS34010 and TMS34020.

Example Use the *get_pmask* function to verify that the plane mask is initialized to 0 by a call to the *set_config* function. Use the *text_out* function to print the default plane mask value to the screen.

```
main()
{
    unsigned long pmask;
    short digit;
    char *s1, *s2;

    set_config(0, !0);
    clear_screen(–1);
    s2 = &s1[strlen(s1 = ”plane mask = 0x00000000”)];
    for (pmask = get_pmask(); pmask; pmask /= 16) {
         digit = pmask & 15;
         *––s2 = (digit < 10) ? (digit + ’0’) : (digit + ’A’ – 10);
    }
    text_out(10, 10, s1);
}
```

Syntax `unsigned short get_ppop()`

Description The *get_ppop* function returns the pixel-processing operation code. The 5-bit PPOP code determines the manner in which pixels are combined (Boolean or arithmetic operation) during drawing operations.

The PPOP code is right-justified in the 5 LSBs of the return value and zero-extended.

Legal PPOP codes are in the range 0 to 21. The source and destination pixel values are combined according to the selected Boolean or arithmetic operation, and the result is written back to the destination pixel. As shown in Table 6–1, Boolean operations are in the range 0 to 15, and arithmetic operations are in the range 16 to 21.

Table 6–1. Pixel-Processing Operations

PPOP Code	Description
0	replace destination with source
1	source AND destination
2	source AND NOT destination
3	set destination to all 0s
4	source OR NOT destination
5	source EQU destination
6	NOT destination
7	source NOR destination
8	source OR destination
9	destination (no change)
10	source XOR destination
11	NOT source AND destination
12	set destination to all 1s
13	NOT source or destination
14	source NAND destination
15	NOT source
16	source plus destination (with overflow)
17	source plus destination (with saturation)
18	destination minus source (with overflow)
19	destination minus source (with saturation)
20	MAX(source, destination)
21	MIN(source, destination)

The PPOP code is set to its default value of 0 (*replace* operation) during initialization of the drawing environment by the *set_config* function. The PPOP code can be altered with a call to the *set_ppop* function.

The pixel-processing operation code corresponds to the 5-bit PPOP field in the TMS340 graphics processor's CONTROL register. The effects of the 22 different codes are described in more detail in the user's guides for the TMS34010 and TMS34020.

Example Use the *get_ppop* function to verify that the pixel-processing operation code is initialized to 0 (replace destination with source) by a call to the *set_config* function. Use the *text_out* function to print the default PPOP code to the screen.

```
main()
{
    unsigned long ppop;
    char *s, c[80];

    set_config(0, !0);
    clear_screen(-1);
    ppop = get_ppop();
    strcpy(c, "PPOP code = ");
    ltoa(ppop, &c[12]);
    text_out(10, 10, c);
}
```

Syntax

```
void get_text_xy(x, y)
short *x, *y;                /* text x–y coordinates */
```

Description

The *get_text_xy* function retrieves the x–y coordinates at the current text drawing position. This is the position at which the next character (or string of characters) will be drawn if a subsequent call is made to the *text_outp* function. Both the *text_outp* and *text_out* functions automatically update the text position to be the right edge of the last string output to the screen.

Arguments *x* and *y* are pointers to variables of type *short*. The x and y coordinate values copied by the function into these variables correspond to the current text position on the screen, specified relative to the current drawing origin. The x coordinate corresponds to the left edge of the next string output by the *text_outp* function. The y coordinate corresponds either to the top of the string, or to the base line, depending on the state of the text alignment attribute (see the description of the *set_textattr* function).

Example

Use the *get_text_xy* function to print four short lines of text in a stairstep pattern on the screen. Each time the *text_outp* function outputs the string "step" to the screen, the *get_text_xy* function is called next to obtain the current text position. The y coordinate of this position is incremented by a call to the *set_text_xy* function, and the next call to the *text_outp* function prints the string at the new position.


```
#include  <gsptypes.h>

main()
{
    short x, y, i;
    FONTINFO fntinf;

    set_config(0, 1);
    clear_screen(-1);
    get_fontinfo(-1, &fntinf);
    x = y = 0;
    for (i = 4; i; i--) {
         set_text_xy(x, y);
         text_outp("step");
         get_text_xy(&x, &y);
         y += fntinf.charhigh;
    }
}
```

Syntax `short get_transp()`

Description The *get_transp* function returns a value indicating whether transparency is enabled. A nonzero value is returned if transparency is enabled; 0 is returned if transparency is disabled.

Transparency is an attribute that affects drawing operations. 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. To avoid modifying destination pixels in the rectangular region surrounding each character shape, transparency can be enabled before the *text_out* or *text_outp* function is called.

The transparency attribute value returned by the function corresponds to the T bit in the TMS340 graphics processor's CONTROL register. The effect of transparency in conjunction with the pixel-processing operation and the plane mask is described in the user's guides for the TMS34010 and TMS34020.

Example Use the *get_transp* function to verify that transparency is disabled by a call to the *set_config* function. Use the *text_out* function to print the value returned by the *get_transp* function to the screen.

```
main()
{
    unsigned long transp;
    char *s, c[80];

    set_config(0, !0);
    clear_screen(–1);
    transp = get_transp();
    strcpy(c, ”transparency = ”);
    ltoa(transp, &c[15]);
    text_out(10, 10, c);
}
```

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

PTR get_vector(trapnum)
short trapnum;               /* trap number */
```

Description

The *get_vector* function returns one of the TMS340 graphics processor's trap vectors. This function provides a portable means of obtaining the entry point to a trap service routine, regardless of whether the actual trap vector is located in RAM or ROM.

Argument *trapnum* specifies a trap number in the range –32768 to 32767 for a TMS34020, and 0 to 31 for a TMS34010.

The value returned by the function is the 32-bit address contained in the designated trap vector.

Example

Use the *get_vector* function to retrieve whatever address happens to be in trap vector 0. Use the *text_out* function to print the value returned by the *get_vector* function to the screen as an 8-digit hexadecimal number.

```
main()
{
    unsigned long vector;
    short digit;
    char *s1, *s2;

    set_config(0, !0);
    clear_screen(–1);
    s2 = &s1[strlen(s1 = ”trap 0 vector = 0x00000000”)];
    for (vector = get_vector(0); vector; vector /= 16) {
         digit = vector & 15;
         *––s2 = (digit < 10) ? (digit + ’0’) : (digit + ’A’ – 10);
    }
    text_out(10, 10, s1);
}
```

Syntax `short get_windowing()`

Description The *get_windowing* function returns a 2-bit code designating the current window-checking mode. This function is provided for the sake of backward compatibility with early versions of TIGA.

The four windowing modes are:

00_2	Window clipping disabled
01_2	Interrupt request on write to pixel inside window
10_2	Interrupt request on write to pixel outside window
11_2	Clip to window

The library's drawing functions assume that the TMS340 graphics processor is configured in windowing mode 3. Changing the windowing mode from this default may result in undefined behavior.

The 2-bit code for the window-clipping mode corresponds to the W field in the TMS340 graphics processor's CONTROL register. The effects of the W field on window-clipping operations are described in the user's guides for the TMS34010 and TMS34020.

Immediately following initialization of the drawing environment by the *set_config* function, the system is configured in windowing mode 3, which is the default.

Syntax

```
typedef unsigned long PTR;/* 32–bit GSP memory address */

short get_wksp(addr, pitch)
PTR *addr;              /*  pointer  to  workspace  address
*/
PTR *pitch;             /*  pointer  to  workspace  pitch
*/
```

Description

The *get_wksp* function retrieves the parameters that define the current off-screen workspace. None of the current TIGA core or extended primitives use this workspace; it is provided to support future graphics extensions that require storage for edge flags or region-of-interest masks. Not all display configurations may have sufficient memory to support an off-screen workspace.

Argument *addr* is the base address of the off-screen workspace. Argument *pitch* is the difference in memory addresses of two adjacent rows in the off-screen workspace.

A nonzero value is returned by the function if a valid off-screen workspace is currently allocated. A value of 0 is returned if no off-screen workspace is allocated; in this case, the workspace address and pitch are not retrieved by the function.

The off-screen workspace is a 1-bit-per-pixel bit map of the same width and height as the screen. If the display hardware provides sufficient off-screen memory, the workspace can be allocated statically at link time. By convention, the workspace pitch retrieved by the *get_wksp* function is nonzero when a workspace is allocated; the pitch can be checked following initialization to determine whether a workspace is statically allocated. The workspace can be allocated dynamically by calling the *set_wksp* function with the address of a valid workspace in memory and a nonzero pitch; it can be deallocated by calling *set_wksp* with a pitch of *0*.

Syntax

```
void gsp2gsp(src, dst, length)
char *src, *dst;       /* source and destination arrays */
unsigned long length;  /* number of bytes to copy */
```

Description The *gsp2gsp* function copies the specified number of bytes from one region of the TMS340 graphics processor's memory to another.

Argument *src* is a pointer to the source array, and argument *dst* is a pointer to the destination array. Argument *length* is the number of contiguous 8-bit bytes to be transferred from the source to the destination.

If the source and destination arrays overlap, the function is intelligent enough to adjust the order in which the bytes are transferred so that no source byte is overwritten before it has been copied to the destination.

Unlike the standard character string function *strncpy*, the *gsp2gsp* function does not restrict the alignment of the source and destination addresses to even byte boundaries in memory. Arguments *src* and *dst* may point to any bit boundaries in memory.

Example Use the *gsp2gsp* function to copy three characters from a source string to a destination string. The source and destination strings overlap. Use the *text_out* function to print the resulting string, "ABCBC", to the screen.

```
main()
{
    static char c[80] = "AAABC";

    set_config(0, !0);
    clear_screen(-1);
    gsp2gsp(&c[2], c, 3);
    text_out(10, 10, c);
}
```

Syntax `void init_palet()`

Description The *init_palet* function initializes the first 16 entries of the palette to the EGA default colors:

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

If the palette contains more than 16 entries, the function replicates the 16 colors through the remainder of the palette. At 2 bits/pixel, palette indices 0–3 are assigned the default colors black, green, red, and white. At 1 bit/pixel, palette indices 0 and 1 are assigned the default colors black and white. If the palette is fixed, the function has no effect.

The palette is also initialized to the default colors above during initialization of the drawing environment by the *set_config* function.

Example Use the *init_palet* function to restore the default colors.

```
main()
{
    short i;

    set_config(0, !0);
    clear_screen(-1);
    for (i = 0; i < 16; i++)       /* overwrite default colors */
        set_palet_entry(i, i, i, i, i);
    init_palet();                  /* restore default colors */
}
```

Syntax `void init_text()`

Description The *init_text* function removes all installed fonts from the font table and selects the system font (font 0) for use in subsequent text operations. It also resets all text drawing attributes to their default states.

The *set_config* function also initializes the font table and text attributes as part of its initialization of the drawing environment.

Example Use the *init_text* function to discard all installed fonts from the font table and select the default font. The *install_font* and *select_font* functions from the Extended Primitives Library are used to install and select a proportionally spaced font. The TI Roman font size 16 must be linked with the program.


```
#include <gsptypes.h>    /* defines FONT and FONTINFO structures */

extern FONT ti_rom16;    /* font name */

main()
{
    FONTINFO fontinfo;
    short x, y, index;

    set_config(0, !0);
    clear_screen(-1);
    x = y = 10;
    index = install_font(&ti_rom16);
    select_font(index);
    get_fontinfo(-1, &fontinfo);
    text_out(x, y, "Hello world.");  /* print in TI Roman 16 */
    y += fontinfo.charhigh;
    init_text();
    text_out(x, y, "Hello world.");  /* print in system font */
}
```

Syntax

```
short lmo(n)
unsigned long n;              /* 32–bit integer */
```

Description

The *lmo* function calculates the bit number of the leftmost 1 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

Use the *lmo* function to return the bit number of the leftmost 1 in the integer value 1234.

```
#include <gsptypes.h>     /* defines FONTINFO structure */

static FONTINFO fontinfo;

main()
{
    long x, n;
    short m;
    char c[80];

    set_config(0, !0);
    clear_screen(–1);
    get_fontinfo(–1, &fontinfo);
    x = 1234;
    n = lmo(x);
    strcpy(c, "The leftmost 1 in ");
    m = strlen(c);
    ltoa(x, &c[m]);
    text_out(10, 10, c);
    strcpy(c, "is bit number ");
    m = strlen(c);
    ltoa(n, &c[m]);
    text_out(10, 10+fontinfo.charhigh, c);
}
```

Syntax `short page_busy()`

Description The *page_busy* function returns a nonzero value as long as a previously requested video page flip has not yet occurred. This function is used in conjunction with the *page_flip* function to achieve flickerless, double-buffered animation.

Before the *page_busy* function is called, the *page_flip* function is called to request the page flip, which is scheduled to occur when the bottom line of the screen has been scanned on the monitor. The *page_flip* function returns immediately without waiting for the requested page flip to be completed, and the *page_busy* function is used thereafter to monitor the status of the request. Between the call to the *page_flip* function and the time the page flip actually occurs, the *page_busy* function returns a nonzero value. After the page flip has occurred, the *page_busy* returns a value of 0 (until the next time *page_flip* is called).

Double buffering is a technique used to achieve flickerless animation in graphics modes supporting more than one video page. The TMS340 graphics processor alternately draws to one page (or frame buffer) while the other page is displayed on the screen of the monitor. When the processor has finished drawing, the new page is ready to be displayed on the screen in place of the old. The actual flipping (or switching) of display pages is delayed until the vertical blanking interval, however, to avoid causing the image on the screen to flicker.

The rationale for providing separate *page_flip* and *page_busy* functions is to make the time between between a page-flip request and the actual completion of the page flip available to the application program for performing background calculations. For example, the main loop of a 3D animation program can be structured as follows:

```
for (disp = 1, draw = 0; ; disp ^= 1, draw ^= 1) {
    page_flip(disp, draw);
    < Perform 3D background calculations. >
    while (page_busy())
        ;
    < Draw updated 3D scene. >
}
```

If the *page_flip* function is used alone without the *page_busy* function, the programmer risks drawing to a page that is still being displayed on the screen.

Example Use the *page_busy* function to smoothly animate an object rotating in a circle. The best effect is achieved in a graphics mode that provides double buffering (more than one video page). If the mode supports only a single page, the program will still run correctly, but the display may flicker.

```
#define  RADIUS  60    /* radius of circle of rotation */
#define  N        4    /* angular increment = 1>>N radians */

main()
{
    short disp, draw;
    long x, y;

    set_config(0, !0);
    x = RADIUS << 16;
    y = 0;
    for (disp = 0, draw = 1; ; disp ^= 1, draw ^= 1) {
        page_flip(disp, draw);
        x -= y >> N;
        y += x >> N;
        while (page_busy())
            ;
        clear_page(-1);
        text_out((x>>16)+RADIUS, (y>>16)+RADIUS, "*");
    }
}
```

Syntax

```
void page_flip(disp, draw)
short disp, draw;           /* display and drawing pages */
```

Description

The *page_flip* function is used to switch between alternate video pages. This function is used in conjunction with the *page_busy* function to achieve flickerless, double-buffered animation.

Argument *disp* is a nonnegative value indicating the number of the video page to be displayed—that is, output to the monitor screen. Argument *draw* is a nonnegative value indicating the number of the video page to be drawn to; this page is the target of all graphics output directed to the screen. All graphics modes support at least one video page, page number 0. In graphics modes supporting more than one page, the pages are numbered 0, 1, and so on.

Valid values for arguments *disp* and *draw* are restricted to video page numbers supported by the current graphics mode. If either argument is invalid, the function behaves as if both arguments are 0; that is, page 0 is selected as both the display page and drawing page. This behavior permits programs written for double-buffered modes to be run in single-buffered modes. Although the single-buffered display may flicker, the program will execute at nearly the same frame rate as in the double-buffered mode.

The number of pages in a particular graphics mode is specified in the *num_pages* field of the CONFIG structure returned by the *get_config* function. If the *num_pages* field contains some value N, the N pages are numbered 0 through N–1.

The *page_flip* function requests that a page flip be performed but returns immediately without waiting for the requested page flip to be completed. Upon return from the function, all subsequent screen drawing operations are directed toward the page specified by argument *draw*. The monitor display, however, is not updated to the page specified by argument *disp* until the start of the next vertical blanking interval (which occurs when the monitor finishes scanning the last line on the screen). Between the call to the *page_flip* function and the time the page flip actually occurs, the *page_busy* function returns a nonzero value. This is true regardless of whether the *disp* and *draw* arguments are the same or whether the new display page is the same as the old display page. After the page flip has occurred, the *page_busy* returns a value of 0 (until the next time *page_flip* is called).

Double buffering is a technique used to achieve flickerless animation in graphics modes supporting more than one video page. The TMS340 graphics processor alternately draws to one page (or frame buffer) while the other page is displayed on the screen of the monitor. When the processor has finished drawing, the new page is ready to be displayed on the screen in place of the old. The actual flipping (or switching) of display pages is

delayed until the vertical blanking interval, however, to avoid causing the image on the screen to flicker.

Example Use the *page_flip* function to smoothly animate two moving rectangles. Use the *fill_rect* function from the Extended Primitives Library to draw the rectangles. The selected graphics mode is assumed to be double-buffered—that is, to support more than one video page. If the mode supports only a single page, the program will still run correctly, but the display may flicker.

```
#define  RADIUS   60     /* radius of circle of rotation */
#define  XOR      10     /* pixel processing operation code */
#define  N         5     /* angular increment = 1>>N radians */

main()
{
    short disp, draw;
    long x, y;

    set_config(1, !0);
    set_ppop(XOR);
    x = RADIUS << 16;
    y = 0;
    for (disp = 0, draw = 1; ; disp ^= 1, draw ^= 1) {
        page_flip(disp, draw);
        x -= y >> N;
        y += x >> N;
        while (page_busy())
            ;
        clear_screen(-1);
        fill_rect(2*RADIUS, RADIUS/4, 10, RADIUS+(y>>16));
        fill_rect(RADIUS/4, 2*RADIUS, RADIUS+(x>>16), 10);
    }
}
```

Syntax

```
unsigned long peek_breg(breg)
short breg;                  /* B–file register number */
```

Description The *peek_breg* function returns the contents of a B-file register. Argument *breg* is a number in the range 0 to 15 that designates a register in the TMS340 graphics processor's B file. Argument values 0 through 14 correspond to registers B0 through B14. An argument value of 15 designates the SP (system stack pointer). The function ignores all but the 4 LSBs of argument *breg*. The return value is 32 bits.

Example Use the *peek_breg* function to read the contents of register B9, also referred to as the COLOR1 register. Register B9 contains the foreground color in pixel-replicated form. For example, at 4 bits per pixel, a foreground pixel value of 7 is replicated 8 times to form the 32-bit value 0x77777777.

```
main()
{
    unsigned long n;
    short digit;
    char *s1, *s2;

    set_config(0, !0);
    clear_screen(–1);
    s2 = &s1[strlen(s1 = ”COLOR1 = 0x00000000”)];
    for (n = peek_breg(9); n; n /= 16) {
        digit = n & 15;
        *––s2 = (digit < 10) ? (digit + ’0’) : (digit + ’A’ – 10);
    }
    text_out(10, 10, s1);
}
```

poke_breg *Poke Value into B-File Register*

Syntax

```
void poke_breg(breg, val)
short breg;                 /* B-file register number   */
unsigned long val;          /* 32-bit register contents */
```

Description

The *poke_breg* function loads a 32-bit value into a B-file register. Argument *breg* is a number in the range 0 to 15 that designates a register in the TMS340 graphics processor's B file. Argument values 0 through 14 correspond to registers B0 through B14. An argument value of 15 designates the SP (system stack pointer). The function ignores all but the 4 LSBs of argument *breg*. Argument *val* is a 32-bit value that is loaded into the designated register.

Example

Use the *poke_breg* function to load the value 0 into the TMS340 graphics processor's register B6, also referred to as the WEND register. Use the *fill_rect* function from the Extended Primitives Library to draw a filled rectangle that is specified to be larger than the clipping window. Register B6 contains the upper x and y limits for the clipping window. Following the *poke_breg* call, the clipping window contains only the single pixel at (0, 0). Obviously, the *set_clip_rect* function provides a safer and more portable means to adjust the clipping window than the one used in this example.

```
main()
{
    set_config(0, !0);
    clear_screen(-1);
    poke_breg(6, 0);
    fill_rect(100, 100, 0, 0);
}
```

Syntax

```
short rmo(n)
unsigned long n;              /* 32–bit integer */
```

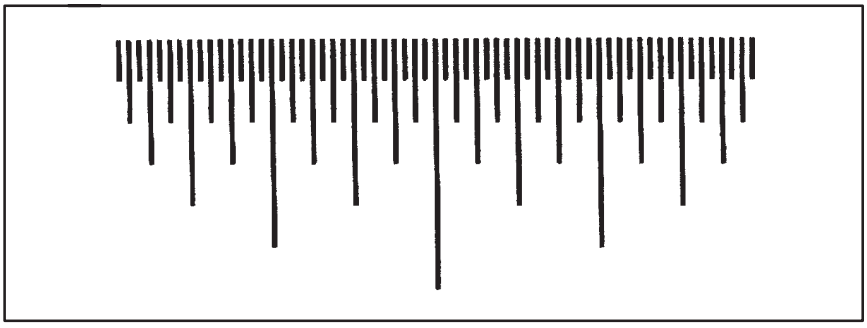
Description

The *rmo* function calculates the bit number of the rightmost 1 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

Use the *rmo* function to calculate the bit number of the rightmost 1 for each integer in the range 1 to 127. Represent the result graphically as a series of 127 adjacent vertical lines. Use the *fill_rect* function from the Extended Primitives Library to draw the vertical lines.



```
main()
{
    unsigned long i;
    short n;

    set_config(0, !0);
    clear_screen(–1);
    for (i = 1; i < 128; i++) {
        n = rmo(i);
        fill_rect(1, 8*n, 10+i, 10);
    }
}
```

Syntax

```
void set_bcolor(color)
unsigned long color;          /* background pixel value */
```

Description

The *set_bcolor* function sets the background color for subsequent drawing operations.

Argument *color* specifies the pixel value to be used to draw background pixels. Given a pixel size of N bits, the pixel value is contained in the N LSBs of the argument; the higher-order bits are ignored.

The function creates a 32-bit replicated pixel value and loads the result into the TMS340 graphics processor's register B8, also referred to as the COLOR0 register. For example, given a pixel size of 4 bits and a pixel value of 6, the replicated pixel value is 0x66666666.

Example

Use the *set_bcolor* function to swap the foreground and background colors.

```
main()
{
    unsigned long fcolor, bcolor;

    set_config(0, !0);
    clear_screen(–1);
    get_colors(&fcolor, &bcolor);
    set_fcolor(bcolor);
    set_bcolor(fcolor);
    text_out(10, 10, "Swap COLOR0 and COLOR1.");
}
```

Syntax

```
void set_clip_rect(w, h, xleft, ytop)
unsigned short w, h;    /* width, height of clip window */
short xleft, ytop;      /* coordinates at top left corner */
```

Description The *set_clip_rect* function specifies the position and size of the rectangular clipping window for subsequent drawing operations.

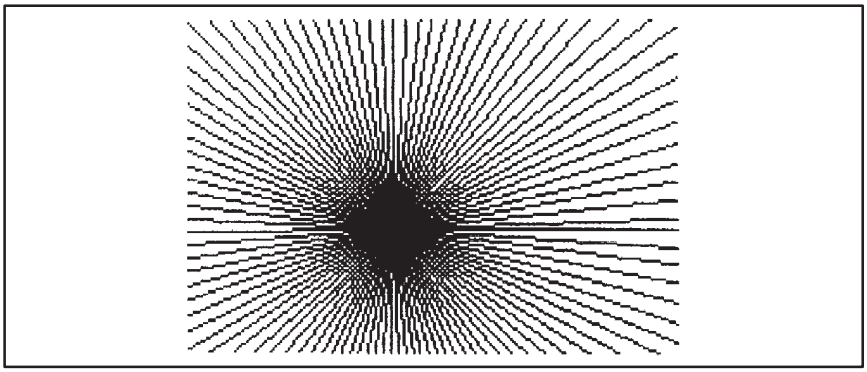
Arguments *w* and *h* specify the width and height of the clipping window in pixels. Arguments *xleft* and *ytop* specify the x and y coordinates at the top-left corner of the window, relative to the drawing origin in effect at the time *set_clip_rect* is called.

If the specified clipping window extends beyond the screen boundaries, the effective window is limited by the function to that portion of the specified window that actually lies on the screen.

A call to the *set_draw_origin* function (in the Extended Primitives Library) has no effect on the position of the clipping window until the *set_clip_rect* function is called. During initialization of the drawing environment by the *set_config* function, the clipping window is set to its default limits, which are the entire screen.

The function updates the contents of the TMS340 graphics processor's registers B5 and B6, which are also referred to as the WSTART (window start) and WEND (window end) registers. These registers are described in the user's guides for the TMS34010 and TMS34020.

Example Use the *set_clip_rect* function to specify a clipping window of width 192 pixels and height 128 pixels. Use the *draw_line* function to draw a series of concentric rays that emanate from a point within the window, but which extend beyond the window. The rays are automatically clipped to the limits of the window. Note that the call to *set_clip_rect* follows the call to the *set_draw_origin* function, and that the x–y coordinates (–80, –80) passed as arguments to *set_clip_rect* are specified relative to the drawing origin at (88, 88).



```
main()
{
    int i;
    long x, y;

    set_config(0, !0);
    clear_screen(-1);
    set_draw_origin(88, 88);
    set_clip_rect(192, 128, -80, -80);
    x = 160 << 16;
    y = 0;
    for (i = 0; i <= 100; i++) {
        draw_line(0, 0, x>>16, y>>16);
        x -= y >> 4;
        y += x >> 4;
    }
}
```

Syntax

```
void set_colors(fcolor, bcolor)
unsigned long fcolor;        /* foreground pixel value */
unsigned long bcolor;        /* background pixel value */
```

Description The *set_colors* function specifies the foreground and background colors to be used in subsequent drawing operations.

Arguments *fcolor* and *bcolor* contain the pixel values used to draw the foreground and background colors, respectively. Given a pixel size of N bits, the pixel value is contained in the N LSBs of each argument; the higher-order bits are ignored.

The function creates 32-bit replicated pixel values and loads the results into the TMS340 graphics processor's registers B8 and B9, also referred to as the COLOR0 and COLOR1 registers. For example, given a pixel size of 4 bits and a pixel value of 3, the replicated pixel value is 0x33333333.

Example Use the *set_colors* function to swap the default foreground and background colors. Use the *text_out* function to print a string of text with the colors swapped.

```
main()
{
    long white, black;

    set_config(0, !0);
    clear_screen(–1);
    get_colors(&white, &black);
    set_colors(black, white);
    text_out(8, 8, ”Black text on white background.”);
}
```

Syntax

```
short set_config(graphics_mode, init_draw)
short graphics_mode;  /* graphics mode */
short init_draw;      /* initialize drawing environment */
```

Description

The *set_config* function configures the display system in the specified graphics mode. Both the display hardware and graphics software environment are initialized. With few exceptions, *set_config* should be called before any of the other functions in the graphics library are called.

Argument *graphics_mode* specifies the graphics mode. All display systems provide at least one graphics mode, mode 0. In display systems supporting multiple modes, the modes are numbered 0, 1, and so on.

Argument *init_draw* specifies whether the function initializes the drawing environment to its default state. If *init_draw* is nonzero, the environment is initialized; otherwise, the drawing environment remains unaltered.

The value returned by the function is nonzero if argument *graphics_mode* corresponds to a valid graphics mode—that is, a mode supported by the display system. If the specified mode number is invalid, the function performs no operation and returns a value of 0.

The number of modes available for a particular hardware configuration is specified in the *num_modes* field of the CONFIG structure returned by the *get_config* function. The modes are numbered 0 through *num_modes* – 1.

Following a call to *set_config*, the display system remains in the specified graphics mode until a subsequent call to *set_config* is made. Associated with each mode is a particular display resolution, pixel size, and so on.

The *set_config* function configures the following system parameters:

- ❏ horizontal and vertical video timing
- ❏ video-RAM screen-refresh cycles
- ❏ screen pixel size in bits
- ❏ screen dimensions (width and height in pixels)
- ❏ location in memory of one or more video pages (or frame buffers)
- ❏ default clipping window (entire screen)
- ❏ default color palette (See description of *init_palet* function.)
- ❏ default display and drawing pages (page 0 for both)
- ❏ default off-screen workspace (which may be *null*)

If a nonzero value is specified for argument *init_draw*, the parameters of the drawing environment are initialized as follows:

- ❏ Pixel transparency is disabled.
- ❏ The pixel-processing operation code is set to its default value of 0 (the code for the *replace* operation).
- ❏ The plane mask is set to its default value of 0, which enables all bit planes.

- The foreground color is set to light gray and the background color to black.
- The screen is designated as both the source bit map and destination bit map.
- The drawing origin is set to screen coordinates (0, 0), which correspond to the pixel at the top left corner of the screen.
- The pen width and height are both set to 1.
- The current area-fill pattern is set to its default state, which is to fill with solid foreground color.
- The current line-style pattern is set to its default value, which is all 1s.
- All installed fonts are removed, and font 0, the permanently installed system font, is selected.
- The text x-y position coordinates are set to (0,0).
- The text attributes are set to their initial states:
  - alignment = 0 (top left)
  - additional intercharacter spacing = 0
  - intercharacter gaps = 0 (leave gaps)

Example Use the *set_config* function to sequence the display through all available graphics modes. Use the *draw_rect* function to draw a box around the visible screen area, and use the *text_out* function to print the mode number and screen width and height to the screen. Use the *wait_scan* function to insert a delay of 120 frames between mode switches.

```
#include <gsptypes.h>      /* MODEINFO, CONFIG and FONTINFO */
#define  NFRAMES    120      /* delay in frames between modes */

main()
{
    CONFIG cfg;
    char c[80];
    short mode, i, w, h;

    for (;;)
         for (mode = 0; set_config(mode, !0); mode++) {
             clear_screen(-1);
             get_config(&cfg);
             w = cfg.mode.disp_hres;
             h = cfg.mode.disp_vres;
             draw_rect(w-1, h-1, 0, 0);
             i = strlen(strcpy(c, "graphics mode "));
             i += ltoa(mode, &c[i]);
             strcpy(&c[i], "...");
             i = strlen(c);
             i += ltoa(w, &c[i]);
             strcpy(&c[i], "-by-");
             i = strlen(c);
             ltoa(h, &c[i]);
             text_out(10, 10, c);
             for (i = NFRAMES; i; i--) /* delay loop */
                    wait_scan(h);
         }
}
```

set_fcolor *Set Foreground Color*

Syntax

```
void set_fcolor(color)
unsigned long color;          /* foreground pixel value */
```

Description The *set_fcolor* function sets the foreground color for subsequent drawing operations.

Argument *color* specifies the pixel value to be used to draw foreground pixels. Given a pixel size of N bits, the pixel value is contained in the N LSBs of the argument; the higher-order bits are ignored.

The function creates a 32-bit replicated pixel value and loads the result into the TMS340 graphics processor's register B9, also referred to as the COLOR1 register. For example, given a pixel size of 8 bits and a pixel value of 5, the replicated pixel value is 0x05050505.

Example Use the *set_fcolor* function to swap the foreground and background colors.

```
main()
{
    unsigned long fcolor, bcolor;

    set_config(0, !0);
    clear_screen(–1);
    get_colors(&fcolor, &bcolor);
    set_fcolor(bcolor);
    set_bcolor(fcolor);
    text_out(10, 10, ”Swap COLOR0 and COLOR1.”);
}
```

Syntax

```
typedef struct { unsigned char r, g, b, i; } PALET;

void set_palet(count, index, palet)
long count;               /* number of palette entries */
long index;               /* index to starting entry */
PALET *palet;             /* list of palette data */
```

Description The *set_palet* function loads multiple palette entries from a specified list of colors.

Argument *count* specifies the number of contiguous palette entries to be loaded. Argument *index* designates the palette entry at which loading is to begin. Argument *palet* is an array containing the colors to be loaded into the palette. The *palet* array must contain at least *count* elements. The palette entry identified by *index* is loaded from *palet* [0], and so on.

Argument *palet* is an array of type PALET. Each array element is a structure containining *r*, *g*, *b,* and *i* fields. Each field specifies an 8-bit red, green, blue, or gray-scale intensity value in the range 0 to 255, where 255 is the brightest intensity and 0 is the darkest. In the case of a graphics mode for a color display, the *r*, *g*, and *b* fields from each array element are loaded into the red, green, and blue component intensities for the corresponding palette entry; the *i* field from the element is ignored, and the gray-scale intensity component for the palette entry is set to 0. In the case of a gray-scale mode, the *i* field from each array element is loaded into the gray-scale intensity value for the corresponding palette entry; the *r, g,* and *b* fields from the element are ignored, and the red, green, and blue intensities for the palette entry are set to 0.

The range of palette entries to be loaded is checked by the function to ensure that it does not overflow the palette. If the starting index plus the number of entries (*count*) is greater than the palette size, the function decreases the *count* value by the appropriate amount.

The entire palette may be loaded at once by specifying a *count* equal to the number of palette entries, and an *index* of 0. The number of palette entries in the current graphics mode is specified in the *palet_size* field of the CONFIG structure returned by the *get_config* function.

The 8-bit *r*, *g*, *b,* and *i* values contained in the *palet* array are modified by the function to represent the color components or gray-scale intensity actually output by the physical display device. For example, assume that the *r*, *g*, *b,* and *i* values of a particular array element are specified as follows: *r* = 0xFF, *g* = 0xFF, *b* = 0xFF, and *i* = 0. If the display hardware supports only 4 bits of red, green, and blue intensity per gun, the values actually loaded into the palette by the *set_palet* function are: *r* = 0xF0, *g* = 0xF0, *b* = 0xF0, and *i* = 0.

In systems that store the palette data in display memory (such as those using the TMS34070 color palette), this function updates every palette area in the frame buffer. If the system contains multiple display pages, the function updates the palette area for every page.

Example Use the *set_palet* function to load a gray-scale palette into the first 16 color palette entries. Use the *fill_rect* function from the Extended Primitives Library to fill a series of rectangles in intensities increasing from left to right. Note that this example requires a color palette with a capacity of at least 16 entries.

```
#include  <gsptypes.h>   /* defines PALET struct */

main()
{
    int n;
    PALET p[16];

    set_config(0, !0);
    clear_screen(–1);
    for (n = 0; n < 15; n++)
         p[n].r = p[n].g = p[n].b = p[n].i = 16*n;
    set_palet(16, 0, p);
    for (n = 0; n < 15; n++) {
         set_fcolor(n);
         fill_rect(12, 80, 8+12*n, 8);
    }
}
```

Syntax

```
short set_palet_entry(index, r, g, b, i)
long index;             /* index to palette entry */
unsigned char r, g, b;  /* red, green and blue components */
unsigned char i;        /* gray–scale intensity */
```

Description

The *set_palet_entry* function updates a single entry in the color palette.

Argument *index* identifies the palette entry to be updated. Arguments *r*, *g*, *b,* and *i* specify 8-bit red, green, blue, and gray-scale intensity values in the range 0 to 255, where 255 is the brightest intensity and 0 is the darkest. If the current graphics mode supports a color display, arguments *r*, *g,* and *b* are the red, green, and blue component intensities. In the case of a gray-scale display, argument *i* is the gray-scale intensity.

If the palette contains N entries, the valid range of argument *index* is 0 through N–1. The number of palette entries in the current graphics mode is specified in the *palet_size* field of the CONFIG structure returned by the *get_config* function.

If argument *index* specifies an invalid value, the function aborts (returns immediately) and returns a value of 0; otherwise, a nonzero value is returned.

In systems that store the palette data in display memory (such as those using the TMS34070 color palette), this function updates every palette area in the frame buffer. If the system contains multiple display pages, the function updates the palette area for every page.

Example

Use the *set_palet_entry* function to load a gray-scale palette into the first 16 color palette entries. Use the *fill_rect* function from the Extended Primitives Library to fill a series of rectangles in intensities increasing from left to right. Note that this example requires a color palette with a capacity of at least 16 entries.

```
main()
{
    int n;

    set_config(0, !0);
    clear_screen(–1);
    for (n = 0; n < 15; n++)
         set_palet_entry(n, 16*n, 16*n, 16*n, 16*n);
    for (n = 0; n < 15; n++) {
         set_fcolor(n);
         fill_rect(12, 80, 8+12*n, 8);
    }
}
```

Syntax

```
void set_pmask(pmask)
unsigned long pmask;          /* plane mask */
```

Description

The *get_pmask* function sets the plane mask to the specified value. The size of the plane mask in bits is the same as the pixel size.

Argument *pmask* contains the plane mask. Given a pixel size of N bits, the plane mask is right-justified in the N LSBs of the argument; the higher-order bits are ignored by the function.

The plane mask designates which bits within a pixel are protected against writes and affects all operations on pixels. During writes, the 1s in the plane mask designate bits in the destination pixel that are protected against modification, while the 0s in the plane mask designate bits that can be altered. During reads, the 1s in the plane mask designate bits in the source pixel that are read as 0s, while the 0s in the plane mask designate bits that can be read from the source pixel as is.

The plane mask is set to its default value of *0* during initialization of the drawing environment by the *set_config* function. The plane mask can be altered with a call to the *set_pmask* function.

The plane mask corresponds to the contents of the TMS340 graphics processor's PMASK register. The effect of the plane mask in conjunction with the pixel-processing operation and the transparency mode is described in the user's guides for the TMS34010 and TMS34020.

Example Use the *set_pmask* function to demonstrate the effects of enabling and disabling particular bit planes. For each bit plane, print a line of text with *all* but the one plane enabled; print another line of text with *only* the one plane enabled. This example assumes that the display has at least 4 bit planes – that is, a pixel size of at least 4 bits.

```
#include  <gsptypes.h>          /* defines CONFIG and FONTINFO */
#define   MINPSIZE    4         /* minimum pixel size */

main()
{
    CONFIG cfg;
    FONTINFO fntinf;
    unsigned long pmask;
    short x, y, n;
    char c[80];

    set_config(0, !0);
    clear_screen(–1);
    get_config(&cfg);
    get_fontinfo(–1, &fntinf);
    x = y = 10;
    for (pmask = 1; pmask != 1<<MINPSIZE; pmask <<= 1) {

        /* Enable all planes except one. */
        set_pmask(pmask);
        n = strlen(strcpy(c, ”all planes enabled except ”));
        ltoa(lmo(pmask), &c[n]);
        text_out(x, y, c);
        y += fntinf.charhigh;

        /* Disable all planes except one. */
        set_pmask(~pmask);
        n = strlen(strcpy(c, ”all planes disabled except ”));
        ltoa(lmo(pmask), &c[n]);
        text_out(x, y, c);
        y += fntinf.charhigh;
    }
}
```

Syntax

```
void set_ppop(ppop)
short ppop;          /* pixel processing operation code */
```

Description

The *set_ppop* function specifies the pixel-processing operation to be used for subsequent drawing operations. The specified Boolean or arithmetic operation determines the manner in which source and destination pixel values are combined during drawing operations.

Argument *ppop* is a pixel-processing operation code in the range 0 to 21. The PPOP code is right-justified in the 5 LSBs of the argument; the higher-order bits are ignored by the function.

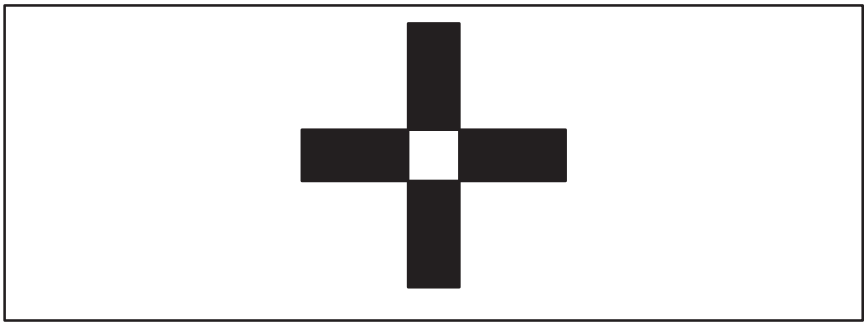
Legal PPOP codes are in the range 0 to 21. The source and destination pixel values are combined according to the selected Boolean or arithmetic operation, and the result is written back to the destination pixel. As shown in Table 6–2, Boolean operations are in the range 0 to 15, and arithmetic operations are in the range 16 to 21.

Table 6–2. Pixel-Processing Operations

PPOP Code	Description
0	replace destination with source
1	source AND destination
2	source AND NOT destination
3	set destination to all 0s
4	source OR NOT destination
5	source EQU destination
6	NOT destination
7	source NOR destination
8	source OR destination
9	destination (no change)
10	source XOR destination
11	NOT source AND destination
12	set destination to all 1s
13	NOT source or destination
14	source NAND destination
15	NOT source
16	source plus destination (with overflow)
17	source plus destination (with saturation)
18	destination minus source (with overflow)
19	destination minus source (with saturation)
20	MAX(source, destination)
21	MIN(source, destination)

When initialized by the *set_config* function, the PPOP code is set to its default value of 0 (*replace* operation). The PPOP code can be altered with a call to the *set_ppop* function.

The pixel-processing operation code corresponds to the 5-bit PPOP field in the TMS340 graphics processor's CONTROL register. The effects of the 22 different codes are described in more detail in the user's guides for the TMS34010 and TMS34020.



Example Use the *set_ppop* function to set the current pixel-processing operation code to 10 (*exclusive-OR*). Use the *fill_rect* function from the Extended Primitives Library to fill two rectangles that partially overlap. The overlapping region shows the effect of exclusive-ORing identical source and destination pixel values.

```
#define  XOR   10          /* pixel processing operation code */

main()
{
     set_config(0, !0);
     clear_screen(-1);

     set_ppop(XOR);
     fill_rect(100, 20, 10, 50);
     fill_rect(20, 100, 50, 10);
}
```

set_text_xy *Set Text x–y Position*

Syntax

```
void set_text_xy(x, y)
short x, y;                    /* text x–y coordinates */
```

Description

The *set_text_xy* function sets the text-drawing position to the specified x-y coordinates. This is the position at which the next character (or string of characters) will be drawn if a subsequent call is made to the *text_outp* function. Both the *text_outp* and *text_out* functions automatically update the text position to be the right edge of the last string output to the screen.

Arguments *x* and *y* are the coordinates of the new text position on the screen, specified relative to the current drawing origin. Argument *x* is the x coordinate at the left edge of the next string output by the *text_outp* function. Argument *y* is the y coordinate at either the top of the string, or the base line, depending on the state of the text alignment attribute (see the description of the *set_textattr* function).

Example

Use the *set_text_xy* function to set the text-drawing position to x-y coordinates (10, 20). Use the *text_outp* function to print a text string to the screen starting at these coordinates.

```
main()
{
    set_config(0, 1);
    clear_screen(–1);
    set_text_xy(10, 20);
    text_outp(”hello, world”);
}
```

Syntax

```
void set_transp(mode)
short mode;              /* transparency mode */
```

Description

The *set_transp* function, if implemented, changes the transparency mode. When the transparency attribute is enabled, the transparency mode sets the conditions under which a pixel is determined to be transparent. During a graphics output operation, a nontransparent pixel replaces the original destination pixel but a transparent pixel does not.

The *set_transp* function is implemented only on TMS34020 systems. Currently, the modes supported on TMS34020 systems are

mode = 0	Transparent if result equal to zero
mode = 1	Transparent if source equal to COLOR0
mode = 5	Transparent if destination equal to COLOR0

Argument *mode* must be set to one of these values. Specifying an invalid mode number may result in undefined behavior.

On TMS34010 systems, the *set_transp* function is not implemented, and only transparency mode 0 is supported.

The enabling and disabling of transparency, regardless of the mode selected, is performed by two other functions, *transp_on* and *transp_off*. Refer to the descriptions of these functions for more information.

Immediately after initialization of the drawing environment by the *set_config* function, the system is configured in transparency mode 0, which is the default.

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

PTR set_vector(trapnum, gptr)
unsigned short trapnum;      /* trap number */
PTR gptr;                    /* pointer to GSP memory */
```

Description

The *set_vector* function loads one of the TMS340 graphics processor's trap vectors with a pointer to a location in the processor's memory. This function provides a portable means of loading the entry point to a trap service routine, regardless of whether the actual trap vector is located in RAM or ROM.

Argument *trapnum* specifies a trap number in the range –32768 to 32767 for a TMS34020, and 0 to 31 for a TMS34010. Argument *gptr* is a pointer containing the 32-bit memory address to be loaded into the trap vector.

The value returned by the function is the original 32-bit TMS340 graphics processor address contained in the designated trap vector at the time of the call.

Example

Use the *set_vector* function to load the trap-3 vector with the address of a trap service routine. The service routine simply increments a global counter. The progress of the count is displayed graphically on the screen as a moving asterisk. Note that the C compiler recognizes "c_int03" as the name of an interrupt routine and terminates the routine with a RETI (return from interrupt) rather than a RETS (return from subroutine) instruction.

```
#include  <gsptypes.h>          /* defines CONFIG and FONTINFO */
static long count;

c_int03()
{
    count++;
}

main()
{
    int n;
    char c[40];

    set_config(0, !0);
    clear_screen(–1);
    for (n = 0; n < 32; n++)
                      c[n] = ’ ’;
    c[32] = ’\0’;

    /* Install trap service routine. */
    count = 0;
    set_vector(3, c_int03);

    /* Trap once per loop. */
    for (n = 0; ; ) {
        asm(”    TRAP   3    ”);
        c[n] = ’ ’;
        c[n = count/32 & 31] = ’*’;
        text_out(10, 10, c);
    }
}
```

set_windowing *Set Window-Clipping Mode*

Syntax

```
void set_windowing(mode)
short mode;
```

Description

The *set_windowing* function loads the specified value into the 2-bit windowing field contained in the CONTROL I/O register. This function is provided for the sake of backward compatibility with early versions of TIGA.

The four windowing modes are

00_2	No windowing.
01_2	Interrupt request on write in window.
10_2	Interrupt request on write outside window.
11_2	Clip to window.

Take care in using this function. The library's drawing functions assume that the TMS340 graphics processor is configured in windowing mode 3. Changing the windowing mode from this default may result in undefined behavior. The code specified for the window-clipping mode corresponds to the 2-bit W field in the TMS340 graphics processor's CONTROL register. The effects of the W field on window-clipping operations are described in the user's guides for the TMS34010 and TMS34020.

Immediately following initialization of the drawing environment by the *set_config* function, the system is configured in windowing mode 3, which is the default.

Syntax

```
typedef unsigned long PTR;   /* 32–bit GSP memory address */

void set_wksp(addr, pitch)
PTR addr;                    /* starting address */
PTR pitch;                   /* workspace pitch */
```

Description

The *set_wksp* function specifies an off-screen workspace. None of the current TIGA core or extended primitives makes use of this workspace; it is provided to support future graphics extensions that require storage for edge flags or region-of-interest masks.

Argument *addr* is the base address of the off-screen workspace. Argument *pitch* is the difference in memory addresses of two adjacent rows in the off-screen workspace. The pitch is required to be a power of two and a multiple of 16. The exception to this requirement is that the pitch argument is specified as 0 in the event that *no* workspace is allocated (in which case the value of the *addr* argument is a “don’t care.”)

The off-screen workspace is a 1-bit-per-pixel bit map of the same width and height as the screen. If the display hardware provides sufficient off-screen memory, the workspace can be allocated statically at link time. By convention, the workspace pitch retrieved by the *get_wksp* function is nonzero when a workspace is allocated; the pitch can be checked following initialization to determine whether a workspace is statically allocated. The workspace can be allocated dynamically by calling the *set_wksp* function with the address of a valid workspace in memory and a nonzero pitch; it can be deallocated by calling *set_wksp* with a pitch of 0.

Not all TMS340 graphics processor-based display configurations may contain sufficient memory to allocate (statically or dynamically) an off-screen workspace. For this reason, proprietary extensions to the Core Primitives Library that require use of the workspace may be unable to execute on some systems.

Syntax

```
short text_out(x, y, s)
short x, y;                 /* starting coordinates */
unsigned char *s;           /* character string */
```

Description

The *text_out* function draws a character string to the screen in the currently selected font.

Arguments *x* and *y* are the starting coordinates of the string, relative to the current drawing origin. Argument *s* is a string of 8-bit ASCII characters terminated by a *null* (0) character.

The string is rendered in the currently selected font using the current text-drawing attributes.

Argument *x* is the x coordinate at the left edge of the string. Argument *y* is the y coordinate at either the top of the string or the base line, depending on the state of the text alignment attribute. During initialization of the drawing environment by the *set_config* function, the alignment is set to its default position, at the top left corner. The attribute can be modified by means of a call to the *set_textattr* function.

The return value is the x coordinate of the next character position to the right of the string. If the string lies entirely above or below the clipping rectangle, the unmodified starting x coordinate is returned.

Example

Use the *text_out* function to write a single line of text to the screen in the system font.

```
main()
{
    set_config(0, !0);
    clear_screen(-1);
    text_out(10, 10, "Hello world.");
}
```

Syntax

```
void text_outp(s)
unsigned char *s;
```

Description

The *text_outp* function outputs text to the screen, starting at the current text drawing position. The specified string of characters is rendered in the currently selected font and with the current text-drawing attributes. The text position must have been specified by a previous call to the *set_text_xy*, *text_out* or *text_outp* function.

Argument *s* is a string of 8-bit ASCII character codes terminated by a *null* (0) character.

After printing the text on the screen, the function automatically updates the text position to be the position of the next character to the right of the string just printed. A subsequent call to the *text_outp* function will result in the next string being printed beginning at this position.

Unlike the *text_out* function, the *text_outp* function does not return a value.

Example

Use the *text_outp* function to mix two fonts in the same line of of text. The TI Roman size 20 and TI Helvetica size 22 fonts will be used. Use the set_textattr function to align the text to the base line.


```
#include <gsptypes.h>           /* FONT type definition */
extern FONT ti_rom20, ti_hel22; /* 2 different fonts    */
static FONTINFO fontinfo;
main()
{
    int i, j;

    set_config(0, 1);
    clear_screen(-1);
    i = install_font(&ti_rom20);
    j = install_font(&ti_hel22);
    set_textattr("%1a", 0, 0);
    select_font(i);
    get_fontinfo(0, &fontinfo);
    set_text_xy(0, fontinfo.charhigh);
    text_outp(" Concatenate");
    select_font(j);
    text_outp(" one font");
    select_font(i);
    text_outp(" with another.");
}
```

transp_off *Turn Transparency Off*

Syntax `void transp_off()`

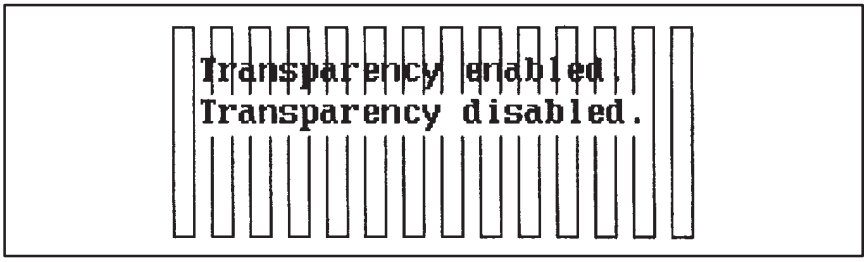
Description The *transp_off* function disables transparency for subsequent drawing operations.

Transparency is an attribute that affects drawing operations. Several transparency modes are supported. During initialization of the drawing environment by the *set_config* function, transparency is disabled and the transparency mode is set to the default, mode 0. The TMS34010 supports only transparency mode 0, but the TMS34020 supports additional modes. Refer to the description of the *set_transp* function for details.

In transparency mode 0, 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. For instance, to avoid modifying destination pixels in the rectangular region surrounding each character shape, you can enable transparency before you call the *text_out* or *text_outp* function.

The effect of transparency in conjunction with the pixel-processing operation and the plane mask is described in the user's guides for the TMS34010 and TMS34020.

Example Use the *transp_off* function to demonstrate the effect of disabling transparency. Use the *draw_rect* function from the Extended Primitives Library to construct a background pattern. To show that the background pattern is preserved in the rectangle surrounding each character, use the *text_out* function to draw a line of text to the screen with transparency enabled. Also, draw a line of text to the screen with transparency disabled to show that the background pattern is overwritten.



```
#include  <gsptypes.h>          /* defines FONTINFO structure */

main()
{
    short x, y;
    FONTINFO fntinf;

    set_config(0, !0);
    clear_screen(–1);
    get_fontinfo(–1, &fntinf);
    for (x = y = 0; x < 200; x += 15)
         draw_rect(8, 80, x, y);
    x = y = 10;
    transp_on();
    text_out(x, y, ”Transparency enabled.”);
    transp_off();
    text_out(x, y+fntinf.charhigh, ”Transparency disabled.”);
}
```

transp_on *Turn Transparency On*

Syntax `void transp_on()`

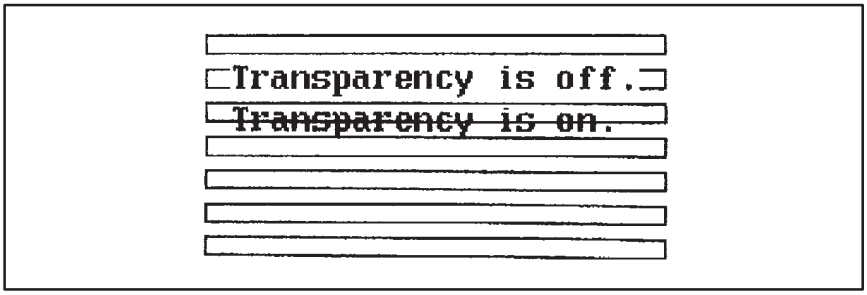
Description The *transp_on* function enables transparency for subsequent drawing operations.

Transparency is an attribute that affects drawing operations. Several transparency modes are supported. During initialization of the drawing environment by the *set_config* function, transparency is disabled and the transparency mode is set to the default, mode 0. The TMS34010 supports only transparency mode 0, but the TMS34020 supports additional modes. Refer to the description of the *set_transp* function for details.

In transparency mode 0, 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. For instance, to avoid modifying destination pixels in the rectangular region surrounding each character shape, you can enable transparency before you call the *text_out* or *text_outp* function.

The effect of transparency in conjunction with the pixel-processing operation and the plane mask is described in the user's guides for the TMS34010 and TMS34020.

Example Use the *transp_on* function to demonstrate the effect of enabling transparency. Use the *draw_rect* function from the Extended Primitives Library to construct a background pattern. To show that the background pattern is overwritten in the rectangle surrounding each character, use the *text_out* function to draw a line of text to the screen with transparency disabled. Also, draw a line of text to the screen with transparency enabled to show that the background pattern is preserved.



```
#include  <gsptypes.h>          /* defines FONTINFO structure */

main()
{
    short x, y;
    FONTINFO fntinf;

    set_config(0, !0);
    clear_screen(–1);
    get_fontinfo(–1, &fntinf);
    for (x = y = 0; y < 80; y += 13)
         draw_rect(180, 7, x, y);
    x = y = 10;
    text_out(x, y, ”Transparency is off.”);
    transp_on();
    text_out(x, y+fntinf.charhigh, ”Transparency is on.”);
}
```

Syntax

```
void wait_scan(line)
short line;                    /* scan line number */
```

Description

The *wait_scan* function waits for the monitor to scan a designated line on the screen.

Argument *line* is the scan line number. Scan lines are numbered in ascending order, starting with line 0 at the top of the screen. Given a display of N lines, valid arguments are in the range 0 to N–1. If argument *line* is less than 0, the function uses the value 0 in place of the argument value. If argument *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 number of scan lines on the screen in the current graphics mode is specified in the *disp_vres* field of the CONFIG structure returned by the *get_config* function.

Once the function is called, it does not return control to the calling routine until the designated line is scanned by the monitor's electron beam. Control is returned at the start of the horizontal blanking interval that follows the scan line.

This function is used to synchronize drawing operations with the position of the electron beam on the monitor screen. 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 being drawn at the same time it is being scanned on the monitor.

The *wait_scan* function is typically used to achieve a limited degree of smooth animation in graphics modes that provide only a single video page (or frame buffer). The *page_flip* and *page_busy* functions support double buffering in modes that provide more than one page. Double buffering, when available, is usually preferred for animation applications.

Example Use the *wait_scan* function to smoothly animate a rotating asterisk. The position of the asterisk is updated once per frame. Before drawing the asterisk in its updated position, the *wait_scan* function is utilized to delay erasing the asterisk until the area just beneath it is being scanned. The asterisk is erased by overwriting it with a space character. This technique works well with the system font, which is a block font, but might produce unexpected results if used with a proportionally spaced font.

```
#include  <gsptypes.h>          /* defines FONTINFO structure */
#define  RADIUS   60            /* radius of revolution */

main()
{
    long x, y;
    short i, j;
    FONTINFO fntinf;

    set_config(0, !0);
    clear_screen(-1);
    get_fontinfo(-1, &fntinf);
    x = RADIUS << 16;
    y = 0;
    for (i = j = 0; ; ) {
        wait_scan(j+fntinf.charhigh);
        text_out(i, j, " ");
        i = RADIUS + (x >> 16);
        j = RADIUS + (y >> 16);
        text_out(i, j, "*");
        x -= y >> 4;
        y += x >> 4;
    }
}
```

Chapter 7

Extended Primitives

This chapter describes the functions in the Extended Primitives Library. The Core Primitives Library is described in the preceding chapter.

Remember to call the *set_config* function (a member of the Core Primitives Library) to initialize the drawing environment before you call any of the other functions in the Core and Extended Primitives Libraries.

The table below summarizes the 58 functions in the Extended Primitives Library. The remainder of this chapter is an alphabetical, detailed description of the syntax, usage, and operation of each function. These descriptions are augmented by complete example programs that can be compiled and run exactly as written.

Function Name	Description
bitblt	Transfer bit-aligned block
decode_rect	Decode rectangular image
delete_font	Delete font
draw_line	Draw line
draw_oval	Draw oval
draw_ovalarc	Draw oval arc
draw_piearc	Draw pie arc
draw_point	Draw point
draw_polyline	Draw polyline
draw_rect	Draw rectangle
encode_rect	Encode rectangular image
fill_convex	Fill convex polygon
fill_oval	Fill oval
fill_piearc	Fill pie arc
fill_polygon	Fill polygon
fill_rect	Fill rectangle
frame_oval	Fill oval frame

Function Name	Description
frame_rect	Fill rectangular frame
get_env	Get graphics environment information
get_pixel	Get pixel
get_textattr	Get text attributes
in_font	Verify characters in font
install_font	Install font
move_pixel	Move pixel
patnfill_convex	Fill convex polygon with pattern
patnfill_oval	Fill oval with pattern
patnfill_piearc	Fill pie arc with pattern
patnfill_polygon	Fill polygon with pattern
patnfill_rect	Fill rectangle with pattern
patnframe_oval	Fill oval frame with pattern
patnframe_rect	Fill rectangular frame with pattern
patnpen_line	Draw line with pen and pattern
patnpen_ovalarc	Draw oval arc with pen and pattern
patnpen_piearc	Draw pie arc with pen and pattern
patnpen_point	Draw point with pen and pattern
patnpen_polyline	Draw polyline with pen and pattern
pen_line	Draw line with pen
pen_ovalarc	Draw oval arc with pen
pen_piearc	Draw pie arc with pen
pen_point	Draw point with pen
pen_polyline	Draw polyline with pen
put_pixel	Put pixel
seed_fill	Seed fill
seed_patnfill	Seed fill with pattern
select_font	Select font
set_draw_origin	Set drawing origin
set_dstbm	Set destination bit map
set_patn	Set fill pattern
set_pensize	Set pen size
set_srcbm	Set source bit map
set_textattr	Set text attributes

Function Name	Description
styled_line	Draw styled line
styled_oval	Draw styled oval
styled_ovalarc	Draw styled oval arc
styled_piearc	Draw styled pie arc
swap_bm	Swap source and destination bit maps
text_width	Get width of text string
zoom_rect	Zoom rectangle

Syntax

```
void bitblt(w, h, xs, ys, xd, yd)
short w, h;      /* width and height of both bit maps */
short xs, ys;    /* source array coordinates */
short xd, yd;    /* destination array coordinates */
```

Description

The *bitblt* function copies a two-dimensional array of pixels from the current source bit map to the current destination bit map. The source and destination bit maps are specified by calling the *set_srcbm* and *set_dstbm* functions before calling the *bitblt* function. Calling the *set_config* function with the *init_draw* argument set to true causes both the source and destination bit maps to be set to the default bit map, which is the screen.

The source and destination arrays are assumed to be rectangular, two-dimensional arrays of pixels. The two arrays are assumed to be of identical width and height. The *bitblt* function accepts source and destination arrays that have the same pixel size. If the pixel sizes are not equal, the pixel size for either the source or the destination must be 1. Other combinations of source and destination pixel sizes are not accepted by the function.

Arguments *w* and *h* specify the width and height common to the source and destination arrays. Arguments *xs* and *ys* specify the x-y coordinates of the top left corner (lowest memory address) of the source array as a displacement from the origin (base address) of the source bit map. Arguments *xd* and *yd* specify the x-y coordinates of the top left corner of the destination array as a displacement from the origin of the destination bit map.

If the source and destination pixel sizes are equal, then pixels in the source array are copied to the destination. During the copying process, the pixels may be modified, depending on the current pixel-processing operation, transparency mode, and plane mask.

If the source bit map's pixel size is 1 and the destination pixel size is greater than 1, source pixels are expanded to color in the destination array. During the expansion process, pixels corresponding to 1s in the source bit map are expanded to the current foreground color before being drawn to the destination; 0s are expanded to the current background color.

If the destination bit map's pixel size is 1 and the source pixel size is greater than 1, *bitblt* performs a contract function on the source before writing to the destination. During the contraction process, destination pixels are set to 0 if they correspond to source pixels that are equal to the background color; all other destination pixels are set to 1.

When the source or destination bit map is the screen, the specified source or destination coordinates are defined relative to the current drawing origin. In the case of a linear bit map contained in an off-screen buffer, the *bitblt* function calculates the memory address of a pixel from the specified x and y coordinates as follows:

address = baseaddr + y*(pitch) + x*(psize)

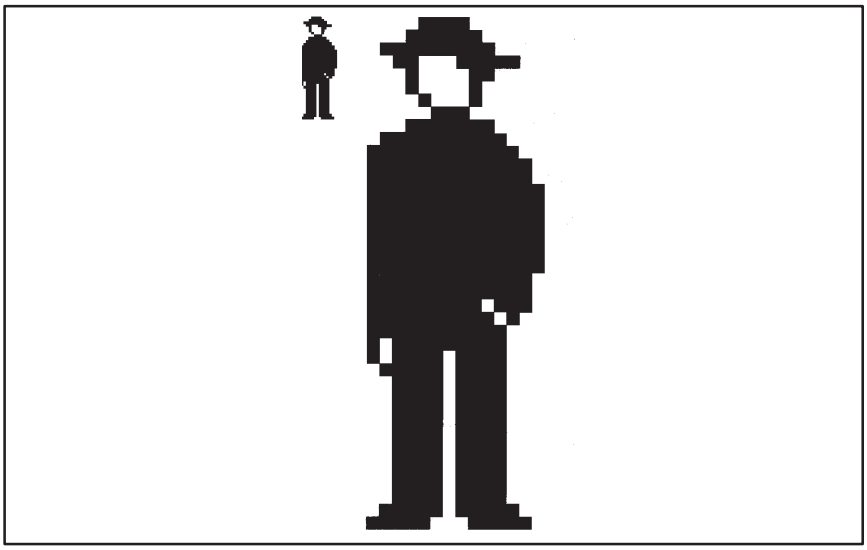
where *baseaddr, pitch,* and *psize* are the argument values passed to the *set_dstbm* or *set_srcbm* function.

When the destination bit map is set to the screen, the function clips the destination bit map to the current rectangular clipping window. When the source bit map is set to the screen and any portion of the source array lies in negative screen coordinate space, the source rectangle is clipped to positive x-y coordinate space; in most systems this means that the source is clipped to the top and left edges of the screen. The resulting clipped source rectangle is copied to the destination rectangle and justified to the lower right corner of the specified destination rectangle. Portions of the destination array corresponding to clipped portions of the source are not modified.

The clipping window for a linear bit map encloses the pixels in the x-y coordinate range (0,0) to (*xext*, *yext*), where *xext* and *yext* are arguments passed to *set_dstbm* or *set_srcbm*. The *bitblt* function itself performs no clipping in the case of a linear bit map; responsibility for clipping is left to the calling routine.

If both source and destination bit maps are set to the screen, then the function correctly handles the case in which the rectangular areas containing the source and destination bit maps overlap. In other words, the order in which the pixels of the source are copied to the destination is automatically adjusted to prevent any portion of the source from being overwritten before it has been copied to the destination.

Example Use the *bitblt* function to color-expand an image contained in a 1-bit-per-pixel bit map to the screen. The original image is 16 pixels wide, 40 pixels high, and has a pitch of 16. Use the *zoom_rect* function to zoom the screen image by a factor of 5.



```
#define  PITCH        16     /* pitch of image bit map */
#define  W            16     /* width of image bit map */
#define  H            40     /* height of image bit map */
#define  IMAGEDEPTH   1      /* pixel depth of image bit map */
#define  SCREENDEPTH  4      /* pixel depth of screen */
#define  ZOOM         5      /* zoom factor */

typedef enum { FIELDWIDTH = 1 } BIT;

static BIT image[] = {
    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,
    0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,
    1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
    1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
    1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
    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,
    0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,
    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,
};

main()
{
    char buf[ZOOM*W*SCREENDEPTH/8]; /* zoom_rect buffer */

    set_config(0, !0);
    clear_screen(0);
    set_srcbm(image, PITCH, W, H, IMAGEDEPTH);
    bitblt(W, H, 0, 0, 10, 10);
    set_srcbm(0);
    zoom_rect(W, H, 10, 10, ZOOM*W, ZOOM*H, 20+W, 10, buf);
}
```

decode_rect *Decode Rectangular Image*

Syntax

```
typedef unsigned long PTR;   /* 32-bit GSP memory address */

short decode_rect(xleft, ytop, buf)
short xleft, ytop;           /* top left corner */
PTR buf;                     /* image buffer */
```

Description

The *decode_rect* function restores a previously compressed image to the screen. The image was previously encoded by the *encode_rect* function. The image is rectangular and is restored at the same width, height, and pixel size as the image originally encoded by the *encode_rect* function.

The first two arguments, *xleft* and *ytop*, specify the x and y coordinates at the top left corner of the destination rectangle and are defined relative to the drawing origin.

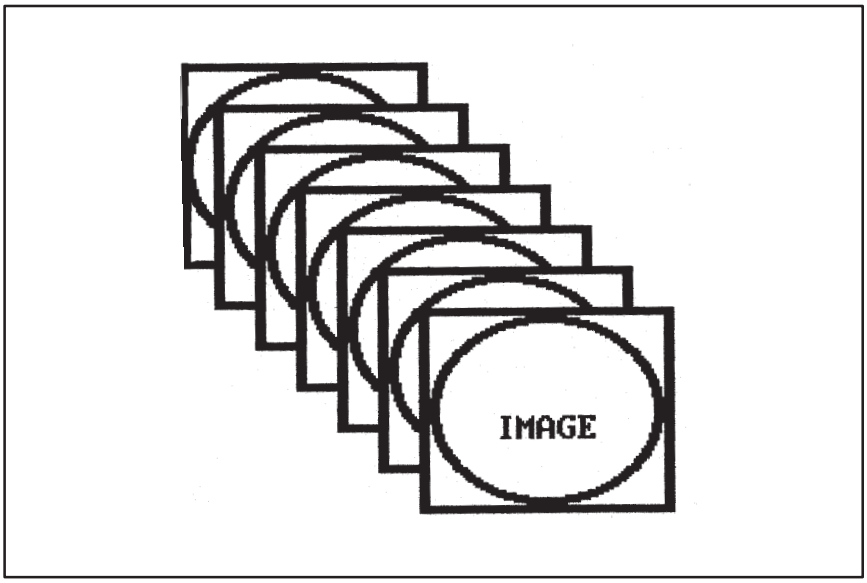
The final argument, *buf*, is a pointer to a buffer in the TMS340 graphics processor's memory in which the compressed image is stored.

The function returns a nonzero value if it has successfully decoded the image; otherwise, the return value is 0.

Refer to the description of the *encode_rect* function for a discussion of the format in which the compressed image is saved.

Example

Use the *decode_rect* function to decompress multiple copies of a rectangular image that was previously captured from the screen by the *encode_rect* function.



```
#define  MAXSIZE      4096    /* max picture size in bytes */

static char image[MAXSIZE];

main()
{
    short w, h, x, y, n;
    char *s;

    set_config(0, !0);
    clear_screen(-1);

    /* Create an image on the screen. */
    w = 100;
    h = 80;
    x = 10;
    y = 10;
    frame_rect(w, h, x, y, 4, 3);
    frame_oval(w-8, h-6, x+4, y+3, 4, 3);
    s = "IMAGE";
    n = text_width(s);
    text_out(x+(w-n)/2, y+h/2, s);

    /* Compress image. */
    encode_rect(w, h, x, y, image, sizeof(image), 0);

    /* Now decompress the image several times. */
    for (n = x ; n <= x + w; n += 16)
         decode_rect(n, n, image);
}
```

delete_font *Delete Font*

Syntax

```
short delete_font(id)
short id;     /* font identifier */
```

Description

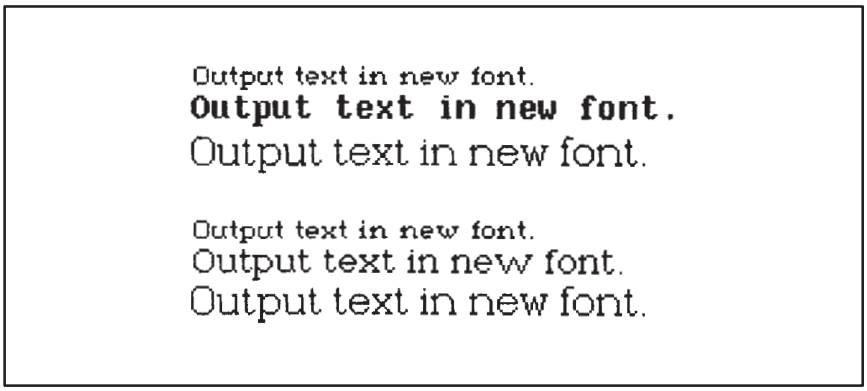
The *delete_font* function removes from the font table the installed font designated by an identifier. The font is identified by argument *id*, which contains the value returned from the *install_font* function at the time the font was installed.

A nonzero value is returned if the font was successfully removed. A value of 0 is returned if argument *id* is invalid; that is, if *id* does not correspond to an installed font.

If the font removed was also the one selected for current text drawing operations, the system font is automatically selected by the function. A request to delete the system font (*id* = 0) will be ignored by the function, and a value of 0 will be returned.

Example

Use the *delete_font* function to delete a font that was previously installed. First, install and select three fonts. The first and third fonts installed by the example program are proportionally spaced fonts. The second font is a block font. The three fonts are used to write three lines of text to the screen. At this point, the block font is deleted with the *delete_font* function, and another proportionally spaced font is installed in its place. An additional three lines of text are written to the screen using the three installed fonts. This example includes the C header file `gsptypes.h`, which defines the FONT and FONTINFO structures. The TI Roman font sizes 11, 14, and 16 must be linked with the program.



```
#include <gsptypes.h>    /* defines FONT and FONTINFO structures */

#define NFONTS   3       /* number of fonts installed */
#define NLINES   2       /* number of lines of text per font */

extern FONT sys16, ti_rom11, ti_rom14, ti_rom16;/* 4 font names */

   main()
   {
       FONTINFO fontinfo;
       short index[NFONTS];
       int i, j, x, y;

       set_config(0, !0);
       clear_screen(0);
       index[0] = install_font(&ti_rom11);
       index[1] = install_font(&sys16);    /* install block font */
       index[2] = install_font(&ti_rom16);
       x = y = 10;
       for (i = 0; i < 2; i++) {
          for (j = 0; j < NFONTS; j++) {
              select_font(index[j]);
              get_fontinfo(index[j], &fontinfo);
              text_out(x, y, "Output text in new font.");
              y += fontinfo.charhigh;
          }
          y += fontinfo.charhigh;
          delete_font(index[1]); /* delete block font */
          index[1] = install_font(&ti_rom14);
       }
   }
```

draw_line *Draw Line*

Syntax

```
void draw_line(x1, y1, x2, y2)
short x1, y1;    /* start coordinates */
short x2, y2;    /* end coordinates */
```

Description

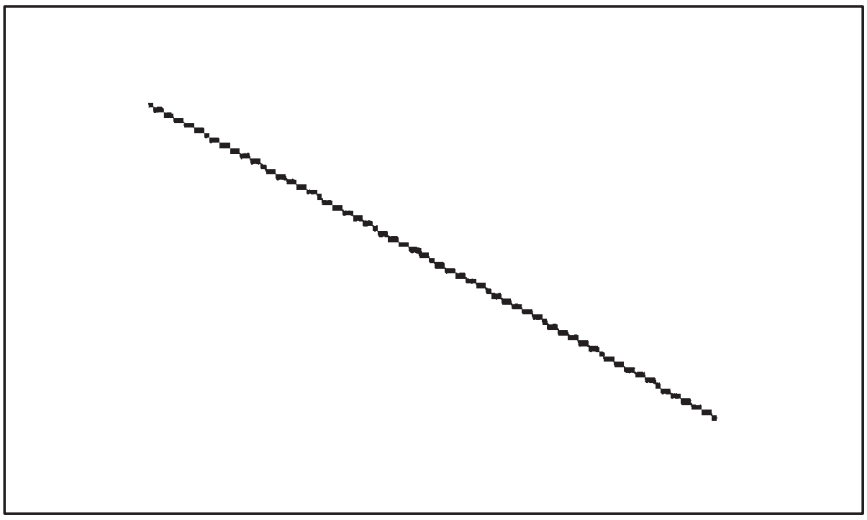
The *draw_line* function uses Bresenham's algorithm to draw a straight line from the starting point to the ending point. The line is one pixel thick and is drawn in the current foreground color.

Arguments *x1* and *y1* specify the starting x and y coordinates of the line, and arguments *x2* and *y2* specify the ending coordinates.

In the case of a line that is more horizontal than vertical, the number of pixels used to render the line is $1 + |x_2 - x_1|$. The number of pixels for a line that is more vertical than horizontal is $1 + |y_2 - y_1|$.

Example

Use the draw_line function to draw a line from (10, 20) to (120, 80).



```
main()
{
    short x1, y1, x2, y2;

    set_config(0, !0);
    clear_screen(0);
    x1 = 10;
    y1 = 20;
    x2 = 120;
    y2 = 80;
    draw_line(x1, y1, x2, y2);
}
```

Syntax

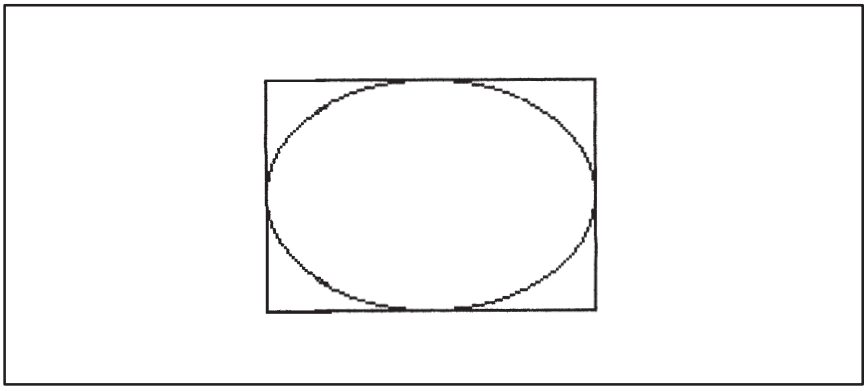
```
void draw_oval(w, h, xleft, ytop)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
```

Description The *draw_oval* function draws the outline of an ellipse, given the 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 outline of the ellipse is one pixel thick and is drawn in the current foreground color.

The four arguments specify the rectangle enclosing the ellipse:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

Example Use the *draw_oval* function to draw an ellipse. The ellipse is 130 pixels wide and 90 pixels high. Also, draw a rectangle that circumscribes the ellipse.



```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    draw_oval(w, h, x, y);
    draw_rect(w, h, x, y);
}
```

draw_ovalarc *Draw Oval Arc*

Syntax

```
void draw_ovalarc(w, h, xleft, ytop, theta, arc)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
short theta;        /* starting angle (degrees) */
short 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 coordinate axes. The ellipse from which the arc is taken is specified in terms of the enclosing rectangle in which it is inscribed. The arc is one pixel thick and is drawn in the current foreground color.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

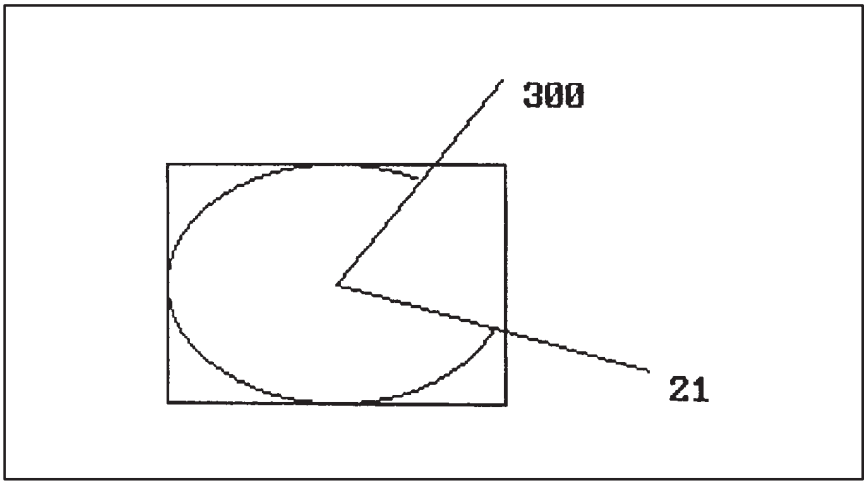
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *draw_ovalarc* function to draw an arc that extends from 21 degrees to 300 degrees. The ellipse from which the arc is taken is 130 pixels wide and 90 pixels high. Also, draw a rectangle enclosing the arc and draw two rays from the center of the ellipse through the start and end points of the arc.




```
#define  PI  3.141592654
#define  K   (PI/180.0)       /* convert degrees to radians */

main()
{
    extern double cos(), sin();
    double a, b;
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 40;
    y = 50;
    draw_rect(w, h, x, y);
    draw_ovalarc(w, h, x, y, 21, 300-21);

    /* Now draw the two rays. */
    set_draw_origin(x+w/2, y+h/2);
    a = w;
    b = h;
    x = a*cos(21.0*K) + 0.5;
    y = b*sin(21.0*K) + 0.5;
    draw_line(0, 0, x, y);
    text_out(x, y, " 21");     /* label ray at 21 degrees */
    x = a*cos(300.0*K) + 0.5;
    y = b*sin(300.0*K) + 0.5;
    draw_line(0, 0, x, y);
    text_out(x, y, " 300");    /* label ray at 300 degrees */
}
```

draw_piearc *Draw Pie Arc*

Syntax

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

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 the standard position, with the major and minor axes parallel to the coordinate axes. The ellipse from which the arc is taken is specified in terms of the enclosing rectangle in which it is inscribed. The arc and the two lines are all one pixel thick and are drawn in the current foreground color.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

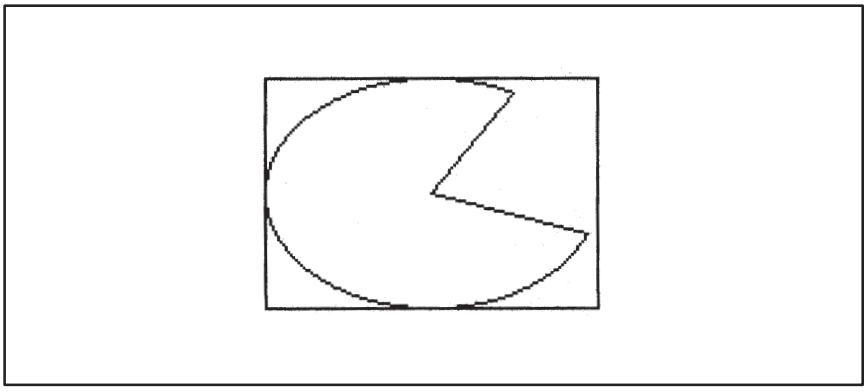
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *draw_piearc* function to draw a pie chart corresponding to a slice of an ellipse from 21 degrees to 300 degrees. The ellipse is 130 pixels wide and 90 pixels high. Draw an enclosing rectangle of the same dimensions.



```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    draw_piearc(w, h, x, y, 21, 300–21);
    draw_rect(w, h, x, y);
}
```

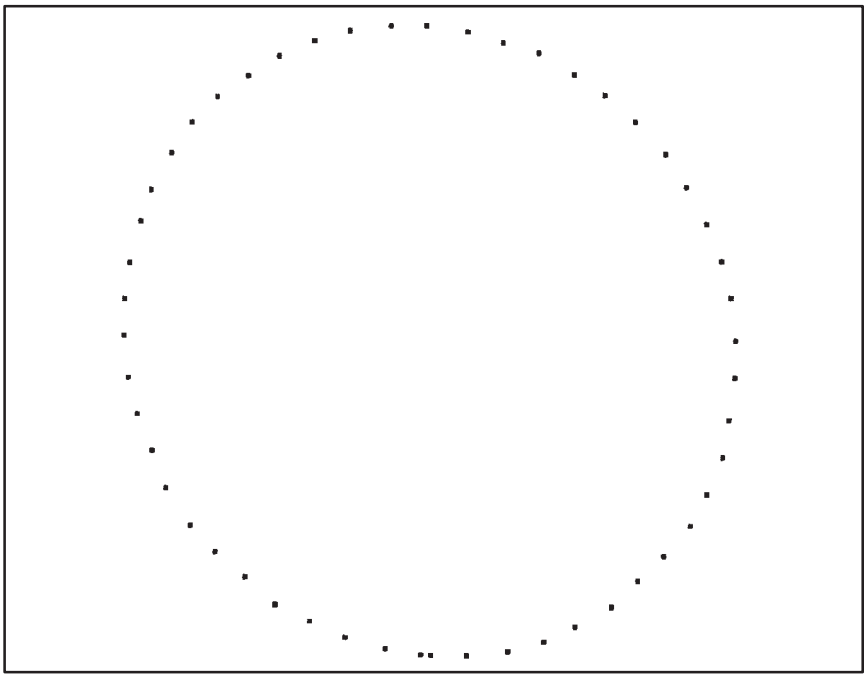
draw_point *Draw Point*

Syntax

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

Description The *draw_point* function draws a point represented as a single pixel. Arguments *x* and *y* specify the x-y coordinates of the designated pixel and are defined relative to the drawing origin. The pixel is drawn in the current foreground color.

Example Use the *draw_point* function to draw a circle of radius 60 in the top left corner of the screen. Each point on the circle is separated from its two neighbors by angular increments of approximately 1/8 radian.



```
#define  TWOPI        411775   /* fixed-point 2*PI */
#define  HALF         32768    /* fixed-point 1/2 */
#define  RADIUS       60       /* radius of circle */
#define  N            3        /* angular increment = 1/2**N
                                    radians */

typedef long FIX;              /* fixed-pt with 16-bit fraction */

main()
{
    short x, y;
    int i;
    FIX u, v, xc, yc;

    set_config(0, !0);
    clear_screen(0);
    u = 0;
    v = RADIUS << 16;    /* convert to fixed-pt */
    xc = yc = v + HALF;   /* fixed-pt center coord's */
    for (i = (TWOPI << N) >> 16; i >= 0; i--) {
        x = (u + xc) >> 16;
        y = (v + yc) >> 16;
        draw_point(x, y);
        u - = v >> N;
        v + = u >> N;
    }
}
```

draw_polyline *Draw Polyline*

Syntax

```
typedef struct { short x, y; } POINT;

void draw_polyline(n, vert)
short n;         /* vertex count */
POINT *vert;     /* vertex coordinates */
```

Description

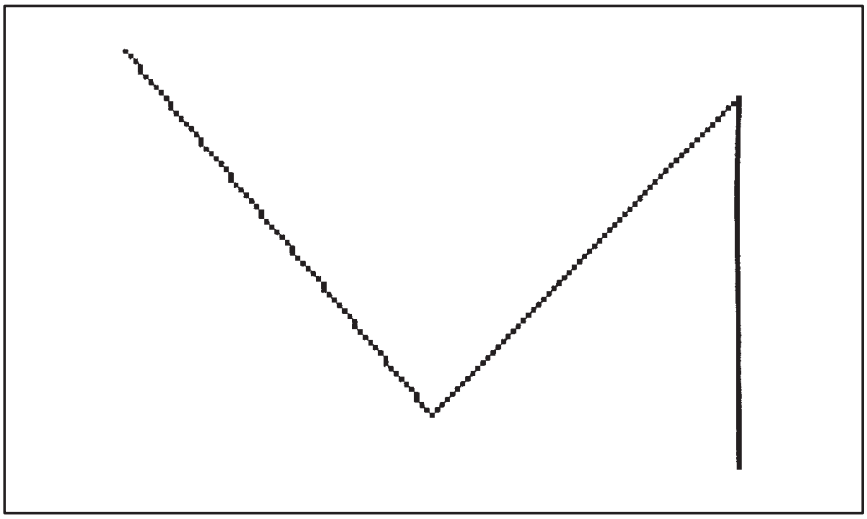
The *draw_polyline* function draws multiple, connected lines. An array of integer x-y coordinates representing the polyline vertices is specified as one of the arguments. A straight line is drawn between each pair of adjacent vertices in the array. Each line is constructed with Bresenham's algorithm, is one pixel thick, and is drawn in the current foreground color.

Argument *n* specifies the number of vertices in the polyline; the number of lines drawn is *n*–1.

Argument *vert* is an array of x-y coordinates representing the polyline vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function draws a line between each adjacent pair of vertices in the array. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

Note that for the polyline to form a closed polygon, the calling program must ensure that the first and last vertices in the *vert* array are the same.

Example Use the *draw_polyline* function to draw a three-segment polyline. The four vertices are located at coordinates (0, 0), (60, 70), (120, 10), and (120, 80).



```
#define  NVERTS    4    /* number of vertices */

typedef struct { short x, y; } POINT;

static POINT xy[NVERTS] =
{
    { 0, 0 }, { 60, 70 }, { 120, 10 }, { 120, 80 }
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    draw_polyline(NVERTS, xy);
}
```

Syntax

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

Description

The *draw_rect* function draws the outline of a rectangle. The rectangle consists of two horizontal and two vertical lines. Each line is one pixel thick and is drawn in the current foreground color.

The four arguments specify the rectangle:

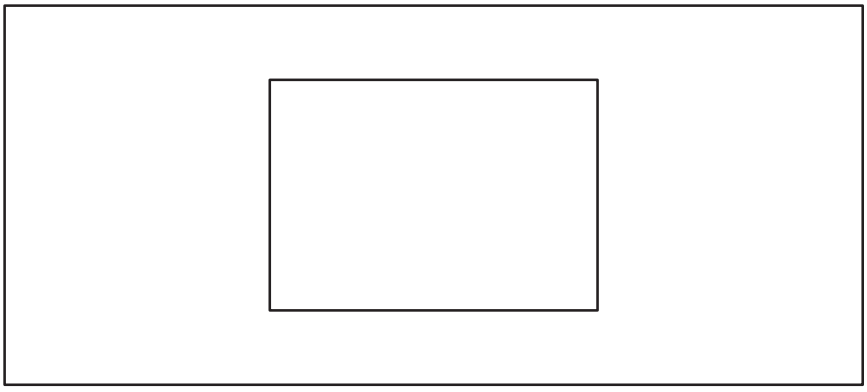
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

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);
```

Example

Use *draw_rect* function to draw a rectangle that is 130 pixels wide and 90 pixels high.



```
main()
{
    set_config(0, !0);
    clear_screen(0);
    draw_rect(130, 90, 10, 10);
}
```

Syntax

```
typedef unsigned long PTR; /* 32-bit GSP memory address */

unsigned long encode_rect(w, h, xleft, ytop, buf, bufsize,
                                                    scheme)
short w, h;                /* rectangle width and height */
short xleft, ytop;         /* top left corner */
PTR buf;                   /* image buffer */
unsigned long bufsize;     /* buffer capacity in bytes */
unsigned short scheme;     /* encoding scheme */
```

Description

The *encode_rect* function uses the specified encoding scheme to save an image in compressed form. The image to be saved is contained in a specified rectangular portion of the screen. The function compresses the image and saves it in a specified destination buffer.

Once an image has been encoded by the *encode_rect* function, it can be decompressed and restored to a designated area of the screen by the *decode_rect* function. The image is restored at the same width, height, and pixel size as the original image saved by the *encode_rect* function.

The first four arguments specify the rectangular region of the screen containing the original image:

- Arguments *w* and *h* specify the width and height (in pixels) of the rectangle containing the image.
- Arguments *xleft* and *ytop* specify the x and y coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The next two arguments specify the destination array for the compressed image:

- Argument *buf* is a pointer to the destination array.
- Argument *bufsize* is the storage capacity of the *buf* array in bytes.

The final argument, *scheme*, specifies the encoding scheme to be used. Currently, only run-length encoding is supported, for which the value of *scheme* must be specified as 0.

The value returned by the function is the number of 8-bit bytes required to encode the image, including the header. If the return value is nonzero and positive, but less than or equal to the size of the output buffer (as specified by the *bufsize* argument), then the encoding is complete. If the value returned by the function is greater than *bufsize*, the specified buffer was not large enough to contain the encoded data. In this case, the *encode_rect* function should be called again with a larger buffer. A value of 0 is returned if the function is unable to perform any encoding. This can happen if argument *bufsize* is specified as 0 or if the intersection of the rectangle to be encoded and the clipping rectangle is empty.

If the original image lies only partially within the current clipping window, only the portion of the image lying within the window is encoded. When the

encoded image is later restored by the *decode_rect* function, only the encoded portion of the image is restored. Relative to the enclosing rectangle, this portion of the restored image occupies the same position as in the original image. If the original image lies entirely outside the clipping window, the encoded image is empty.

Currently, the only encoding scheme supported by the function is run-length encoding. This is a simple but effective image-compression technique that 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 (the length of the *run*) before the next color transition. To illustrate, a run of 7 yellow pixels followed by a run of 5 red pixels could be stored as [7] [yellow] [5] [red]. As expected, the greatest amount of compression is achieved in the case of images that contain large regions of uniform color.

The compressed image format consists of a 20-byte header followed by the data representing the image in compressed form. The header structure is invariant across all encoding schemes and is defined as follows:

```
typedef struct {
    unsigned short magic;     /* magic number */
    unsigned long length;     /* length of data in bytes */
    unsigned short scheme;    /* encoding scheme */
    short width, height;      /* dimensions of image rectangle */
    short psize;              /* pixel size of image */
    short flags;              /* usage varies with scheme */
    unsigned long clipadj;    /* x–y clipping adjustments */
} ENCODED_RECT;
```

The fields of the ENCODED_RECT data structure above are used as follows:

magic A TIGA data structure identifier. The value for this data structure is 0x8101.

length The length of the entire compressed image in bytes, including the header. This value is useful for allocating memory for a data structure and for reading it from a disk.

scheme The type of encoding scheme that was used to encode the rectangle. Only one scheme is currently supported: *scheme* = 0 — run-length encoding.

width The width of the rectangle containing the original image.

height The height of the rectangle containing the original image.

psize The original pixel size of the encoded image. This value is 1, 2, 4, 8, 16, or 32.

flags Reserved for future enhancements. Bits in this field are currently set to 0.

clipadj Set to 0 except in the case in which the top left corner of the original image rectangle is located above or to the left of the clipping window. In this case, the *clipadj* field contains the concatenated x and y displacements of the top left corner of the

clipping window from the top left corner of the image. (The x displacement is in the 16 LSBs, and the y displacement in the 16 MSBs.) If the left edge of the window is to the right of the left edge of the image, the x displacement is set to the positive distance between these two edges; otherwise it is 0. If the top edge of the window is below the top edge of the image, the y displacement is set to the positive distance between these two edges; otherwise it is 0.

The encoded image immediately follows the *clipadj* field. This data is of variable length, and its format depends on the encoding scheme used to compress the image.

The run-length encoded image consists of a number of run-length encoded horizontal scan lines; the number of lines is given by the height entry in the ENCODED_RECT structure. Each line is encoded according to the following format:

[REPSIZ] [OPSIZ] [OPCODE] [DATA] [OPCODE] [DATA]...

The REPSIZ and OPSIZ fields, which appear at the start of each line, are defined as follows:

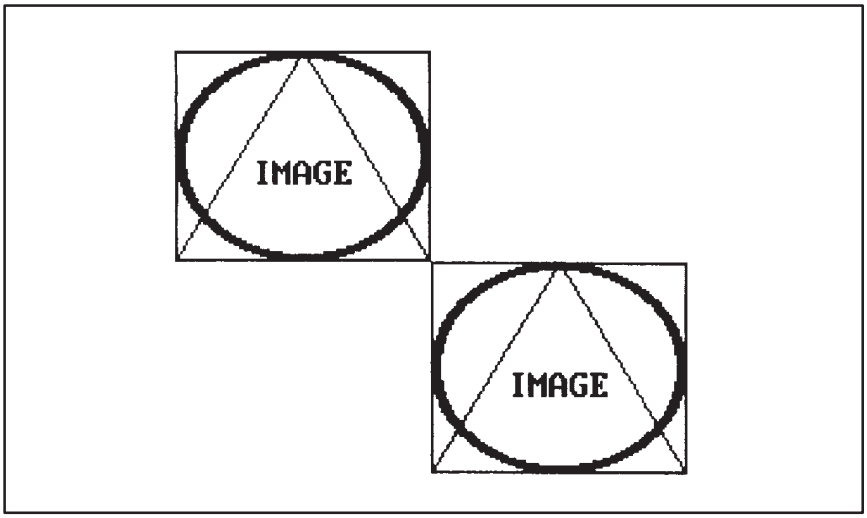
REPSIZ Bits 0–2 specify the size of the repeating data. Repeating data can be 1, 2, 4, 8, 16, or 32 bits in length. REPSIZ is the log to the base 2 of the data size (that is, 1 shifted left by the value of REPSIZ will give the size of the repeating data).

OPSIZ Bits 3–7 specify the length in bits of the OPCODE entry. This can be a value between 1 and 32 indicating the signed integer size of OPCODE. For example, if the value of OPSIZ is 8, then OPCODES are 8-bit signed integers. If OPSIZ is 3, then OPCODES are 3-bit signed integers. Beginning with bit 8, the remainder of the line consists of a variable number of [OPCODE] [DATA] sequences. If the opcode value is positive, it indicates a repeating sequence and will be followed by 1, 2, 4, 8, 16, or 32 bits worth of repeating data, as indicated by REPSIZ. If the opcode is negative, then it is followed by *n* pixels of absolute (unencoded) data, where *n* is the absolute value of the OPCODE, and the pixel size is specified in the PSIZE field of the ENCODED_RECT structure.

Within each line of the image, the absolute value of all the opcodes read equals the width of the encoded rectangle. This fact is utilized by the *decode_rect* function during decompression of the image.

Example Use the *encode_rect* function to capture a rectangular image from the screen. Verify that the image buffer used by the *encode_rect* function is large enough to contain the entire compressed image. Use the *decode_rect*

function to decompress the image to a different region of the screen to verify that the image was captured correctly by the *encode_rect* function.




```
#define  MAXSIZE      4096    /* max picture size in bytes */

static char picture[MAXSIZE];

main()
{
    short w, h, x, y, n;
    char *s;

    set_config(0, !0);
    clear_screen(0);

    /* Create an image on the screen. */
    w = 100;
    h = 80;
    x = 10;
    y = 10;
    frame_rect(w, h, x, y, 1, 1);
    frame_oval(w, h, x, y, 4, 3);
    draw_line(x+w/2, y, x, y+h-1);
    draw_line(x+w/2, y, x+w-1, y+h-1);
    s = "IMAGE";
    n = text_width(s);
    text_out(x+(w-n)/2, y+h/2, s);

    /* Compress image, and verify buffer doesn't overflow. */
    n = encode_rect(w, h, x, y, picture, sizeof(picture), 0);
    if (n > MAXSIZE) {
        text_out(x, y+h+20, "Image buffer too small!");
        exit(1);
    }

    /* Now decompress the image. */
    decode_rect(x+w, y+h, picture);
}
```

Syntax

```
typedef struct { short x, y; } POINT;

void fill_convex(n, vert)
short n;        /* vertex count */
POINT *vert;    /* vertex coordinates */
```

Description

The *fill_convex* function fills a convex polygon with a solid color. The polygon is specified by a list of points representing the polygon vertices in the order in which they are traversed in tracing the boundary of the polygon. The polygon is filled with the current foreground color.

Argument *n* specifies the number of vertices in the polygon, which is the same as the number of sides.

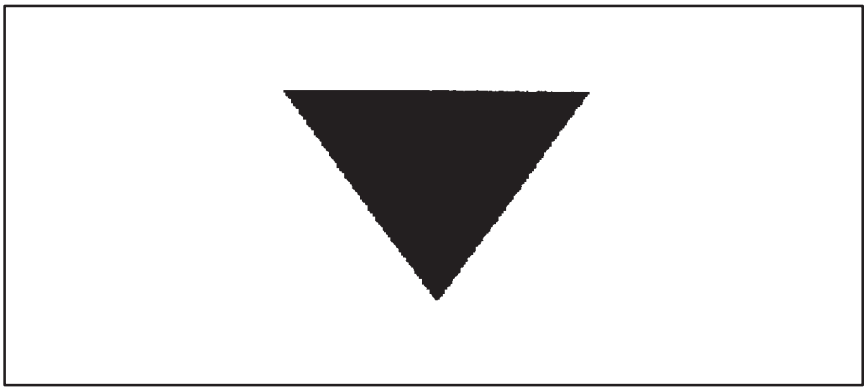
Argument *vert* is an array of integer x-y coordinates representing the polygon vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function assumes that an edge connects each adjacent pair of vertices in the array and also assumes that vertex *n*–1 is connected to vertex 0 by an edge. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

The *fill_convex* function is similar to the *fill_polygon* function but is specialized for rapid drawing of convex polygons. It also executes more rapidly and supports realtime applications such as animation. The function assumes that the polygon contains no concavities; if this requirement is violated, the polygon may be drawn incorrectly.

In order to conveniently support 3D applications, the *fill_convex* function automatically culls back faces. A polygon is drawn only if its front side is visible—that is, if it is facing toward the viewer. The direction in which the polygon is facing is determined by the order in which the vertices are listed in the *vert* array. If the vertices are specified in clockwise order, the polygon is assumed to be facing forward. If the vertices are specified in counterclockwise order, the polygon is assumed to face away from the viewer and is therefore not drawn.

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 facing) or counterclockwise (back facing) order. This test assumes the polygon contains no concavities. If the three vertices are collinear, the back-face test is performed again using the next three vertices, *n*–1, 0, and 1. The test repeats until three vertices are found that are not collinear. If all the vertices are collinear, the polygon is invisible.

Example Use the *fill_convex function* to fill a triangle. The three vertices are at coordinates (10, 10), (130, 10), and (70, 90).



```
#define  NVERTS    3   /* number of vertices */

typedef struct { short x, y; } POINT;

static POINT xy[NVERTS] =
{
    { 10, 10 }, { 130, 10 }, { 70, 90 }
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    fill_convex(NVERTS, xy);
}
```

Syntax

```
void fill_oval(w, h, xleft, ytop)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
```

Description

The *fill_oval* function fills an ellipse with a solid color. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which the ellipse is inscribed. The ellipse is filled with the current foreground color.

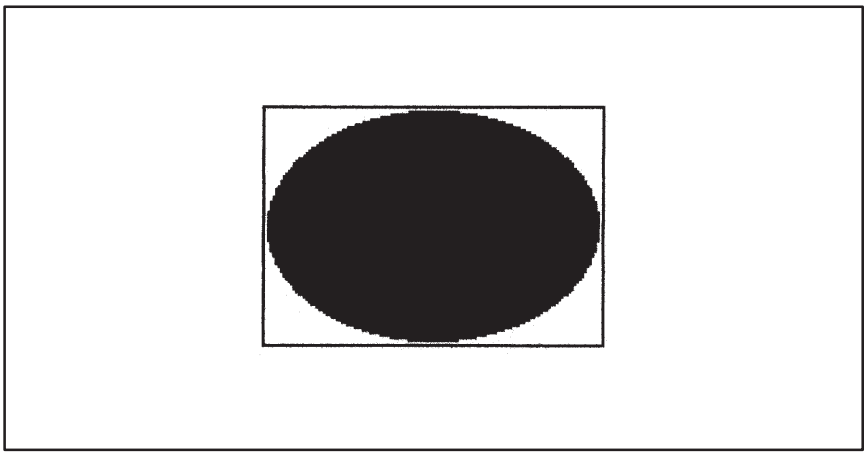
The four arguments specify the rectangle enclosing the ellipse:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

If the specified width or height is 0, nothing is drawn.

Example

Use the *fill_oval* function to draw an ellipse that is 130 pixels wide and 90 pixels high. Also, draw the outline of a rectangle that encloses the ellipse without touching it.



```
main()
{
    set_config(0, !0);
    clear_screen(0);
    fill_oval(130, 90, 10, 10);
    draw_rect(130+3, 90+3, 10–2, 10–2);
}
```

fill_piearc *Fill Pie Arc*

Syntax

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

Description

The *fill_piearc* function fills a pie-slice-shaped wedge with a solid color. The wedge is bounded by an arc and two straight edges. The two straight edges connect the end points of the arc with the center of the ellipse. The arc is taken from an ellipse in standard position, with its major and minor axes parallel to the coordinate axes. The ellipse is specified by the enclosing rectangle in which it is inscribed. The wedge is filled with the current foreground color.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

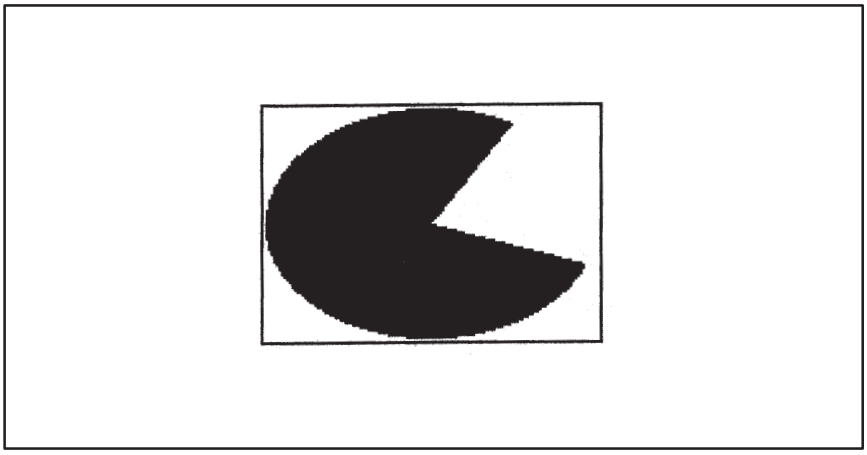
If the specified width or height is 0, nothing is drawn.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is filled.

Example Use the *fill_piearc* function to draw a pie chart corresponding to a slice of an ellipse from 21 degrees to 300 degrees. The ellipse is 130 pixels wide and 90 pixels high. Also, draw a rectangle that encloses the ellipse without touching it.



```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    fill_piearc(w, h, x, y, 21, 300-21);
    draw_rect(w+3, h+3, x-2, y-2);
}
```

fill_polygon *Fill Polygon*

Syntax

```
typedef struct { short x, y; } POINT;

void fill_polygon(n, vert)
short n;        /* vertex count */
POINT *vert;    /* vertex coordinates */
```

Description

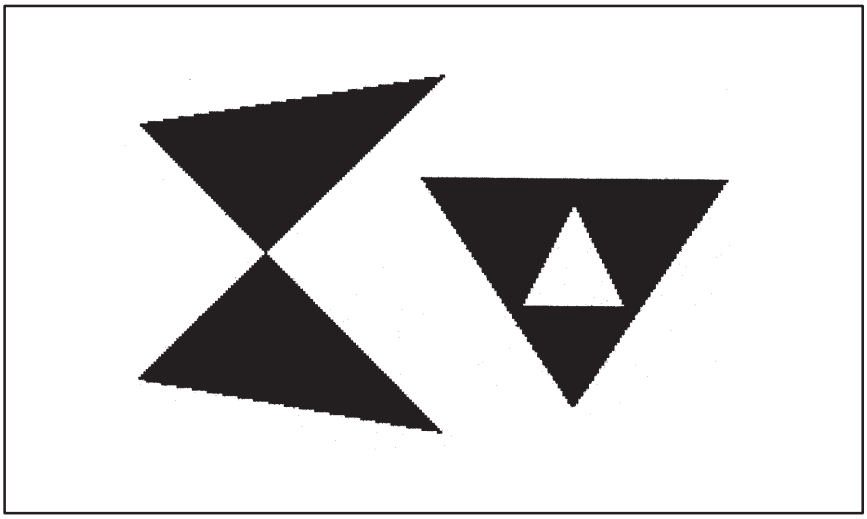
The *fill_polygon* function fills an arbitrarily shaped polygon with a solid color. The polygon is specified by a list of points representing the polygon vertices in the order in which they are traversed in tracing the boundary of the polygon. The interior of the polygon is determined according to the parity (or odd- even) rule. A pixel is considered to be part of the filled region representing the polygon if an infinite, arbitrarily oriented ray emanating from the center of the pixel crosses the boundary of the polygon an odd number of times. The polygon is filled with the current foreground color.

Argument *n* specifies the number of vertices in the polygon, which is the same as the number of sides.

Argument *vert* is an array of integer x-y coordinates representing the polygon vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function assumes that an edge connects each adjacent pair of vertices in the array and also assumes that vertex *n*–1 is connected to vertex 0 by an edge. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

No restrictions are placed on the shape of the polygons filled by this function. Edges may cross each other. Filled areas can contain holes (this is accomplished by connecting a hole to the outside edge of the polygon by an infinitely thin region of the polygon). Two or more filled regions can be disconnected from each other (or more precisely, be connected by infinitely thin regions of the polygon).

Example Use the *fill_polygon* function to fill a polygon that has a hole, two disconnected regions, and two edges that cross each other.



```
#define  NVERTS    14      /* number of vertices */

typedef struct { short x, y; } POINT;

static POINT xy[NVERTS] = {
    { 150, 170 }, {  30, 150 }, { 150,30 }, {  30,  50 },
    { 150, 170 }, { 140,  70 }, { 260,70 }, { 200, 160 },
    { 140,  70 }, { 200,  80 }, { 220, 120 }, { 180, 120 },
    { 200,  80 }, { 140,  70 },
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    fill_polygon(NVERTS, xy);
}
```

fill_rect *Fill Rectangle*

Syntax

```
void fill_rect(w, h, xleft, ytop)
short w, h;         /* rectangle width and height */
short xleft, ytop   /* top left corner */
```

Description

The *fill_rect* function fills a rectangle with a solid color. The rectangle is filled with the current foreground color.

The four arguments specify the rectangle:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

If the specified width or height is 0, nothing is drawn.

Example

Use the *fill_rect* function to fill a rectangle that is 130 pixels wide and 90 pixels high.

```
main()
{
    set_config(0, !0);
    clear_screen(0);
    fill_rect(130, 90, 10, 10);
}
```

Syntax

```
void frame_oval(w, h, xleft, ytop, dx, dy)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
short dx, dy;       /* frame thickness in x, y */
```

Description

The *frame_oval* function fills an ellipse-shaped frame with a solid color. The frame consists of a filled region between two concentric ellipses. The outer ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The frame thickness is specified separately for the x and y dimensions. The portion of the screen enclosed by the frame is not altered. The frame is filled with the current foreground color.

The first four arguments define the rectangle enclosing the outer edge of the elliptical frame:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle, and are defined relative to the drawing origin.

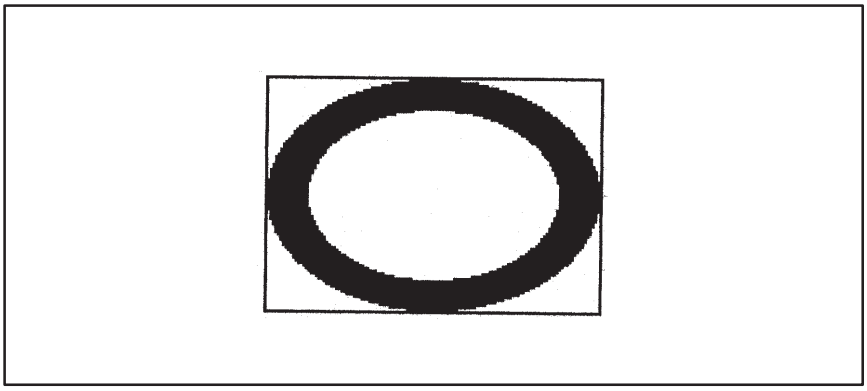
If the specified width or height is 0, nothing is drawn.

The last two arguments control the thickness of the frame:

- Arguments *dx* and *dy* specify the horizontal and vertical separation between the outer and inner ellipses, respectively.

Example

Use the *frame_oval* function to draw an elliptical frame. The outer border of the frame is an ellipse that is 130 pixels wide and 90 pixels high. The thickness of the frame in the x and y dimensions is 16 and 12, respectively. Also, outline the outer border of the frame with the *draw_rect* function.



```
main()
{
    short w, h, x, y, dx, dy;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    dx = 16;
    dy = 12;
    frame_oval(w, h, x, y, dx, dy);
    draw_rect(w+1, h+1, x-1, y-1);
}
```

Syntax

```
void frame_rect(w, h, xleft, ytop, dx, dy)
short w, h;         /* rectangle width and height */
short xleft, ytop;  /* top left corner */
short dx, dy        /* frame thickness in x, y */
```

Description

The *frame_rect* function fills a rectangular-shaped frame with a solid color. The frame consists of a filled region between two concentric rectangles. The outer edge of the frame is a rectangle specified in terms of its width, height, and position. The frame thickness is specified separately for the x and y dimensions. The portion of the screen enclosed by the frame is not altered. The frame is filled with the current foreground color.

The first four arguments define the rectangle enclosing the outer edge of the elliptical frame:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

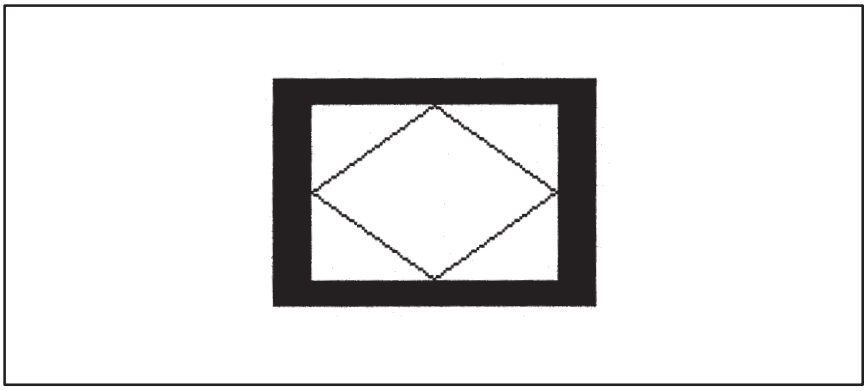
If the specified width or height is 0, nothing is drawn.

The last two arguments control the thickness of the frame:

- Arguments *dx* and *dy* specify the horizontal and vertical separation between the outer and inner rectangles, respectively.

Example

Use the *frame_rect* function to draw an rectangular frame. The outer border of the frame is a rectangle that is 127 pixels wide and 89 pixels high. The thickness of the frame in the x and y dimensions is 15 and 10, respectively. Also draw a diamond shape inside the frame with four calls to the *draw_line* function. The vertices of the diamond touch the center of each of the four inner edges of the frame.



```
main()
{
    short w, h, x, y, dx, dy;

    set_config(0, !0);
    clear_screen(0);
    w = 127;
    h = 89;
    x = 10;
    y = 10;
    dx = 15;
    dy = 10;
    frame_rect(w, h, x, y, dx, dy);
    draw_line(x+w/2, y+dy, x+w-dx-1, y+h/2);
    draw_line(x+w-dx-1, y+h/2, x+w/2, y+h-dy-1);
    draw_line(x+w/2, y+h-dy-1, x+dx, y+h/2);
    draw_line(x+dx, y+h/2, x+w/2, y+dy);
}
```

Syntax

```
typedef unsigned long PTR;    /*32-bit address*/
typedef struct
{
       PTR       addr;
       unsigned short pitch
       unsigned short xext, yext;
       unsigned short psize;
} BITMAP;
typedef struct
{
       unsigned long xyorigin;
       unsigned long pensize;
       BITMAP *srcbm, *dstbm;
       unsigned long stylemask;
} ENVIRONMENT;

void get_env(env)
ENVIRONMENT *env;   /* graphics environment pointer */
```

Description

The *get_env* function retrieves the current graphics environment information. Although the library contains other functions that manipulate individual environment parameters, this function retrieves the entire graphics environment as a single structure.

Argument *env* is a pointer to a structure of type ENVIRONMENT. The function copies the graphics environment information into the structure pointed to by this argument.

The fields of the ENVIRONMENT structure are defined as follows:

xyorigin	Current drawing origin in *y::x* format, set by call to *set_draw_origin* function
pensize	Current pen size in *y::x* format, set by call to *set_pen_size* function
patnaddr	Address of current pattern, set by call to *set_patnaddr* function
srcbm	Address of current source bit map structure, set by call to *set_srcbm* function
dstbm	Address of current destination bit map structure, set by call to *set_dstbm* function
stylemask	Line-style mask used by *styled_line* function.

In *y::x* format, 16-bit x and y components are concatenated to form a 32-bit value. The x component is followed by the y component. Note that the structure described above may change in subsequent revisions. To minimize the impact of such changes, write application programs to refer to the elements of the structure symbolically by their field names, rather than as offsets from the start of the structure. The include files provided with the library will be updated in future revisions to track any such changes in data structure definitions.

Example Use the *get_env* function to verify the initial state of the graphics environment parameters immediately following a call to the *set_config* function. Use the *text_out* function to print the parameter values on the screen. This example includes the C header file `gsptypes.h`, which defines the ENVIRONMENT and FONTINFO structures.

```
INITIAL GRAPHICS ENVIRONMENT:
x origin = 0
y origin = 0
pen width = 1
pen height = 1
source bitmap = 0
destination bitmap = 0
line-style pattern = -1
```

```
#include <gsptypes.h>   /* define ENVIRONMENT and FONTINFO */

main()
{
    ENVIRONMENT env;
    FONTINFO fontinfo;
    char c[80];
    short val;
    int n, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    get_fontinfo(0, &fontinfo);
    h = fontinfo.charhigh;
    x = y = 10;
    get_env(&env);     /* get graphics environment */

    text_out(x, y, "INITIAL GRAPHICS ENVIRONMENT:");

    strcpy(c, "x origin = ");
    n = strlen(c);
    ltoa(val = env.xyorigin, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "y origin = ");
    n = strlen(c);
    ltoa(env.xyorigin>>16, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "pen width = ");
    n = strlen(c);
    ltoa(val = env.pensize, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "pen height = ");
    n = strlen(c);
    ltoa(env.pensize>>16, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "source bit map = ");
    n = strlen(c);
    ltoa(env.srcbm, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "destination bit map = ");
    n = strlen(c);
    ltoa(env.dstbm, &c[n]);
    text_out(x, y += h, c);

    strcpy(c, "line-style pattern = ");
    n = strlen(c);
    ltoa(env.stylemask, &c[n]);
    text_out(x, y += h, c);
}
```

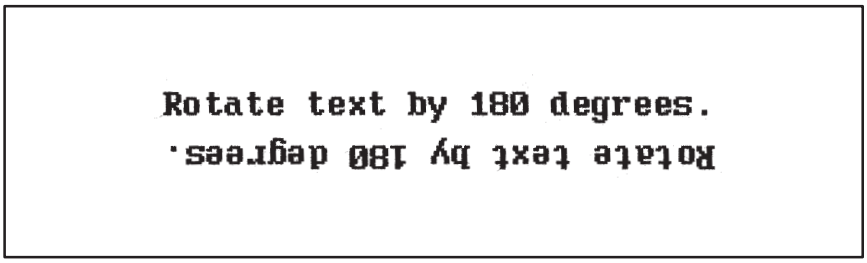
get_pixel *Get Pixel*

Syntax

```
unsigned long get_pixel(x, y)
short x, y;      /* pixel coordinates */
```

Description The *get_pixel* function returns the value of the pixel at x-y coordinates (*x*, *y*) on the screen. The coordinates are defined relative to the drawing origin. Given a pixel size of *n* bits, the pixel is contained in the *n* LSBs of the return value; the 32–*n* MSBs are set to 0.

Example Use the *get_pixel* function to rotate a text image on the screen by 180 degrees. This example includes the C header file `gsptypes.h`, which defines the FONTINFO structure.




```
#include <gsptypes.h>    /* defines FONTINFO structure */

main()
{
    FONTINFO fontinfo;
    short xs, ys, xd, yd, w, h;
    long val;
    char *s;

    set_config(0, !0);
    clear_screen(0);
    s = "Rotate text by 180 degrees.";
    get_fontinfo(0, &fontinfo);
    w = text_width(s);
    h = fontinfo.charhigh;
    xs = ys = 0;
    text_out(xs, ys, s);
    for (yd = 2*h+1; ys <= h; ys++, yd--)
        for (xs = 0, xd = w-1; xs <= w; xs++, xd--) {
            val = get_pixel(xs, ys);
            put_pixel(val, xd, yd);
        }
}
```

Syntax

```
short get_textattr(pcontrol, count, val)
char *pcontrol;  /* control string */
short count;     /* val array length */
short *val;      /* array of attribute values */
```

Description

The *get_textattr* function retrieves the text rendering attributes. The three text attributes currently supported are text alignment, additional intercharacter spacing, and intercharacter gaps.

Argument *pcontrol* is a control string specifying the attributes (one or more) to be retrieved. Argument *count* is the number of attributes designated in the *pcontrol* string and is also the number of attributes stored in the *val* array. Argument *val* is the array into which the designated attributes are stored. The attribute values are stored into the consecutive elements of the *val* array, beginning with *val*[0], in the order in which they appear in the *pcontrol* string.

The function returns a value indicating the number of attributes actually loaded into the *val* array.

The following attributes are currently supported:

Symbol	Attribute Description	Option Value
%a	alignment	0 = top left, 1 = base line
%e	additional intercharacter spacing	16-bit signed integer
%f	intercharacter gaps	0 = leave gaps, 1 = fill gaps

Example

Use the *get_textattr* function to verify the initial state of the text attributes immediately following a call to the *set_config* function. Use the *text_out* function to print the attribute values on the screen. This example includes the C header file `gsptypes.h`, which defines the FONTINFO structure.

```
#include <gsptypes.h>   /* define FONTINFO structure */

main()
{
    ENVIRONMENT env;
    FONTINFO fontinfo;
    char c[80];
    short val[3];
    int n, h, x, y;

    set_config(0, !0);
    clear_screen(-1);
    get_fontinfo(0, &fontinfo);
    h = fontinfo.charhigh;
    x = y = 10;
    get_textattr("%a%e%f", 3, val);  /* get text attributes */

    text_out(x, y, "DEFAULT TEXT ATTRIBUTES:");
    y += h;

    strcpy(c, "text alignment = ");
    n = strlen(c);
    ltoa(val[0], &c[n]);
    text_out(x, y, c);
    y += h;

    strcpy(c, "extra interchar spacing = ");
    n = strlen(c);
    ltoa(val[1], &c[n]);
    text_out(x, y, c);
    y += h;

    strcpy(c, "interchar gaps = ");
    n = strlen(c);
    ltoa(val[1], &c[n]);
    text_out(x, y, c);
}
```

Syntax

```
short in_font(start_code, end_code)
short start_code;       /* starting character code */
short end_code;         /* ending character code */
```

Description

The *in_font* function returns a value indicating whether the current font defines all the characters within a specified range of ASCII codes.

The two arguments specify the range of characters:

- Argument *start_code* specifies the ASCII code at the start of the range. (This is the first character included in the range.)
- Argument *end_code* specifies the ASCII code at the end of the range. (This is the last character included in the range.)

The value of *start_code* should be less than or equal to the value of *end_code*. Valid arguments are restricted to the range 1 to 255.

The value returned by the function is 0 if the current font defines all characters in the range specified by the arguments. Otherwise, the return value is the ASCII code of the first character (lowest ASCII code) in the specified range that is undefined in the current font.

Example

Use the *in_font* function to determine whether the system font defines all characters from ASCII code 32 to ASCII code 126. Use the *text_out* function to print the result of the test on the screen.

```
main()
{
    int n;
    unsigned char v, c[80];

    set_config(0, 1);
    clear_screen(-1);
    if (v = in_font(' ', '~')) {
        n = strlen(strcpy(c, "ASCII character code "));
        n += ltoa(v, &c[n]);
        strcpy(&c[n], " is undefined.");
        text_out(10, 10, c);
    } else
        text_out(10, 10, "Characters ' ' to '~' are defined.");
}
```

Syntax

```
short install_font(pfont)
FONT *pfont;     /* font structure pointer */
```

Description

The *install_font* function installs a font in the font table and returns an identifier (ID) of type *short*. The ID can be used to refer to the font in subsequent text operations.

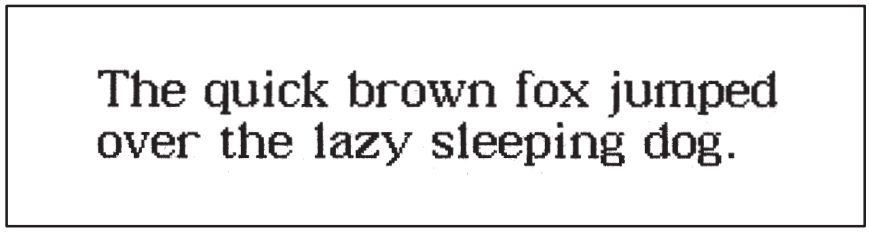
Argument *pfont* is a pointer to a structure of type FONT. (The FONT structure is described in Appendix A.) The *install_font* function merely adds the address of the font to the font table. It does not select the font.

The ID returned is nonzero if the installation was successful. If unsuccessful, 0 is returned.

The maximum size of the font table is a constant that can vary from system to system. In all systems, the font table will be large enough to contain at least 16 installed fonts (in addition to the permanently installed system font). Once the font table is full, an attempt to install an additional font will be ignored, and a value of 0 will be returned.

Example

Use the *install_font* function to install a proportionally spaced font. Use the *text_out* function to print a sample of the font on the screen. This example program includes the `gsptypes.h` file, which defines the FONT and FONTINFO structures. The TI Roman font size 20 must be linked with the program.



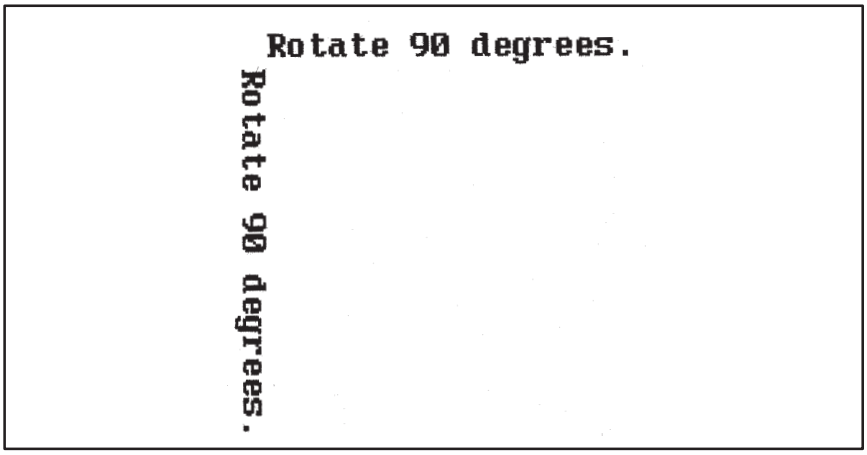

```
#include <gsptypes.h>  /* defines FONT and FONTINFO structures */

extern FONT ti_rom20;  /* proportionally-spaced font */

main()
{
    FONTINFO fontinfo;
    short x, y, id;

    set_config(0, !0);
    clear_screen(0);
    id = install_font(&ti_rom20);  /* install the font */
    get_fontinfo(id, &fontinfo);
    select_font(id);
    x = y = 10;
    text_out(x, y, "The quick brown fox jumped");
    y += fontinfo.charhigh;
    text_out(x, y, "over the lazy sleeping dog.");
}
```

Syntax

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

Description The *move_pixel* function copies a pixel from one screen location to another. Arguments *xs* and *ys* are the x and y coordinates of the source pixel. Arguments *xd* and *yd* are the x and y coordinates of the destination pixel. Coordinates are defined relative to the drawing origin.

Example Use the *move_pixel* function to rotate text image on screen by 90 degrees. This example includes the C header file `gsptypes.h`, which defines the FONTINFO structure.




```
#include <gsptypes.h>    /* define FONTINFO structure */

main()
{
    FONTINFO fontinfo;
    short xs, ys, xd, yd, w, h;
    char *s;

    set_config(0, !0);
    clear_screen(0);
    s = "Rotate 90 degrees.";
    get_fontinfo(0, &fontinfo);
    w = text_width(s);
    h = fontinfo.charhigh;
    xs = h;
    ys = 0;
    text_out(xs, ys, s);
    for (xd = yd = h; ys < h; ys++, xd = h-ys, yd = h)
        for (xs = h; xs < w+h; xs++, yd++)
            move_pixel(xs, ys, xd, yd);
}
```

patnfill_convex *Fill Convex Polygon with Pattern*

Syntax

```
typedef struct { short x, y; } POINT;

void patnfill_convex(n, vert)
short n;        /* vertex count */
POINT *vert;    /* vertex coordinates */
```

Description

The *patnfill_convex* function fills a convex polygon with the current area-fill pattern. The polygon is specified by a list of points representing the polygon vertices in the order in which they are traversed in tracing the boundary of the polygon.

Argument *n* specifies the number of vertices in the polygon, which is the same as the number of sides.

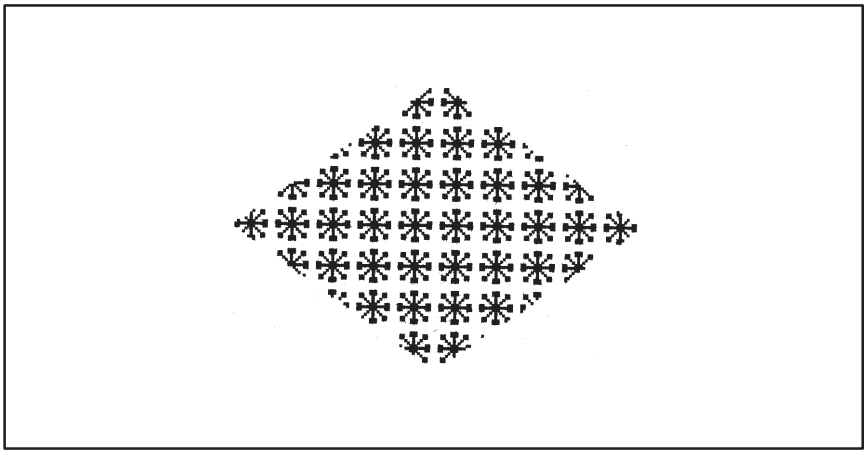
Argument *vert* is an array of integer x-y coordinates representing the polygon vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function assumes that an edge connects each adjacent pair of vertices in the array and that an edge connects vertex *n*–1 to vertex 0. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

The *patnfill_convex* function is similar to the *patnfill_polygon* function but is specialized for rapid drawing of convex polygons. It also executes more rapidly and supports realtime applications, such as animation. The function assumes that the polygon contains no concavities; if this requirement is violated, the polygon may be drawn incorrectly.

In order to conveniently support 3D applications, the *patnfill_convex* function automatically culls back faces. A polygon is drawn only if its front side is visible—that is, if it is facing toward the viewer. The direction in which the polygon is facing is determined by the order in which the vertices are listed in the *vert* array. If the vertices are specified in clockwise order, the polygon is assumed to be facing forward. If the vertices are specified in counterclockwise order, the polygon is assumed to face away from the viewer and is therefore not drawn.

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 facing) or counterclockwise (back facing) order. This test assumes the polygon contains no concavities. If the three vertices are collinear, 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 collinear. If all the vertices are collinear, the polygon is invisible.

Example Use the *patnfill_convex* function to fill a quadrilateral with a pattern. The four vertices are located at (96, 16), (176, 72), (96, 128), and (16, 72). This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h> /* define PATTERN structure */
#define  NVERTS    4  /* number of vertices in quadrilateral */

typedef struct { short x, y; } POINT;

static short snowflake[16] =
{
       0x0000,  0x01C0,  0x19CC,  0x188C,  0x0490,  0x02A0,  0x31C6,
0x3FFE,
       0x31C6,  0x02A0,  0x0490,  0x188C,  0x19CC,  0x01C0,  0x0000,
0x0000,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)snowflake };
static POINT xy[NVERTS] =
{
    { 96, 16 }, { 176, 72 }, { 96, 128 }, { 16, 72 },
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);
    patnfill_convex(NVERTS, xy);
}
```

Syntax

```
void patnfill_oval(w, h, xleft, ytop)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
```

Description

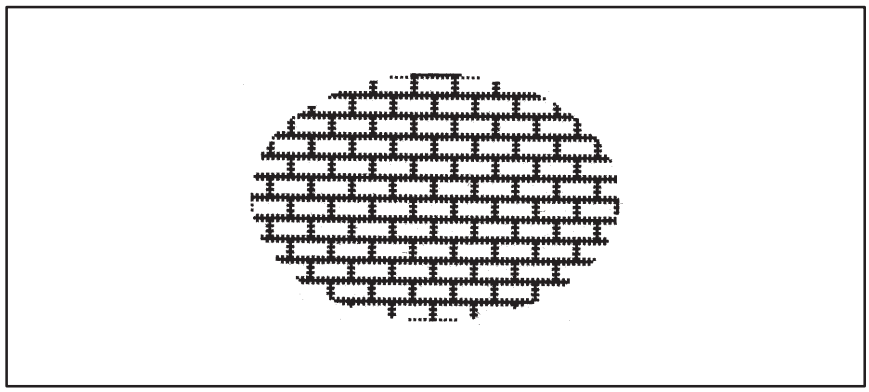
The *patnfill_oval* function fills an ellipse with the current area-fill pattern. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which the ellipse is inscribed.

The four arguments specify the rectangle enclosing the ellipse:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

Example

Use the *patnfill_oval* function to fill an ellipse that is 144 pixels wide by 96 pixels high with a 16-by-16 area-fill pattern. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>   /* define PATTERN structure */

typedef struct { unsigned short row[16]; } PATNBITS;

static PATNBITS patnbits =     /* brick pattern */
{
       0xFFFF,  0xD555,  0x8000,  0xC001,  0x8000,  0xC001,  0x8000,
0xD555,
       0xFFFF,  0x55D5,  0x0080,  0x01C0,  0x0080,  0x01C0,  0x0080,
0x55D5,
};
static PATTERN current_patn = { 16, 16, 1, (PTR)&patnbits };

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&current_patn);
    patnfill_oval(144, 96, 16, 16);
}
```

Syntax

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

Description

The *patnfill_piearc* function fills a pie-slice-shaped wedge with an area-fill pattern. The wedge is bounded by an arc and two straight edges. The two straight edges connect the end points of the arc with the center of the ellipse. The arc is taken from an ellipse in standard position, with its major and minor axes parallel to the coordinate axes. The ellipse is specified by the enclosing rectangle in which it is inscribed. The wedge is filled with the current area-fill pattern.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

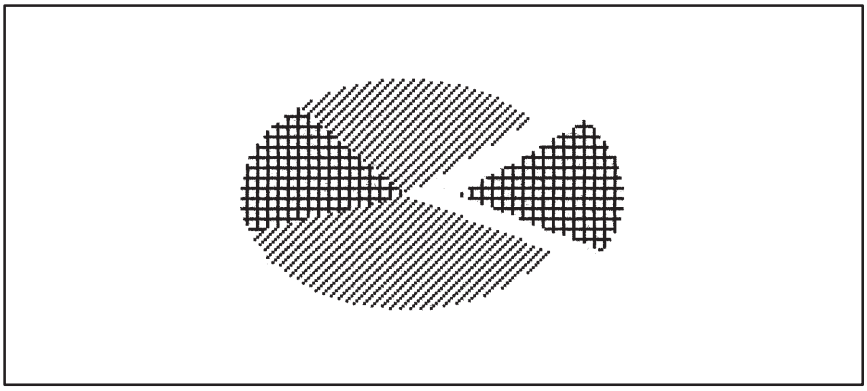
The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is filled.

Example

Use the *patnfill_piearc* function to draw a pie chart 144 pixels wide by 96 pixels high with a 16-by-16 area-fill pattern. The pie chart contains four pie slices. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>     /* define PATTERN structure */
#define  W     130        /* width of pie chart */
#define  H      90        /* height of pie chart */
#define  X      10        /* left edge of pie chart */
#define  Y      10        /* top edge of pie chart */

typedef struct { unsigned short row[16]; } PATNBITS;

/* Two contrasting area-fill patterns */
static PATNBITS patnbits[2] =
{
 { 0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
      0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111},
 { 0xFFFF, 0x1111, 0x1111, 0x1111, 0xFFFF, 0x1111, 0x1111, 0x1111,
      0xFFFF,  0x1111,  0x1111,  0x1111,  0xFFFF,  0x1111,  0x1111,
0x1111},
};

main()
{
    static PATTERN piepatn = { 16, 16, 1, (PTR)0 };

    set_config(0, !0);
    clear_screen(0);
    piepatn.data = (PTR)&patnbits[0];
    set_patn(&piepatn);
    patnfill_piearc(W, H, X, Y, 30, 160-30);      /* slice #1 */
    piepatn.data = (PTR)&patnbits[1];
    set_patn(&piepatn);
    patnfill_piearc(W, H, X, Y, 160, 230-160);    /* slice #2 */
    piepatn.data = (PTR)&patnbits[0];
    set_patn((PTR)&piepatn);
    patnfill_piearc(W, H, X, Y, 230, 320-230);    /* slice #3 */
    piepatn.data = (PTR)&patnbits[1];
    set_patn(&piepatn);
    patnfill_piearc(W, H, X+20, Y, 320, 390-320); /* slice #4 */
}
```

Syntax

```
typedef struct { short x, y; } POINT;

void patnfill_polygon(n, vert)
short n;        /* vertex count */
POINT *vert;    /* vertex coordinates */
```

Description

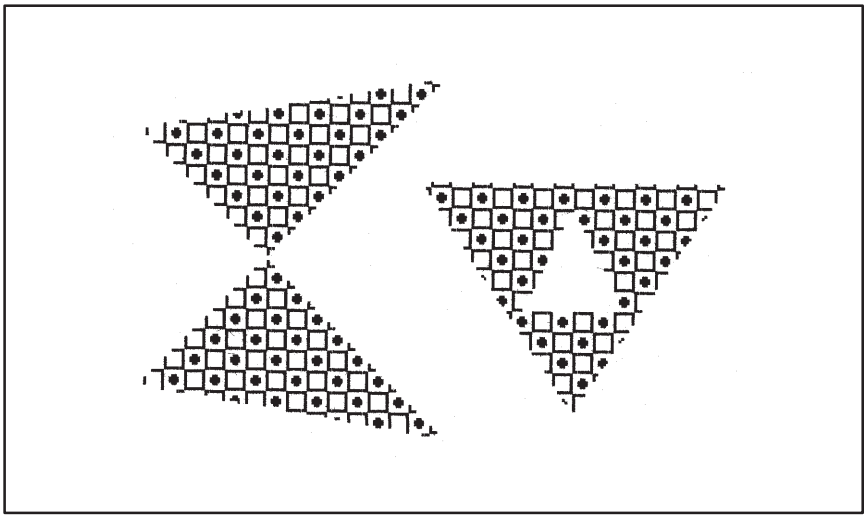
The *patnfill_polygon* function fills an arbitrarily shaped polygon with the current area-fill pattern. The polygon is specified by a list of points representing the polygon vertices in the order in which they are traversed in tracing the boundary of the polygon. The interior of the polygon is determined according to the parity (or odd-even) rule. A pixel is considered to be part of the filled region representing the polygon if an infinite, arbitrarily oriented ray emanating from the center of the pixel crosses the boundary of the polygon an odd number of times.

Argument *n* specifies the number of vertices in the polygon, which is the same as the number of sides.

Argument *vert* is an array of integer x–y coordinates representing the polygon vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function assumes that an edge connects each adjacent pair of vertices in the array and also assumes that an edge connects vertex *n*–1 to vertex 0. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

No restrictions are placed on the shape of the polygons filled by this function. Edges may cross each other. Filled areas can contain holes (this is accomplished by connecting a hole to the outside edge of the polygon by an infinitely thin region of the polygon). Two or more filled regions can be disconnected from each other (or more precisely, be connected by infinitely thin regions of the polygon).

Example Use the *patnfill_polygon* function to fill a polygon that has a hole, two disconnected regions, and two edges that cross each other. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>   /* define PATTERN structure */
#define  NVERTS    14   /* 14 vertices in polygon */

typedef struct { short x, y; } POINT;

static short patnbits[16] =     /* squares pattern */
{
       0x00FF,  0x0081,  0x1881,  0x3C81,  0x3C81,  0x1881,  0x0081,
0x00FF,
       0xFF00,  0x8100,  0x8118,  0x813C,  0x813C,  0x8118,  0x8100,
0xFF00,
};
static PATTERN current_patn = { 16, 16, 1, (PTR)patnbits };
static POINT xy[NVERTS] = {
    { 150, 170 }, {  30, 150 }, { 150,30 }, {  30,  50 },
    { 150, 170 }, { 140,  70 }, { 260,70 }, { 200, 160 },
    { 140,  70 }, { 200,  80 }, { 220, 120 }, { 180, 120 },
    { 200,  80 }, { 140,  70 },
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&current_patn);
    patnfill_polygon(NVERTS, xy);
}
```

Syntax

```
void patnfill_rect(w, h, xleft, ytop)
short w, h;         /* rectangle width and height */
short xleft, ytop   /* top left corner */
```

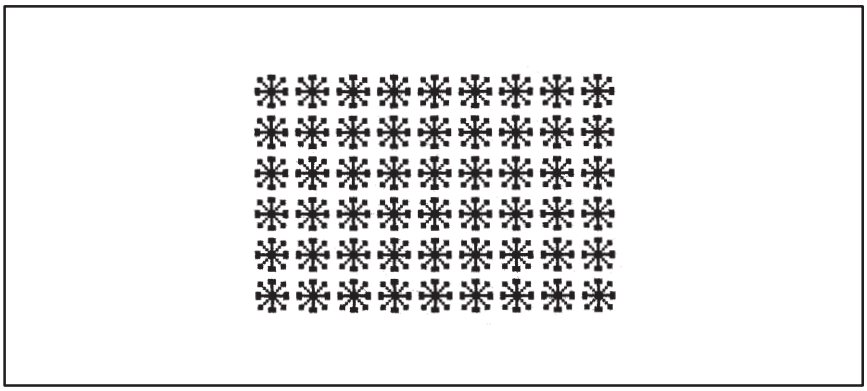
Description The *patnfill_rect* function fills a rectangle with the current area-fill pattern.

The four arguments specify the rectangle:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

If the specified width or height is 0, nothing is drawn.

Example Use the *patnfill_rect* function to fill a rectangle that is 144 pixels wide by 96 pixels high with a 16-by-16 area-fill pattern. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */
typedef struct { unsigned short row[16]; } PATNBITS;
static PATNBITS patnbits =
{
       0x0000,  0x01C0,  0x19CC,  0x188C,  0x0490,  0x02A0,  0x31C6,
0x3FFE,
       0x31C6,  0x02A0,  0x0490,  0x188C,  0x19CC,  0x01C0,  0x0000,
0x0000,
};
static PATTERN current_patn = { 16, 16, 1, (PTR)&patnbits };
main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&current_patn);
    patnfill_rect(144, 96, 16, 16);
}
```

patnframe_oval *Fill Oval Frame with Pattern*

Syntax

```
void patnframe_oval(w, h, xleft, ytop, dx, dy)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
short dx, dy;       /* frame thickness in x, y */
```

Description

The *patnframe_oval* function fills an ellipse-shaped frame with the current area-fill pattern. The frame consists of a filled region between two concentric ellipses. The outer ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The frame thickness is specified separately for the x and y dimensions. The portion of the screen enclosed by the frame is not altered.

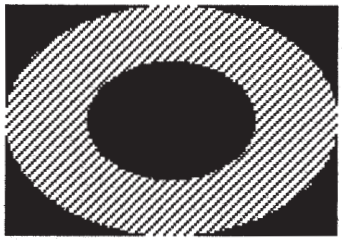
The first four arguments define the rectangle enclosing the outer edge of the elliptical frame:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments control the thickness of the frame:

- Arguments *dx* and *dy* specify the horizontal and vertical separation, respectively, between the outer and inner ellipses.

Example Use the *patnframe_oval* function to draw an elliptical frame rendered with an area-fill pattern. The elliptical frame is superimposed upon a filled rectangle. Both the rectangle and the outer boundary of the elliptical frame are of width 130 and height 90. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */

static short fillpatn[] = {     /* 16-by-16 area-fill pattern */
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
    0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111
};
static PATTERN framepatn = { 16, 16, 1, (PTR)fillpatn };

main()
{
    short w, h, x, y, dx, dy;

    set_config(0, !0);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    dx = w/4;
    dy = h/4;
    set_patn(&framepatn);
    fill_rect(w, h, x, y);
    patnframe_oval(w, h, x, y, dx, dy);
}
```

Syntax

```
void patnframe_rect(w, h, xleft, ytop, dx, dy)
short w, h;         /* rectangle width and height */
short xleft, ytop;  /* top left corner */
short dx, dy        /* frame thickness in x, y */
```

Description

The *patnframe_rect* function fills a rectangle-shaped frame with the current area-fill pattern. The frame consists of a filled region between two concentric rectangles. The outer edge of the frame is a rectangle specified in terms of its width, height, and position. The frame thickness is specified separately for the x and y dimensions. The portion of the screen enclosed by the frame is not altered.

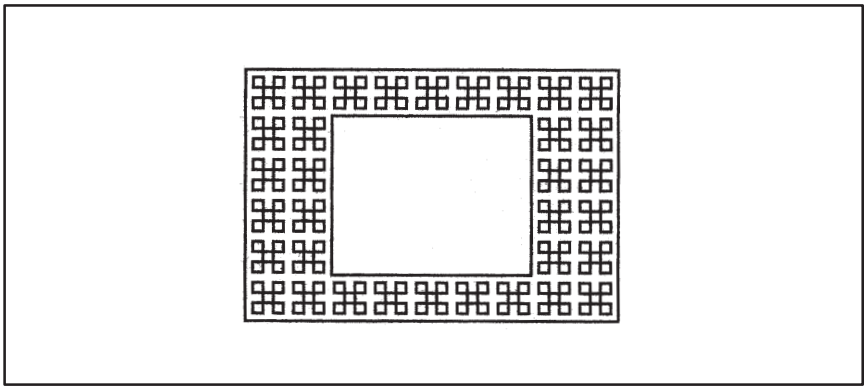
The first four arguments define the rectangle enclosing the outer edge of the elliptical frame:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments control the thickness of the frame:

- Arguments *dx* and *dy* specify the horizontal and vertical separation, respectively, between the outer and inner rectangles.

Example Use the *patnframe_rect* function to draw a rectangular frame rendered with a 16-by-16 area-fill pattern. Also, outline the outer and inner borders of the frame with the *draw_rect* function. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */

static short fillpatn[] = {     /* 16-by-16 area-fill pattern */
       0x0000,  0x0000,  0x7C7C,  0x4444,  0x4444,  0x4444,  0x7FFC,
0x0440,
       0x0440,  0x0440,  0x7FFC,  0x4444,  0x4444,  0x4444,  0x7C7C,
0x0000,
};

main()
{
    static PATTERN framepatn = { 16, 16, 1, (PTR)fillpatn };
    short w, h, x, y, dx, dy;

    set_config(0, !0);
    clear_screen(0);
    w = 144;
    h = 96;
    x = 16;
    y = 16;
    dx = 32;
    dy = 16;
    set_patn(&framepatn);
    patnframe_rect(w, h, x, y, dx, dy);
    draw_rect(w+2, h+2, x-1, y-1);
    draw_rect(w-2*dx-2, h-2*dy-2, x+dx+1, y+dy+1);
}
```

patnpen_line *Draw Line with Pen and Pattern*

Syntax

```
void patnpen_line(x1, y1, x2, y2)
short x1, y1;    /* start coordinates */
short x2, y2;    /* end coordinates */
```

Description

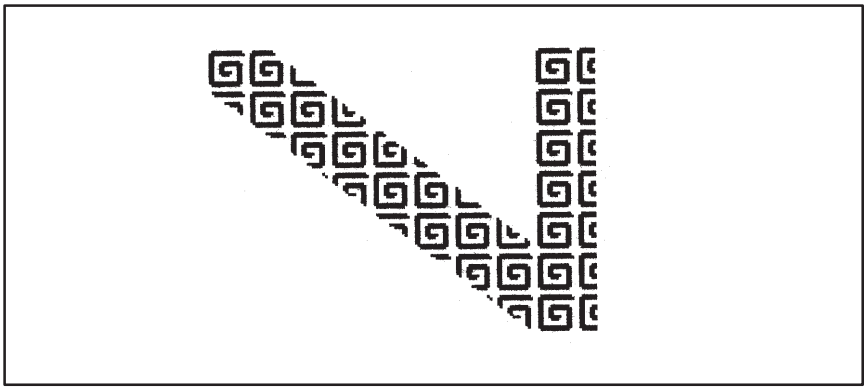
The *patnpen_line* function draws a line with a pen and an area-fill pattern. The thickness of the line is determined by the width and height of the rectangular drawing pen. The area covered by the pen to represent the line is filled with the current area-fill pattern.

Arguments *x1* and *y1* specify the starting x and y coordinates of the line. Arguments *x2* and *y2* specify the ending x and y coordinates of the line.

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 with its top left corner touching the line. The area covered by the pen as it traverses the line from start to end is filled with a pattern.

Example

Use the *patnpen_line* function to draw two lines. The first line goes from (16, 16) to (144, 112), and the second line goes from (144, 112) to (144, 16). Use the *set_pensize* function to set the pen dimensions to 24-by-16. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */

static short spiral[16] =
{                       /* 16x16 area-fill pattern */
       0x0000,  0x3FFC,  0x7FFE,  0x0006,  0x0006,  0x1FC6,  0x3FE6,
0x3066,
       0x3066,  0x33E6,  0x31C6,  0x3006,  0x3006,  0x3FFE,  0x1FFC,
0x0000,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)spiral };

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_pensize(24, 16);
    set_patn(&fillpatn);
    patnpen_line(16, 16, 144, 112);
    patnpen_line(144, 112, 144, 16);
}
```

Syntax

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

Description

The *patnpen_ovalarc* function draws an arc of an ellipse with a pen and an area-fill pattern. The ellipse from which the arc is taken is in standard position, with the major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The area swept out by the pen as it traverses the arc is filled with the current area-fill pattern. The thickness of the arc is determined by the width and height of the rectangular drawing pen.

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_ovalarc* function, the pen is located with its top left corner touching the arc. The area covered by the pen as it traverses the arc from start to end is filled with a pattern.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

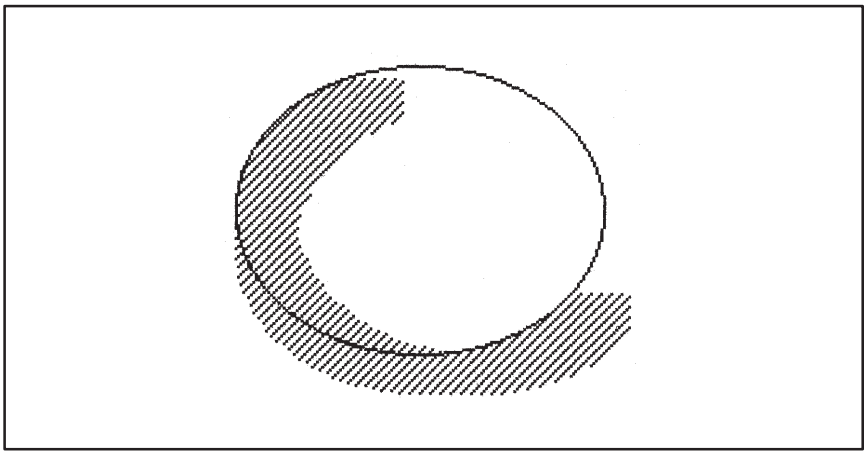
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *patnpen_ovalarc* function to draw an arc taken from an ellipse. Set the pen dimensions to 24-by-16, and set the width and height of the ellipse to 144 and 112, respectively. Use the *draw_oval* function to superimpose a thin ellipse having the same width and height on the path taken by the pen in tracing the arc. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */

static short stripes[16] =
{
                          /* 16x16 area-fill pattern */
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)stripes };

main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    set_pensize(24, 16);
    set_patn(&fillpatn);
    w = 144;
    h = 112;
    x = 16;
    y = 16;
    patnpen_ovalarc(w, h, x, y, 35, 255-45);
    draw_oval(w, h, x, y);
}
```

Syntax

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

Description

The *patnpen_piearc* function draws a pie-slice-shaped wedge from an ellipse with a pen and an area-fill pattern. 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 from which the arc is taken is in standard position, with the major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The area swept out by the pen as it traverses the perimeter of the wedge is filled with the current area-fill pattern. The thickness of the arc and of two lines drawn to represent the wedge is determined by the width and height of the rectangular drawing pen.

The pen is a rectangle whose width and height can be modified by means of the *set_pensize* function. As the pen traverses the arc from start to end, the pen is located with its top left corner touching the arc. The two lines connecting the arc start and end points with the center of the ellipse are drawn in similar fashion, with the top left corner of the pen touching each line as it traverses the line from start to end.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

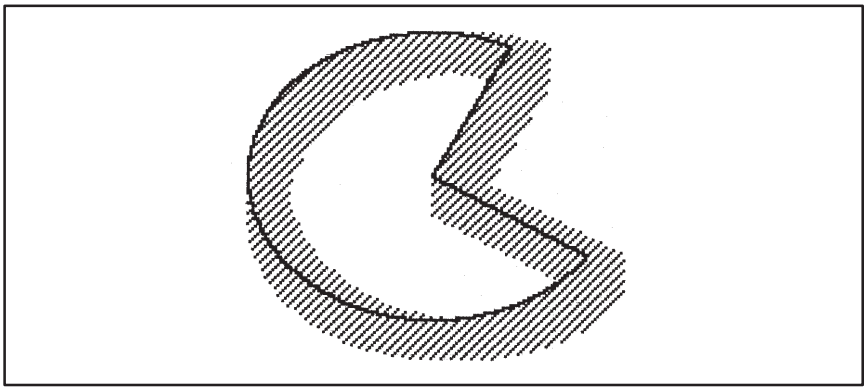
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *patnpen_piearc* function to draw an arc taken from an ellipse. Set the pen dimensions to 16-by-16. Use the *pen_piearc* function to superimpose a “thin” pie slice on the path taken by the pen in tracing the “fat” pie slice. Both the fat and thin slices are taken from the same ellipse, which has width 144 and height 112. The arc extends from 33 degrees to 295 degrees. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */

static short stripes[16] =
{
                        /* 16x16 area-fill pattern */
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)stripes };

main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);
    w = 144;
    h = 112;
    x = 16;
    y = 16;
    set_pensize(16, 16);
    patnpen_piearc(w, h, x, y, 33, 295-33);
    set_pensize(1, 1);
    pen_piearc(w, h, x, y, 33, 295-33);
}
```

patnpen_point *Draw Point with Pen and Pattern*

Syntax

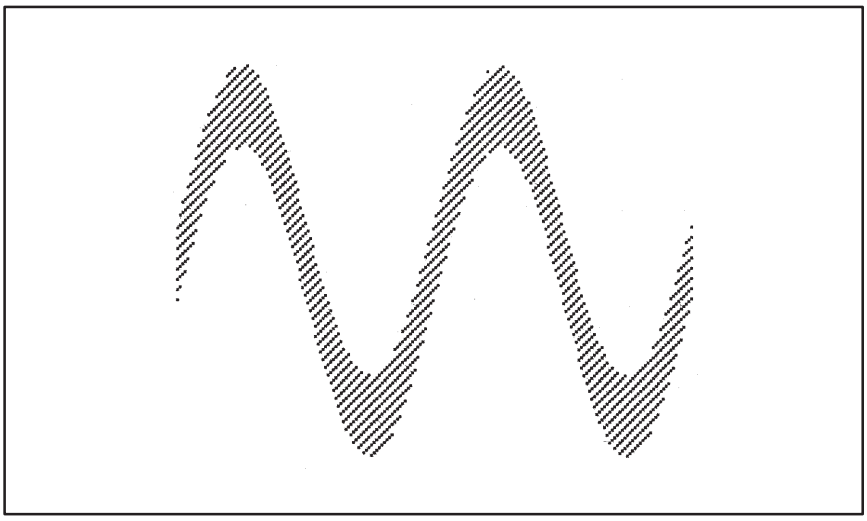
```
void patnpen_point(x, y)
short x, y;      /* pen coordinates */
```

Description

The *patnpen_point* function draws a point with a pen and an area-fill pattern. Arguments *x* and *y* specify where the top left corner of the rectangular drawing pen is positioned. The resulting figure is a rectangle the width and height of the pen and filled with the current area-fill pattern.

Example

Use the *patnpen_point* function to draw a sine wave of amplitude 60. Each point on the wave is separated from the next by an angular increment of approximately 1/16 radian. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>     /* define PATTERN structure */
#define  FOURPI   823550   /* fixed-point 4*PI */
#define  HALF     32768    /* fixed-point 1/2 */
#define  AMPL     60       /* sine wave amplitude */
#define  N        4        /* angular increment = 1/2**N radians */

typedef long FIX;          /* fixed-pt with 16-bit fraction */

static short stripes[16] =
{
                          /* 16x16 area-fill pattern */
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
       0x8888,  0x4444,  0x2222,  0x1111,  0x8888,  0x4444,  0x2222,
0x1111,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)stripes };

main()
{
    int i;
    short x, y;
    FIX u, v;

    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);
    set_pensize(1, 32);
    set_draw_origin(10, 10+AMPL);

    u = AMPL << 16;           /* convert to fixed-pt */
    v = 0;
    for (i = (FOURPI << N) >> 16, x = 0 ; i >= 0; i--, x++) {
         y = (v + HALF) >> 16;
         patnpen_point(x, y);   /* draw next point */
         u += v >> N;
         v -= u >> N;
    }
}
```

patnpen_polyline *Draw Polyline with Pen and Pattern*

Syntax

```
typedef struct { short x, y; } POINT;

void patnpen_polyline(n, vert)
short n;      /* vertex count */
POINT *vert;  /* vertex coordinates */
```

Description

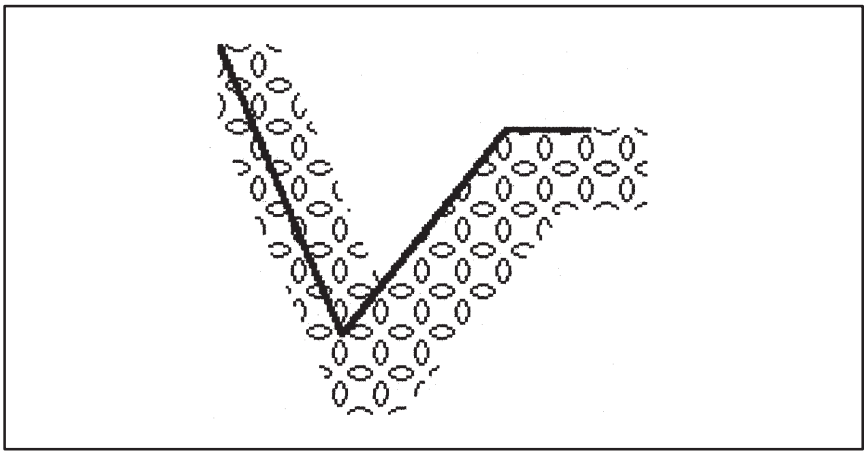
The *patnpen_polyline* function draws multiple, connected lines with a pen and an area-fill pattern. The thickness of the lines is determined by the width and height of the rectangular drawing pen. An array of integer x-y coordinates representing the polyline vertices is specified as one of the arguments. A line is drawn between each pair of adjacent vertices in the array. The area covered by the rectangular drawing pen as it traverses each line is drawn in the current area-fill pattern.

Argument *n* specifies the number of vertices in the polyline; the number of lines drawn is *n*–1.

Argument *vert* is an array of x-y coordinates representing the polyline vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function draws a line between each adjacent pair of vertices in the array. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

For the polyline to form a closed polygon, the calling program must ensure that the first and last vertices in the *vert* array are the same.

Example Use the *patnpen_polyline* function to draw a polyline with four vertices. Also use the *pen_polyline* function to superimpose a “thin” line on the “fat” line to mark the position of the pen relative to the specified polyline. The vertex coordinates given to both polyline functions are (16, 16), (64, 128), (128, 48), and (160, 48). This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>  /* define PATTERN structure */
#define  NVERTS    4   /* number of vertices in polyline */

typedef struct { short x, y; } POINT;

static short amoeba[16] =
{
                        /* 16x16 area-fill pattern */
       0x1008,  0x0C30,  0x03C0,  0x8001,  0x4002,  0x4002,  0x2004,
0x2004,
       0x2004,  0x2004,  0x4002,  0x4002,  0x8001,  0x03C0,  0x0C30,
0x1008,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)amoeba };
static POINT xy[NVERTS] =
{
    { 16, 16 }, { 64, 128 }, { 128, 48 }, { 160, 48 },
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);
    set_pensize(24, 32);
    patnpen_polyline(NVERTS, xy);     /* fat polyline */
    set_pensize(2, 2);
    pen_polyline(NVERTS, xy);         /* thin polyline */
}
```

pen_line *Draw Line with Pen*

Syntax

```
void pen_line(x1, y1, x2, y2)
short x1, y1;    /* start coordinates */
short x2, y2;    /* end coordinates */
```

Description

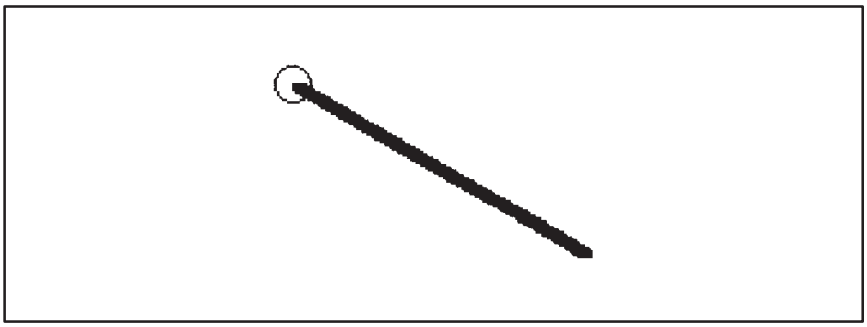
The *pen_line* function draws draw a line with a pen and a solid color. The thickness of the line is determined by the width and height of the rectangular drawing pen. The area covered by the pen to represent the line is filled with the current foreground color.

Arguments *x1* and *y1* specify the starting x and y coordinates of the line. Arguments *x2* and *y2* specify the ending x and y coordinates of the line.

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 with its top left corner touching the line. The area covered by the pen as it traverses the line from start to end is filled with a solid color.

Example

Use the *pen_line* function to draw a thick line from (16, 16) to (128, 80) with a 5-by-3 pen. Use the *draw_oval* function to draw a small circle around the start point of the line.



```
main()
{
    short x1, y1, x2, y2, r;

    set_config(0, !0);
    clear_screen(0);
    set_pensize(5, 3);
    x1 = 16;
    y1 = 16;
    x2 = 128;
    y2 = 80;
    pen_line(x1, y1, x2, y2);
    r = 7;
    draw_oval(2*r, 2*r, x1-r, y1-r);
}
```

Syntax

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

Description

The *pen_ovalarc* function draws an arc of an ellipse with a pen and a solid color. The ellipse from which the arc is taken is in standard position, with the major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The area swept out by the pen as it traverses the arc is filled with the current foreground color. The thickness of the arc is determined by the width and height of the rectangular drawing pen.

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_ovalarc* function, the pen is located with its top left corner touching the arc. The area covered by the pen as it traverses the arc from start to end is filled with a solid color.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

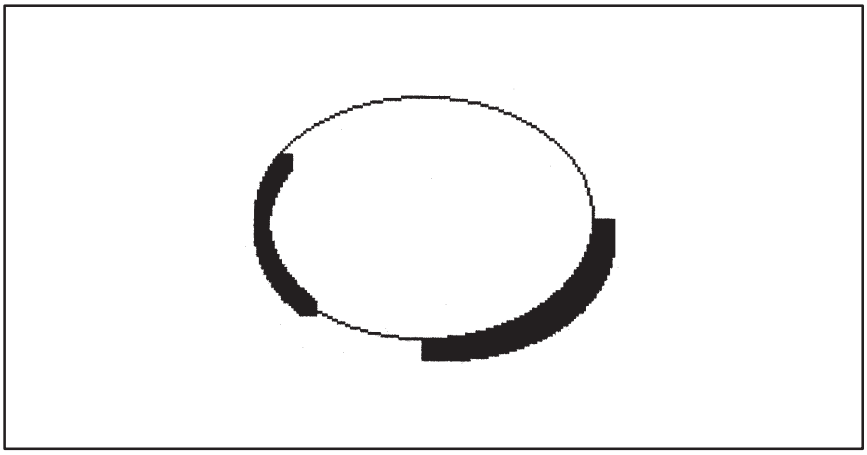
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent—that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *pen_ovalarc* function to draw two thick arcs taken from an ellipse of width 132 and height 94. Also, draw the ellipse with the *draw_oval* function.



```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 132;
    h = 94;
    x = 10;
    y = 10;
    draw_oval(w, h, x, y);
    set_pensize(9, 9);
    pen_ovalarc(w, h, x, y, 0, 90);
    set_pensize(6, 6);
    pen_ovalarc(w, h, x, y, 135, 210–135);
}
```

Syntax

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

Description

The *pen_piearc* function draws a pie-slice-shaped wedge from an ellipse with a pen and a solid color. 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 from which the arc is taken is in standard position, with the major and minor axes parallel to the coordinate axes. The ellipse is specified in terms of the enclosing rectangle in which it is inscribed. The area swept out by the pen as it traverses the perimeter of the wedge is filled with the current foreground color. The thickness of the arc and two lines drawn to represent the wedge is determined by the width and height of the rectangular drawing pen.

The pen is a rectangle whose width and height can be modified by means of the *set_pensize* function. As the pen traverses the arc from start to end, the pen is located with its top left corner touching the arc. The two lines connecting the arc start and end points with the center of the ellipse are drawn in similar fashion, with the top left corner of the pen touching each line as it traverses the line from start to end.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

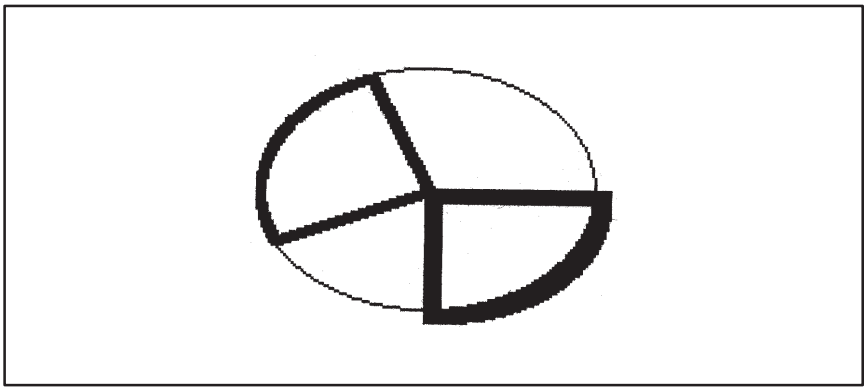
- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.

The last two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent—that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Example Use the *pen_piearc* function to draw two pie slices taken from an ellipse of width 132 and height 94. Also, draw the ellipse with the *draw_oval* function.



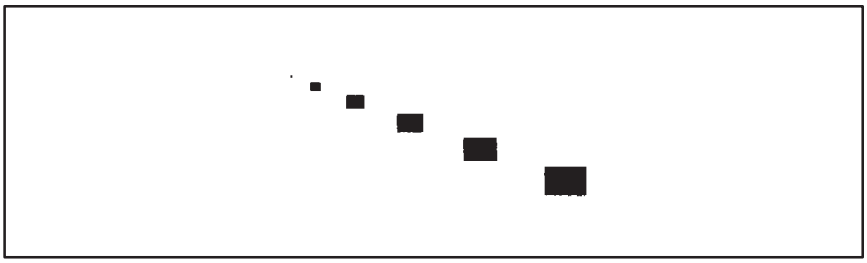
```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    w = 132;
    h = 94;
    x = 10;
    y = 10;
    draw_oval(w, h, x, y);
    set_pensize(7, 6);
    pen_piearc(w, h, x, y, 0, 90);
    set_pensize(4, 3);
    pen_piearc(w, h, x, y, 155, 250–155);
}
```

Syntax

```
void pen_point(x, y)
short x, y;      /* pen coordinates */
```

Description The *pen_point* function draws a point with a pen and a solid color. Arguments *x* and *y* specify where the top left corner of the rectangular drawing pen is positioned. The resulting figure is a rectangle the width and height of the pen and filled with the current foreground color.

Example Use the *pen_point* function to draw a series of rectangular pens of increasing size.



```
main()
{
    short w, h, x, y;

    set_config(0, !0);
    clear_screen(0);
    x = y = 10;
    w = h = 1;
    for ( ; x < 140; w += 3, h += 2, x += 2*w, y += h) {
        set_pensize(w, h);
        pen_point(x, y);
    }
}
```

pen_polyline *Draw Polyline with Pen*

Syntax

```
typedef struct { short x, y; } POINT;

void pen_polyline(n, vert)
short n;        /* vertex count */
POINT *vert;    /* vertex coordinates */
```

Description

The *pen_polyline* function draws multiple, connected lines with a pen and a solid color. The thickness of the lines is determined by the width and height of the rectangular drawing pen. An array of x-y coordinates representing the polyline vertices is specified as one of the arguments. A line is drawn between each pair of adjacent vertices in the array. The area covered by the rectangular drawing pen as it traverses each line is drawn in the current foreground color.

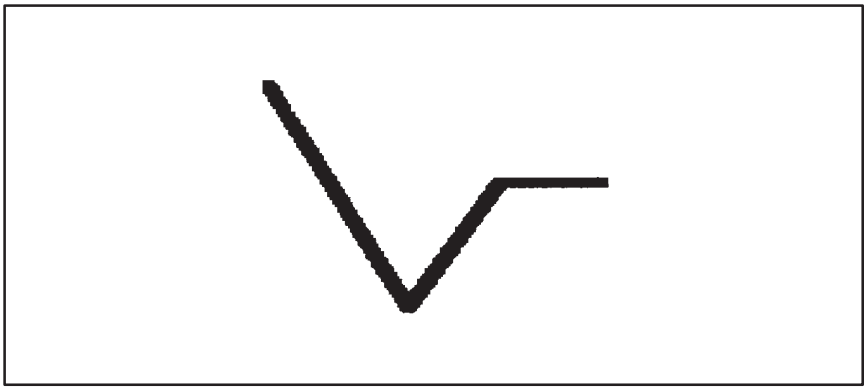
Argument *n* specifies the number of vertices in the polyline; the number of lines drawn is n–1.

Argument *vert* is an array of integer x-y coordinates representing the polyline vertices in the order in which they are to be traversed. The x-y coordinate pairs 0 through *n*–1 of the *vert* array contain the coordinates for the *n* vertices. The function draws a line between each adjacent pair of vertices in the array. Each vertex is represented by a 16-bit x-coordinate value followed by a 16-bit y-coordinate value. Coordinates are specified relative to the drawing origin.

Note that for the polyline to form a closed polygon, the calling program must ensure that the first and last vertices in the *vert* array are the same.

Example

Use the *pen_polyline* function to draw a “fat” polyline. The polyline vertices are at coordinates (10, 10), (64, 96), (100, 48), and (140, 48).



```
#define  NVERTS  4    /* number of vertices in polyline */

typedef struct { short x, y; } POINT;

static POINT xy[NVERTS] =
{
    { 10, 10 }, { 64, 96 }, { 100, 48 }, { 140, 48 }
};

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_pensize(5, 4);
    pen_polyline(NVERTS, xy);
}
```

Syntax

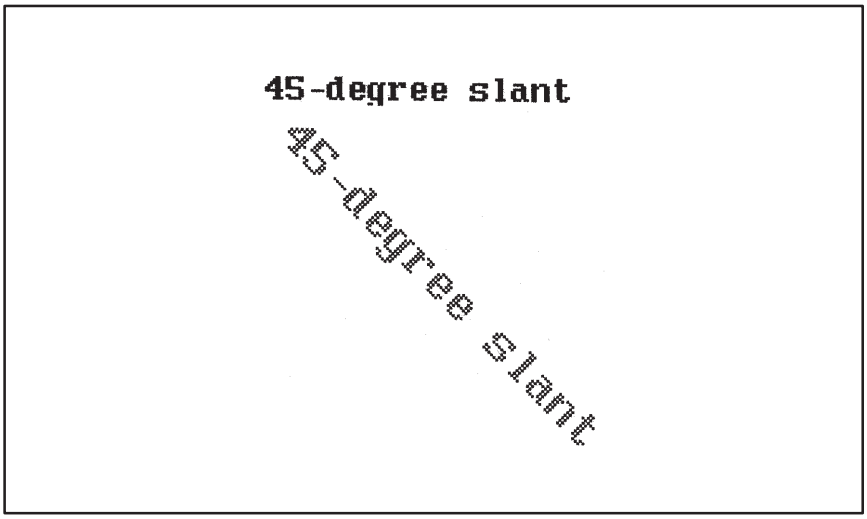
```
void put_pixel(val, x, y)
unsigned long val;          /* pixel value */
short x, y;                 /* pixel coordinates */
```

Description

The *put_pixel* function sets a pixel on the screen to a specified value. Argument *val* is the value written to the pixel. Arguments *x* and *y* are the coordinates of the pixel, defined relative to the drawing origin. If the screen pixel size is *n* bits, the pixel value is contained in the *n* LSBs of argument *val*; the higher-order bits of *val* are ignored.

Example

Use the *put_pixel* function to rotate a text image on the screen by 45 degrees. This example includes the C header file `gsptypes.h`, which defines the FONTINFO structure.




```
#include <gsptypes.h>     /* define FONTINFO structure */

main()
{
    FONTINFO fontinfo;
    short xs, ys, xd, yd, w, h;
    unsigned long val;
    char *s;

    set_config(0, !0);
    clear_screen(0);
    s = ”45–degree slant”;
    get_fontinfo(0, &fontinfo);
    w = text_width(s);
    h = fontinfo.charhigh;
    xs = ys = 0;
    text_out(xs, ys, s);
    for (xd = h, yd = h; ys < h; ys++, xd = h–ys, yd = ys+h)
        for (xs = 0; xs < w; xs++, xd++, yd++) {
            val = get_pixel(xs, ys);
            put_pixel(val, xd, yd);
        }
 }
```

Syntax

```
void seed_fill(x, y, buf, maxbytes)
short x, y;      /* seed pixel coordinates */
char *buf;          /* temporary buffer */
short maxbytes;  /* buffer capacity in bytes */
```

Description

The *seed_fill* function fills a connected region of pixels on the screen with a solid color, starting at a specified *seed pixel*. All pixels that are part of the connected region that includes the seed pixel are filled with the current foreground color.

The *seed color* is the original color of the specified seed pixel. All pixels in the connected region match the seed color before being filled with the foreground color.

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. (Having a diagonally adjacent neighbor that is part of the region is not sufficient.)

Arguments *x* and *y* specify the coordinates of the seed pixel, defined relative to the current drawing origin.

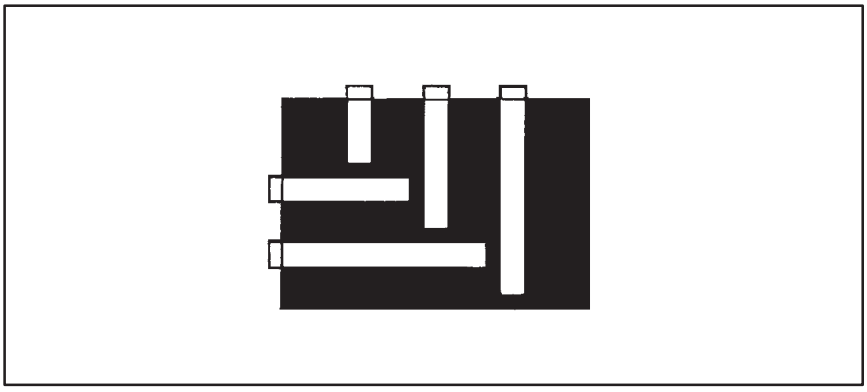
The last two arguments specify the temporary buffer used as a working storage during the seed fill. Argument *buf* is an array large enough to contain the temporary data that the function uses. Argument *maxbytes* is the number of 8-bit bytes available in the *buf* array. Working storage requirements can be expected to increase with the complexity of the connected region being filled.

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

- ❏ The seed pixel matches the current foreground color.
- ❏ The seed pixel lies outside the current clipping window.
- ❏ The storage buffer space specified by argument *maxbytes* is insufficient to continue.

In the last case, the function may have filled some portion of the connected region prior to aborting.

Example Use the *seed_fill* function to fill a connected region of pixels on the screen. Use the *draw_rect* function to draw a maze, the interior of which is filled by the *seed_fill* function.



```
#define  MAXBYTES   2048      /* size of temp buffer in bytes */

static char buf[MAXBYTES];    /* seed-fill temp buffer */

main()
{
    set_config(0, !0);
    clear_screen(0);

    /* Construct a maze consisting of 6 rectangles. */
    draw_rect(120, 80, 10, 10);
    draw_rect(10, 30, 35, 5);
    draw_rect(55, 10, 5, 40);
    draw_rect(10, 55, 65, 5);
    draw_rect(85, 10, 5, 65);
    draw_rect(10, 80, 95, 5);

    /* Now seed fill the interior of the maze. */
    seed_fill(20, 20, buf, MAXBYTES);
}
```

Syntax

```
void seed_patnfill(x, y, buf, maxbytes)
short x, y;      /* seed pixel coordinates */
char *buf;          /* temporary buffer */
short maxbytes;  /* buffer capacity in bytes */
```

Description

The *seed_patnfill* function fills a connected region of pixels with a pattern, starting at a specified *seed pixel*. All pixels that are part of the connected region that includes the seed pixel are filled with the current area-fill pattern.

The *seed color* is the original color of the specified seed pixel. All pixels in the connected region match the seed color before being filled with the pattern.

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. (Having a diagonally adjacent neighbor that is part of the region is not sufficient.)

Arguments *x* and *y* specify the coordinates of the seed pixel, defined relative to the current drawing origin.

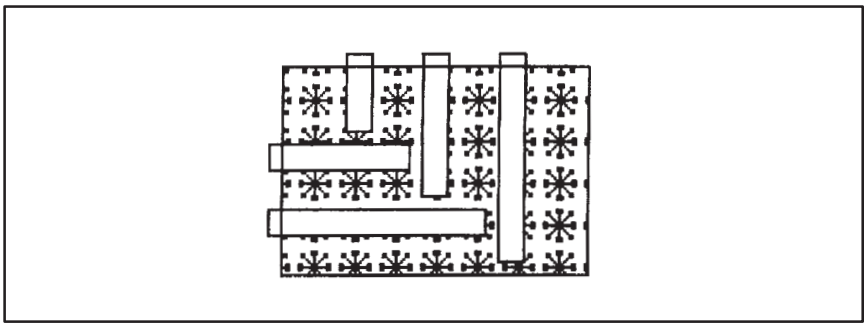
The last two arguments specify a buffer used as a working storage during the seed fill. Argument *buf* is an array large enough to contain the temporary data that the function uses. Argument *maxbytes* is the number of 8-bit bytes available in the *buf* array. Working storage requirements can be expected to increase with the complexity of the connected region being filled.

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

- The seed pixel matches either the current foreground color or background color. (The area-fill pattern is rendered in these two colors.)
- The seed pixel lies outside the current clipping window.
- The storage buffer space specified by *maxbytes* is insufficient to continue.

In the last case, the function may have filled some portion of the connected region prior to aborting.

Example Use the *seed_patnfill* function to fill a connected region of pixels on the screen with a pattern. Use the *draw_rect* function to draw a maze, the interior of which is filled by the *seed_patnfill* function. Note that the two colors in the area-fill pattern, white and blue, differ from the original color of the connected region, black. If either color in the pattern matches the seed pixel color, the *seed_patnfill* function will return immediately without drawing anything. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure, and the header file `colors.h`, which defines the color BLUE.



```
#include <gsptypes.h>  /* define PATTERN structure */
#include "colors.h"      /* define color "BLUE" */
#define  MAXBYTES   2048      /* size of temp buffer in bytes */

static char buf[MAXBYTES];    /* seed-fill temp buffer */
static short snowflake[16] = {    /* area-fill pattern */
       0x0000,  0x01C0,  0x19CC,  0x188C,  0x0490,  0x02A0,  0x31C6,
0x3FFE,
       0x31C6,  0x02A0,  0x0490,  0x188C,  0x19CC,  0x01C0,  0x0000,
0x0000,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)snowflake };

main()
{
    short w, h, x, y, n;

    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);

    /* Construct a maze consisting of 6 rectangles. */
    draw_rect(120, 80, 10, 10);
    draw_rect(10, 30, 35, 5);
    draw_rect(55, 10, 5, 40);
    draw_rect(10, 55, 65, 5);
    draw_rect(85, 10, 5, 65);
    draw_rect(10, 80, 95, 5);

    /* Fill the interior of the maze with a pattern. */
    set_bcolor(BLUE);
    seed_patnfill(20, 20, buf, MAXBYTES);
}
```

Syntax

```
short select_font(id)
short id;     /* font identifier */
```

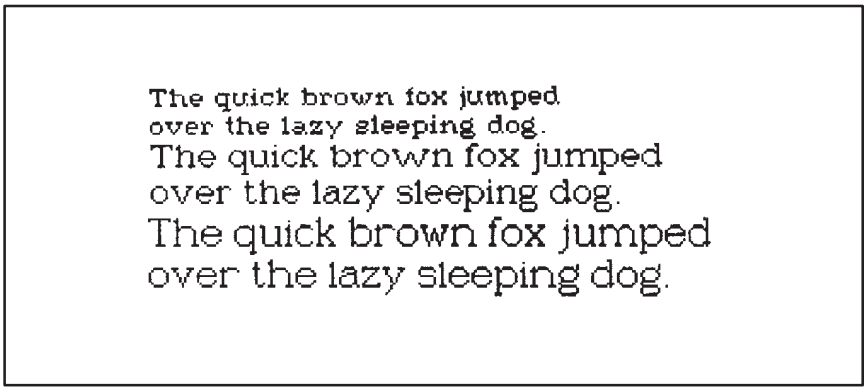
Description

The *select_font* function selects one of the installed fonts for use by the text functions. The input argument, *id*, is valid only if it identifies a font currently installed in the font table. Argument *id* must either be a valid identifier value returned by a previous call to the *install_font* function, or be 0, indicating selection of the system font.

A value of 0 is returned if the argument *id* is not valid; in this case, the function returns without attempting to select a new font. A nonzero value is returned if the selection is successful.

Example

Use the *select_font* function to select a previously installed font. Use the *install_font* function to install three proportionally spaced fonts, and for each of the three fonts in turn, select the font and use it to print a couple of lines of text to the screen. This example includes the C header file `gsptypes.h`, which defines the FONT and FONTINFO structures.



```
#include <gsptypes.h>    /* define FONT and FONTINFO struct's */
#define  NFONTS     3    /* number of fonts installed */

extern FONT ti_rom11, ti_rom14, ti_rom16;   /* 3 font names */

main()
{
    FONTINFO fontinfo;
    short i, n, x, y, index[NFONTS];

    set_config(0, !0);
    clear_screen(0);

    /* Install 3 proportionally-spaced fonts. */
    index[0] = install_font(&ti_rom11);
    index[1] = install_font(&ti_rom14);
    index[2] = install_font(&ti_rom16);

    /* Now select each of the three fonts in turn. */
    x = y = 10;
    for (i = 0; i < NFONTS; i++) {
        n = select_font(index[i]);  /* select next font */
        if (!n) {
            select_font(0);         /* select system font */
            text_out(x, y, "ERROR-- Font not installed!");
            exit(1);
        }
        get_fontinfo(index[i], &fontinfo);
        text_out(x, y, "The quick brown fox jumped");
        y += fontinfo.charhigh;
        text_out(x, y, "over the lazy sleeping dog.");
        y += fontinfo.charhigh;
    }
}
```

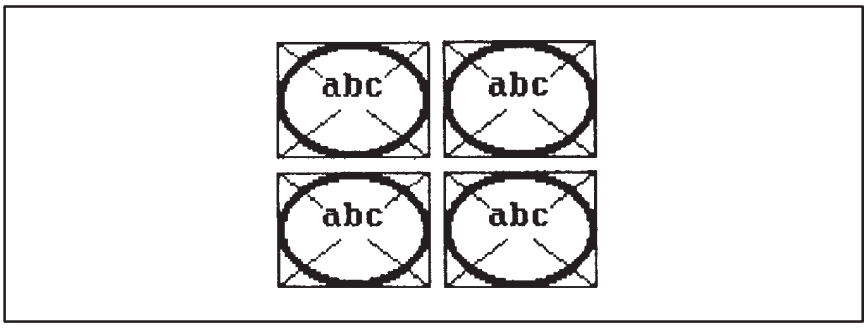
Syntax

```
void set_draw_origin(x, y)
short x, y;      /* new drawing origin */
```

Description The *set_draw_origin* function sets the position of the drawing origin for all subsequent drawing operations to the screen. The coordinates specified for all drawing functions are defined relative to the drawing origin. The x and y axes for drawing operations pass through the drawing origin, with x increasing to the right, and y increasing in the downward direction.

Arguments *x* and *y* are the horizontal and vertical coordinates of the new drawing origin relative to the screen origin at the top left corner of the screen.

Example Use the *set_draw_origin* function to move the drawing origin to various locations on the screen. In each case, verify that subsequent text and graphics output are positioned relative to the current origin.




```
main()
{
    short x, y, w;
    char *s;

    set_config(0, !0);
    clear_screen(0);
    s = "abc";
    w = text_width(s);

    for (y = 10; y < 100; y += 50)
        for (x = 10; x < 100; x += 65) {
            set_draw_origin(x, y);
            draw_line(0, 0, 60-1, 45-1);
            draw_line(0, 45-1, 60-1, 0);
            text_out(30-w/2, 10, "abc");
            frame_rect(60, 45, 0, 0, 1, 1);
            frame_oval(60, 45, 0, 0, 3, 3);
        }
}
```

Syntax

```
typedef long PTR;   /* 32-bit address */

void set_dstbm(baseaddr, pitch, xext, yext, psize)
PTR baseaddr;       /* bit map base address */
short pitch;        /* bit map pitch */
short xext, yext;   /* x and y extents */
short psize;        /* pixel size */
```

Description

The *set_dstbm* function sets the destination bit map for subsequent drawing functions. Currently, only the *bitblt* function can write to a bit map other than the screen. All other drawing functions abort (return without drawing anything) if the destination bit map is set to a bit map other than the screen.

Argument *baseaddr* is a pointer to the destination bit map. Invoking the function with a *baseaddr* value of 0 sets the destination bit map to the screen and causes the last four arguments to the function to be ignored. A nonzero *baseaddr* is interpreted as a pointer to a linear bit map; in other words, the destination bit map is contained in an off-screen buffer. The specified bit map should begin on an even pixel boundary in memory. For instance, when the pixel size is 32 bits, the 5 LSBs of the bit map's base address should be 0s.

Argument *pitch* is the difference in bit addresses from the start of one row of the bit map to the next. The *bitblt* function requires that the destination pitch be specified as a positive, nonzero multiple of the destination bit map's pixel size. The *bitblt* function executes more rapidly if the pitch is further restricted to be a multiple of 16.

Arguments *xext* and *yext* define the upper limits of the effective clipping window for a linear destination bit map. The pixel having the lowest memory address in the window is the pixel at (0, 0), whose address is *baseaddr*. The pixel having the highest memory address in the window is the pixel at (*xext*, *yext*), whose address is calculated as

address = baseaddr + yext*(pitch) + xext*(psize)

In the case of a linear bit map, responsibility for clipping is left to the calling program.

Example Use the *set_dstbm* function to designate an off-screen buffer as the destination bit map. Contract an image from the screen to 1 bit per pixel and store the contracted image in the off-screen buffer. Next, expand the image from 1 bit per pixel to the screen pixel size and copy to another area of the screen below the original image. This example includes the C header file `gsptypes.h`, which defines the FONT and FONTINFO structures.

```
#include <gsptypes.h>    /* define FONT and FONTINFO */
#define  MAXBYTES   4096   /* size of image buffer in bytes */

static char image[MAXBYTES];
static FONTINFO fontinfo;

main()
{
    short w, h, x, y, pitch;
    char *s;

    set_config(0, !0);
    clear_screen(0);

    /* Print one line of text to screen. */
    x = y = 10;
    s = "Capture this text image.";
    text_out(x, y, s);
    w = text_width(s);
    get_fontinfo(0, &fontinfo);
    h = fontinfo.charhigh;

    /* Make sure buffer is big enough to contain image. */
    pitch = ((w + 15)/16)*16;
    if (pitch*h/8 > MAXBYTES) {
        text_out(x, y+h, "Image too big!");
        exit(1);
    }

    /* Capture text image from screen. */
    set_dstbm(image, pitch, w, h, 1);  /* off-screen bit map */
    bitblt(w, h, x, y, 0, 0);       /* contract */

    /* Now copy text image to another area of screen. */
    swap_bm();
    bitblt(w, h, 0, 0, x, y+h);        /* expand */
}
```

Syntax

```
typedef long PTR;   /* 32-bit address */
typedef struct
{
   unsigned short width, height;
   unsigned short depth;
   PTR data;
} PATTERN;

void set_patn(ppatn)
PATTERN *ppatn;
```

Description

The *set_patn* function sets the fill pattern for subsequent drawing operations. This pattern is used for drawing functions such as *patnfill_rect* and *patnfill_oval* that fill regions with patterns. All pattern-filling functions are easily identified by their function names, which include the four-letter descriptor *patn*.

Argument *ppatn* is a pointer to a PATTERN structure.

The fields of the PATTERN structure are defined as follows:

- Fields *width* and *height* specify the dimensions of the pattern.
- Field *depth* specifies the pixel size of the pattern.
- Field *data* is a pointer to a bit map containing the actual pattern.

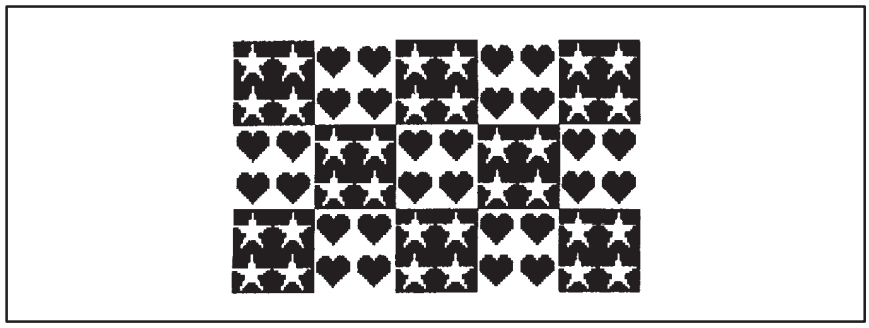
Only two-color 16-by-16 patterns are currently supported by the pattern-fill drawing functions. This means that the fields *width*, *height,* and *depth* of the PATTERN structure pointed to by argument *ppatn* must be specified as 16, 16, and 1, respectively. The *data* field is assumed to be a pointer to a 16-by-16, 1-bit-per-pixel bit map. A bit value of 1 in the pattern bit map specifies that the foreground color be used to draw the corresponding pixel; a bit value of 0 specifies the background color. The first pattern bit controls the pixel in the top left corner of the pattern; the last pattern bit controls the pixel in the bottom right corner.

The tiling of patterns to the screen is currently fixed relative to the top left corner of the screen. In other words, changing the drawing origin causes no shift in the mapping of the pattern to the screen, although the boundaries of the geometric primitives themselves (rectangles, ovals, and so on) are positioned relative to the drawing origin. The pixel at screen coordinates (x, y) is controlled by the bit at coordinates (x *mod* 16, y *mod* 16) in the pattern bit map.

The entire PATTERN structure is saved by the *set_patn* function, and the original structure pointed to by argument *ppatn* need not be preserved following the call to the function. However, the actual bit map containing the pattern is not saved by the function; this bit map must be preserved by the calling program as long as the pattern remains in use.

During initialization of the drawing environment by the *set_config* function, the area-fill pattern is set to its default state, which is to fill with solid foreground color.

Example Use *set_patn* function to change the area-fill pattern. With each change in pattern, call the *patnfill_rect* function to tile the screen with alternating star and heart patterns. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>    /* define PATTERN structure */

typedef struct { short row[16]; } PATNBITS;

static PATTERN fillpatn = { 16, 16, 1, (PTR)0 };
static PATNBITS patnbits[2] =
{
    {
        0x0000, 0x0000, 0x0E38, 0x1F7C,   /* heart pattern */
        0x3FFE, 0x3FFE, 0x3FFE, 0x3FFE,
        0x1FFC, 0x0FF8, 0x07F0, 0x03E0,
        0x01C0, 0x0080, 0x0000, 0x0000
    },
    {
        0xFFFF, 0xFF7F, 0xFF7F, 0xFF7F,   /* star pattern */
        0xFE3F, 0xFE3F, 0x8000, 0xE003,
        0xF007, 0xFC1F, 0xFC1F, 0xF80F,
        0xF9CF, 0xF3E7, 0xF7F7, 0xFFFF
    }
};

main()
{
    short x, y, index;

    set_config(0, !0);
    clear_screen(0);
    index = 0;
    for (x = 16; x < 160; x += 32)
        for (y = 16; y < 96; y += 32) {
            fillpatn.data = (PTR)&patnbits[index ^= 1];
            set_patn(&fillpatn);
            patnfill_rect(32, 32, x, y);
        }
}
```

set_pensize *Set Pen Size*

Syntax

```
void set_pensize(w, h)
short w, h;      /* pen width and height */
```

Description

The *set_pensize* function sets the dimensions of the pen for subsequent drawing operations. The pen is a rectangular shape that is used by drawing functions such as *pen_line* and *pen_ovalarc* to sweep out wide lines and arcs. All functions that utilize the pen are easily identified by their function names, which include the three-letter descriptor *pen*.

Arguments *w* and *h* specify the width and height of the pen. The width and height are specified in terms of pixels.

A mathematically ideal line is infinitely thin. Conceptually, a function such as *pen_line* renders a wide line by positioning the top left corner of the pen to coincide with the ideal line as the pen is moved from one end of the line to the other. The area swept out by the pen is filled with either a solid color (for instance, *pen_line*) or pattern (for instance, *patnpen_line*). Arcs are rendered in similar fashion.

Example

Use *set_pensize* function to change dimensions of rectangular drawing pen. Draw a point and a line to show the effect of the change in pen size.

```
main()
{
    set_config(0, !0);
    clear_screen(0);

    /* Draw point and line with default pen. */
    pen_point(10, 10);
    pen_line(20, 10, 100, 30);

    /* Set pen dimensions to 8x6. */
    set_pensize(8, 6);

    /* Draw new point and line. */
    pen_point(10, 30);
    pen_line(30, 30, 110, 50);
}
```

Syntax

```
typedef long PTR;   /* 32-bit address */

void set_srcbm(baseaddr, pitch, xext, yext, psize)
PTR baseaddr;       /* bit map base address */
short pitch;        /* bit map pitch */
short xext, yext;   /* x and y extents */
short psize;        /* pixel size */
```

Description

The *set_srcbm* function sets the source bit map for subsequent drawing functions. Currently, only the *bitblt* and *zoom_rect* functions can access a source bit map other than the screen.

Argument *baseaddr* is a pointer to the source bit map. Invoking the function with a *baseaddr* value of 0 designates the screen as the source bit map. In this case, the last four arguments are ignored by the function. A nonzero *baseaddr* is interpreted as a pointer to a linear bit map; that is, the source bit map is contained in an off-screen buffer. The specified bit map should begin on an even pixel boundary in memory. For instance, when the pixel size is 32 bits, the 5 LSBs of the bit map's base address should all be 0s.

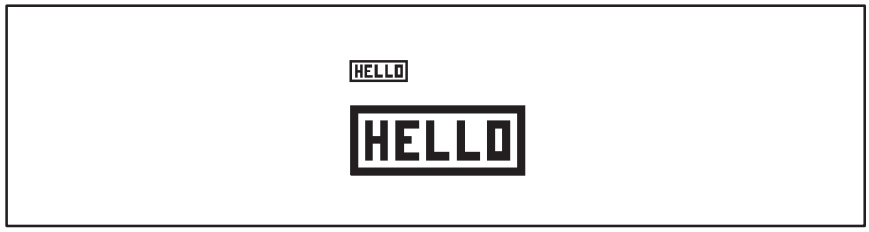
Argument *pitch* is the difference in bit addresses from the start of one row of the linear bit map to the next. The *bitblt* function requires that the source pitch be specified as a positive, nonzero multiple of the source bit map's pixel size. The *bitblt* function executes more rapidly if the pitch is further restricted to be a multiple of 16. The *zoom_rect* function requires that the source pitch be specified as a positive, nonzero multiple of 16. In the case of a 32-bit source pixel size, *zoom_rect* requires a multiple-of-32 pitch.

Arguments *xext* and *yext* define the upper limits of the effective clipping window for the linear bit map. The pixel having the lowest memory address in the window is the pixel at (0,0), whose address is *baseaddr*. The pixel having the highest memory address in the window is the pixel at (*xext,yext*), whose address is calculated as

address = baseaddr + yext*(pitch) + xext*(psize)

In the case of a linear bit map, responsibility for clipping is left to the application program.

Example Use the *set_srcbm* function to designate an off-screen buffer as the source bit map. Expand the image from 1 bit per pixel to the screen pixel size and copy the image to the screen.




```
#define  W            23      /* width of image in pixels */
#define  H            9       /* height of image in pixels */
#define  PITCH        32      /* pitch of image in bits */
#define  DEPTH        4       /* screen pixel size */
#define  MAXBYTES   DEPTH*W/8 /* zoom_rect buffer size in bytes */

static short image[H*PITCH/16] = {
    0xFFFF, 0x007F, 0x0001, 0x0040, 0x45D5, 0x005C,
    0x4455, 0x0054, 0x44DD, 0x0054, 0x4455, 0x0054,
    0xDDD5, 0x005D, 0x0001, 0x0040, 0xFFFF, 0x007F,
};
static char buf[4*W/8];     /* temp buffer for zoom_rect */

main()
{
    short x, y;

    set_config(0, !0);
    clear_screen(0);

    /* Expand image to screen. */
    x = y = 10;
    set_srcbm(image, PITCH, W, H, 1);  /* off-screen bit map */
    bitblt(W, H, 0, 0, x, y);

    /* Blow the image up so it's big enough to see. */
    set_srcbm(0, 0, 0, 0, 0);       /* screen */
    zoom_rect(W, H, x, y, 3*W, 3*H, x, y+2*H, buf);
}
```

Syntax

```
short set_textattr(pcontrol, count, val)
char *pcontrol;  /* control string */
short count;     /* val array length */
short *val;      /* array of attribute values */
```

Description

The *set_textattr* function sets text-rendering attributes. The function provides control over text attributes such as alignment, additional intercharacter spacing, and intercharacter gaps. The attributes specified by the function remain in effect during subsequent calls to the *install_font*, *select_font,* and *delete_font* functions.

Argument *pcontrol* is a control string specifying the attributes (one or more) to be updated. Argument *count* is the number of elements in the *val* array and is also the number of asterisks in the control string. Argument *val* is the array containing the attribute values designated by asterisks in the control string. The attribute values are contained in the consecutive elements of the *val* array, beginning with *val* [0], in the order in which they appear in the *pcontrol* string.

The following attributes are currently supported:

Symbol	Attribute Description	Option Value
%a	alignment	0 = top left, 1 = base line
%e	additional intercharacter spacing	16-bit signed integer
%f	fill gaps	0 = leave gaps, 1 = fill gaps
%r	reset all options	ignored

Values associated with attributes can be specified either as immediate values in the control string or as values in the *val* array. When an attribute value is passed as a string literal, it should be placed between the percent (%) character and the attribute symbol. When an attribute value is passed as a *val* array element, an asterisk (*) is placed between the percent character and the attribute symbol. Upon encountering the asterisk, the function will retrieve the value from the *val* array and increment its internal pointer to the next *val* array element.

The value returned by the function is the number of attributes successfully set.

Only the text attributes of proportionally spaced fonts can be modified by this function; the attributes of block fonts are fixed. Block fonts are characterized by uniform horizontal spacing between adjacent characters. Block fonts are always aligned to the top left corner of the character cell; that is, the position of a string of block text is always specified in terms of the x-y coordinates at the top left corner of the first character in the string. The intercharacter gaps between block-font characters are always filled with the background color.

The system font, font 0, is always a block font. Fonts installed by calls to the *install_font* function (identified by font indices 1, 2, and so on) may be selected to be either block fonts or proportionally spaced fonts.

In the case of a proportionally spaced font, text alignment in the y dimension can be set either to the top of the character or to the base line of the character. Text alignment in the x dimension is fixed at the left edge of the character. Immediately following initialization of the drawing environment by the *set_config* function, the alignment is to the top left corner of the character, which is the default.

The additional intercharacter spacing attribute specifies how many extra pixels of space are to be added (or subtracted in the case of a negative value) to the default horizontal separation between adjacent characters, as specified in the FONT data structure. Immediately following initialization of the drawing environment by the *set_config* function, the additional intercharacter spacing is 0, which is the default.

The intercharacter gaps attribute controls whether the gaps between horizontally adjacent characters are automatically filled with the background color. When this attribute is enabled, one line of proportionally spaced text may be cleanly written directly on top of another without first erasing the text underneath. Immediately following initialization of the drawing environment by the *set_config* function, the filling of intercharacter gaps is disabled, which is the default.

Examples Set the text alignment mode to top-left-corner position. This can accomplished by assigning the value 1 to attribute symbol *%a* by means of the literal method:

```
set_textattr("%1a", 0, 0);
```

Note that in the example above the third argument is ignored by the function.

The same effect can be achieved by passing the attribute value in the *val* array. An asterisk is placed between the "%" and the "a" in the control string, and *val* [0] contains the attribute value, 1:

```
short val[1];
val[0] = 1;
set_textattr("%*a", 1, val);
```

The following example sets two attributes in a single call to *set_textattr*. It sets the text alignment mode to base line position using a literal value embedded in the control string, and sets the additional intercharacter spacing to –21 by passing the value through the *val* array:

```
short val[1];
val[0] = -21;
set_textattr("%0a%*e", 1, val);
```

The same effect can be achieved by passing both values through the *val* array:

```
short val[1];
val[0] = 0;
val[1] = -21;
set_textattr("%*a%*e", 2, val);
```

Finally, the following function call resets all text attributes to their default values:

```
set_textattr("%0r",0,0);
```

styled_line *Draw Styled Line*

Syntax

```
void styled_line(x1, y1, x2, y2, style, mode)
short  x1, y1;      /* start coordinates */
short  x2, y2;      /* end coordinates */
long style;         /* 32-bit line style pattern */
short mode             /* 1 of 4 drawing modes */
```

Description

The *styled_line* function uses Bresenham's algorithm to draw a styled line from the specified start point to the specified end point. The line is a single pixel thick and is drawn in the specified line-style pattern.

Arguments *x1* and *y1* specify the starting coordinates of the line. Arguments *x2* and *y2* specify the ending coordinates. Coordinates are specified relative to the drawing origin. 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 consumed 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 the foreground color is used to draw the corresponding pixel. A bit value of 0 in the pattern means that the corresponding pixel is either drawn in the background color (drawing modes 1 and 3) or not drawn (modes 0 and 2).

The function supports four drawing modes:

mode 0 – Does not draw background pixels (leaves gaps); loads new line-style pattern from *style* argument.

mode 1 – Draws background pixels, and loads new line-style pattern from *style* argument.

mode 2 – Does not draw background pixels (leaves gaps); reuses old line-style pattern (ignores *style* argument).

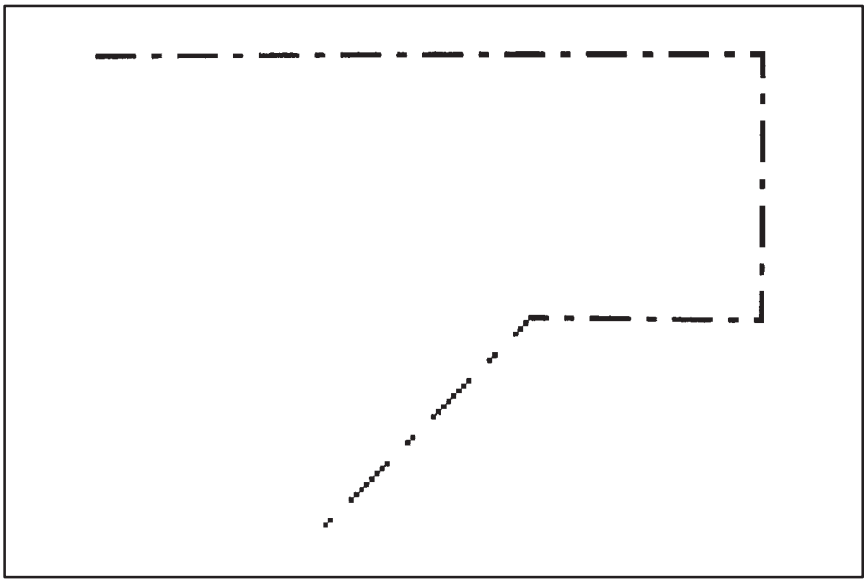
mode 3 – Draws background pixels and reuses old line-style pattern (ignores *style* argument).

Drawing modes 2 and 3 support line-style pattern reuse in instances in which the pattern must be continuous across two or more connecting lines. During the course of drawing a line of length *n* (in pixels), the original line-style pattern is rotated left (*n*–1) modulo 32 bits. The rotated pattern is always saved by the function before returning. The saved pattern is ready to be used as the pattern for a new line that continues from the end of the line just drawn.

During initialization of the drawing environment by the *set_config* function, the line-style pattern is set to its default value, which is all 1s.

The current line-style pattern can be obtained by calling the *get_env* function. See the *get_env* function description for more information.

Example Use the *styled_line* function to draw four connected lines. The line-style pattern is continuous from one line segment to the next.



```
#define  DOTDASH    0x18FF18FF  /* dot-dash line-style pattern */
#define  NEW        0           /* mode = load new line style */
#define  OLD        2           /* mode = re-use old line style */

main()
{
    set_config(0, !0);
    clear_screen(0);
    styled_line(10, 10, 140, 10, DOTDASH, NEW);
    styled_line(140, 10, 140, 60, 0, OLD);
    styled_line(140, 60, 95, 60, 0, OLD);
    styled_line(95, 60, 55, 100, 0, OLD);
}
```

styled_oval *Draw Styled Oval*

Syntax

```
void styled_oval(w, h, xleft, ytop, style, mode)
short w, h;         /* ellipse width and height */
short xleft, ytop;  /* top left corner */
long style;         /* 32-bit line-style pattern */
short mode;         /* selects 1 of 4 drawing modes */
```

Description

The *styled_oval* function draws the styled outline of an ellipse, given the enclosing rectangle in which the ellipse is inscribed. The outline of the ellipse is only one pixel in thickness and is drawn using a 32-bit line-style pattern. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes.

The first four arguments specify the rectangle enclosing the ellipse:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.
- If either the width or height is 0, the oval is not drawn.

The line-style pattern is specified in argument *style*, a long integer containing a 32-bit repeating line-style pattern. Pattern bits are consumed in the order 0,1,...,31, where bit 0 is the LSB. The pattern is repeated modulo 32, as the ellipse is drawn. A bit value of 1 in the pattern specifies that the foreground color is used to draw the corresponding pixel. A bit value of 0 means that the corresponding pixel is either drawn in the background color (modes 1 and 3) or not drawn (modes 0 and 2). The ellipse is drawn in the clockwise direction on the screen, beginning at the rightmost point of the ellipse if $w < h$, or at the bottom of the ellipse if $w \geq h$.

The function supports four drawing modes:

mode 0 – Does not draw background pixels (leaves gaps); loads new line-style pattern from *style* argument.

mode 1 – Draws background pixels and loads new line-style pattern from *style* argument.

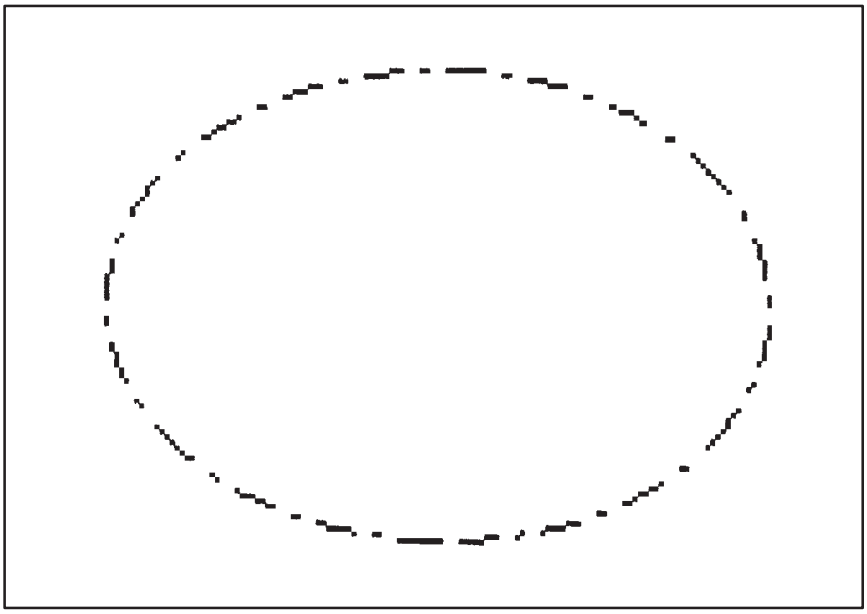
mode 2 – Does not draw background pixels (leaves gaps); reuses old line-style pattern (ignores *style* argument).

mode 3 – Draws background pixels and reuses old line-style pattern (ignores *style* argument).

The (rotated) pattern is always saved by the function before returning. This pattern is available to draw a subsequent arc or line.

During initialization of the drawing environment by the *set_config* function, the line-style pattern is set to its default value, which is all 1s.

Example Use the *styled_oval* function to render the outline of an ellipse with a 32-bit repeating line-style pattern.



```
#define  DOTDASH   0x18FF18FF   /* dot–dash line-style pattern */

main()
{
    set_config(0, !0);
    clear_screen(0);
    styled_oval(130, 90, 10, 10, DOTDASH, 0);
}
```

styled_ovalarc *Draw Styled Oval Arc*

Syntax

```
void styled_ovalarc(w, h, xleft, ytop, theta, arc, style,
                                                    mode)
short w, h;         /* width and height    */
short xleft, ytop;  /* top left corner     */
short theta;        /* starting angle (degrees)   */
short arc;             /* angle extent (degrees)  */
long style;         /* 32-bit line-style pattern  */
short mode;         /* selects 1 of 4 drawing modes */
```

Description

The *styled_ovalarc* function draws a styled arc taken from an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the x and y axes. The arc is drawn one pixel in thickness using the specified repeating line-style pattern. The ellipse from which the arc is taken is specified in terms of the enclosing rectangle in which it is inscribed.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin. If either the width or height is 0, the arc is not drawn.

The next two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Argument *style* is a long integer containing a 32-bit repeating line-style pattern. Pattern bits are consumed 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 the foreground color is used to draw the corresponding pixel. A bit value of 0 in the pattern means that the corresponding pixel is either drawn in the background color (drawing modes 1 and 3) or not drawn (modes 0 and 2).

The function supports four drawing modes:

mode 0 – Does not draw background pixels (leaves gaps); loads new line-style pattern from *style* argument.

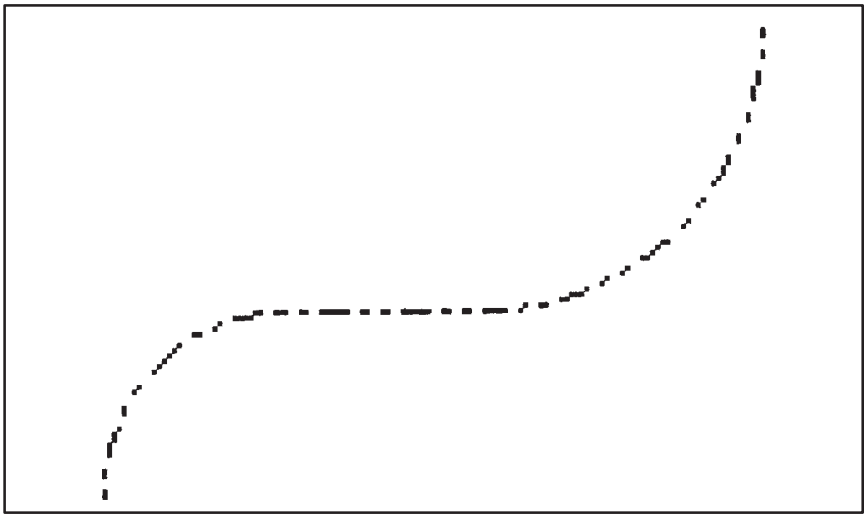
mode 1 – Draws background pixels and loads new line-style pattern from *style* argument.

mode 2 – Does not draw background pixels (leaves gaps); reuses old line-style pattern (ignores *style* argument).

mode 3 – Draws background pixels and reuses old line-style pattern (ignores *style* argument).

The (rotated) pattern is always saved by the function before returning. This pattern is available to draw a subsequent arc or line.

Example Use the *styled_ovalarc* function to draw two arcs that are rendered with a dot-dot-dash line-style pattern. Use the *styled_line* function to draw a line connecting the two arcs. The line-style pattern is continuous at the joints between the arcs and the line.



```
#define  DOTDOTDASH 0x3F333F33  /* ..-..- line-style pattern */
#define  NEW        0           /* mode = load new line style */
#define  OLD        2           /* mode = re-use old line style */

main()
{
    set_config(0, !0);
    clear_screen(0);
    styled_ovalarc(70, 70, 10, 65, 180, 90, DOTDOTDASH, NEW);
    styled_line(45, 65, 85, 65, -1, OLD);
    styled_ovalarc(110, 110, 30, -45, 90, -90, -1, OLD);
}
```

styled_piearc *Draw Styled Pie Arc*

Syntax

```
void styled_piearc(w, h, xleft, ytop, theta, arc, style,

mode)
short w, h;         /* width and height */
short xleft, ytop;  /* top left corner */
short theta;        /* starting angle (degrees) */
short arc;             /* angle extent (degrees) */
long style;         /* 32-bit line-style pattern */
short mode;         /* selects 1 of 4 drawing modes */
```

Description

The *styled_piearc* function draws a styled arc taken from an ellipse. Two straight, styled 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 x and y axes. The arc and the two lines from the center are drawn one pixel in thickness using the specified repeating line-style pattern. The ellipse from which the arc is taken is specified in terms of the enclosing rectangle in which it is inscribed.

The first four arguments specify the rectangle enclosing the ellipse from which the arc is taken:

- Arguments *w* and *h* specify the width and height of the rectangle.
- Arguments *xleft* and *ytop* specify the coordinates at the top left corner of the rectangle and are defined relative to the drawing origin.
- If either the width or height is 0, the arc is not drawn.

The next two arguments define the limits of the arc and are specified in integer degrees:

- Argument *theta* specifies the starting angle and is measured from the center of the right side of the enclosing rectangle.
- Argument *arc* specifies the arc's extent – that is, the number of degrees (positive or negative) spanned by the angle.

Both arguments are expressed in degrees of elliptical arc, with 360 degrees representing one full rotation around the ellipse. For both arguments, positive angles are in the clockwise direction, and negative angles are counterclockwise. Argument *theta* is treated as modulus 360. If the value of argument *arc* is outside the range [–359,+359], the entire ellipse is drawn.

Argument *style* is a long integer containing a 32-bit repeating line-style pattern. Pattern bits are consumed 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 the foreground color is used to draw the corresponding pixel. A bit value of 0 in the pattern means that the corresponding pixel is either drawn in background color (drawing modes 1 and 3) or not drawn (modes 0 and 2).

The function supports four drawing modes:

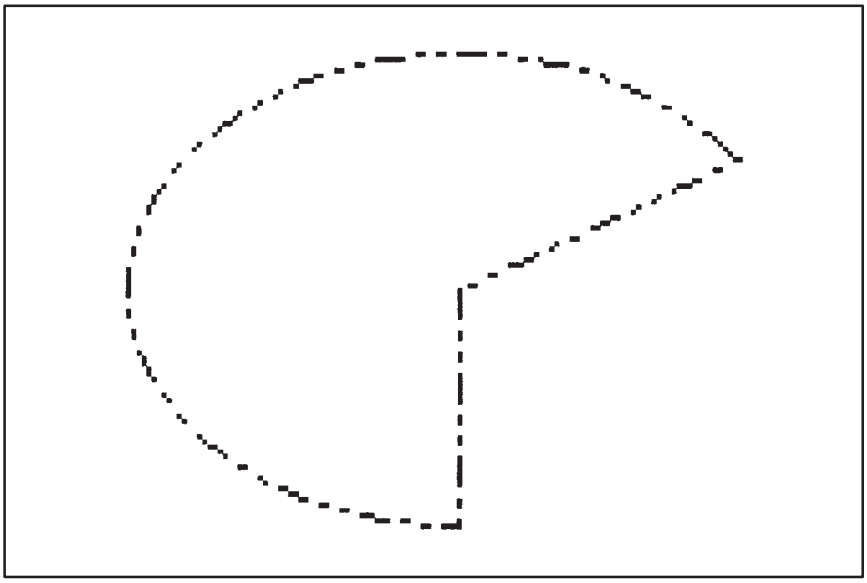
mode 0 – Does not draw background pixels (leaves gaps); loads new line-style pattern from *style* argument.

mode 1 – Draws background pixels and loads new line-style pattern from *style* argument.

mode 2 – Does not draw background pixels (leaves gaps); reuses old line-style pattern (ignores *style* argument).

mode 3 – Draws background pixels and reuses old line-style pattern (ignores *style* argument).

Example Use the *styled_piearc* function to draw a pie slice taken from an ellipse of width 130 and height 90. The slice traverses a 237-degree arc of the ellipse extending from –33 degrees to –270 degrees, drawn in the counterclockwise direction around the perimeter of the ellipse.



```
#define  DOTDOTDASH 0x3F333F33       /* line-style pattern */

main()
{
    set_config(0, !0);
    clear_screen(0);
    styled_piearc(130, 90, 10, 10, –33, –270+33, DOTDOTDASH, 0);
}
```

swap_bm *Swap Source and Destination Bit Maps*

Syntax `void swap_bm()`

Description The *swap_bm* function swaps the source and destination bit maps. To move pixels back and forth between two bit maps, this function is more convenient than calling both the *set_srcbm* and *set_dstbm* functions.

Example Use the *swap_bm* function to swap the source and destination bit maps. Initially, the destination bit map is designated as an off-screen buffer, and the source bit map is the screen. A line of text is rendered on the screen, and its image is contracted from the screen pixel depth to one bit per pixel and stored in the off-screen buffer by a call to the *bitblt* function. Following a call to *swap_bm*, the destination bit map is the screen, and the source bit map is the off-screen buffer. The captured image is copied to the screen three times by three calls to the *bitblt* function. This example includes the C header file `gsptypes.h`, which defines the FONT and FONTINFO structures.

```
#include <gsptypes.h>     /* define FONT and FONTINFO */
#define  MAXBYTES    2048   /* size of image buffer in bytes */

static char image[MAXBYTES];
static FONTINFO fontinfo;

main()
{
    short w, h, x, y, pitch;
    char *s;

    set_config(0, !0);
    clear_screen(0);

    /* Print one line of text to screen. */
    x = y = 10;
    s = "TEXT IMAGE";
    text_out(x, y, s);
    w = text_width(s);
    get_fontinfo(0, &fontinfo);
    h = fontinfo.charhigh;

    /* Make sure buffer is big enough to contain image. */
    pitch = ((w + 15)/16)*16;
    if (pitch*h/8 > MAXBYTES) {
        text_out(x, y+h, "Image won't fit!");
        exit(1);
    }

    /* Capture text image from screen. */
    set_dstbm(image, pitch, w, h, 1);   /* off-screen bit map */
    bitblt(w, h, x, y, 0, 0);           /* contract */

    /* Now copy text image to 3 other areas of screen. */
    swap_bm();
    bitblt(w, h, 0, 0, x, y+h);         /* expand copy #1 */
    bitblt(w, h, 0, 0, x, y+2*h);       /* expand copy #2 */
    bitblt(w, h, 0, 0, x, y+3*h);       /* expand copy #3 */
}
```

Syntax

```
short text_width(s)
unsigned char *s;   /* character string */
```

Description The *text_width* returns the width of the string in pixels, as if it were rendered using the current selected font and the current set of text-drawing attributes. Argument *s* is a string of 8-bit ASCII character codes terminated by a *null* (0) character code.

Example Use the *text_width* function to enclose a line of text in a rectangular frame. This example includes the C header file `gsptypes.h`, which defines the FONTINFO structure.


```
#include <gsptypes.h>         /* define FONTINFO structure */
#define  DX      5            /* frame thickness in x dimension */
#define  DY      4            /* frame thickness in y dimension */

main()
{
    FONTINFO fontinfo;
    short w, h, x, y;
    char *s;

    set_config(0, !0);
    clear_screen(0);
    s = "Enclose this text.";
    get_fontinfo(0, &fontinfo);
    w = text_width(s);
    h = fontinfo.charhigh;
    x = y = 10;
    text_out(x+2*DX, y+2*DY, s);
    frame_rect(w+4*DX, h+4*DY, x, y, DX, DY);
}
```

Syntax

```
void zoom_rect(ws, hs, xs, ys, wd, hd, xd, yd, rowbuf)
short ws, hs;    /* source width and height */
short xs, ys;    /* source top left corner */
short wd, hd     /* destination width and height   */
short xd, yd;    /* destination top left corner    */
char *rowbuf;    /* temporary row buffer */
```

Description

The *zoom_rect* function expands or shrinks a two-dimensional source array of pixels to fit the dimensions of a rectangular destination array on the screen. The source array may be either a rectangular area of the screen or a pixel array contained in an off-screen buffer. The width and height of the source array are specified independently from (and in general differ from) those of the destination array. Horizontal zooming is accomplished by replicating or collapsing (by deleting, for instance) columns of pixels from the source array to fit the width of the destination array. Vertical zooming is accomplished by replicating or collapsing rows of pixels from the source array to fit the height of the destination array. This type of function is sometimes referred to as a *stretch blit*.

The source and destination arrays are contained within the currently selected source and destination bit maps; these bit maps are selected by calling the *set_srcbm* and *set_dstbm* functions before calling *zoom_rect*. Calling the *set_config* function with the *init_draw* argument set to a nonzero value causes both the source and destination bit maps to be set to the default bit map, which is the screen. The *zoom_rect* function requires that the pixel sizes for the source and destination bit maps be the same. The destination bit map must be the screen.

The first four arguments define the source array:

- Arguments *ws* and *hs* specify the width and height of the source array.
- Arguments *xs* and *ys* specify the x and y displacements of the top left corner of the source array from the origin. If the source bit map is the screen, the current drawing origin is used. If the source bit map is an off-screen buffer, the origin lies at the bit map's base address, as specified to the *set_srcbm* function.

The next four arguments define the destination array on the screen:

- Arguments *wd* and *hd* specify the width and height of the destination array.
- Arguments *xd* and *yd* specify the x and y coordinates at the top left corner of the source array, defined relative to the drawing origin.

The final argument, *rowbuf*, is a buffer large enough to contain one complete row of either the destination array or the source array, whichever has the greater width. (A buffer the width of the screen will *always* be sufficient.)

The required storage capacity in 8-bit bytes is calculated by multiplying the array width by the pixel size and dividing the result by 8.

Each of the following conditions is treated as an error that causes the *zoom_rect* function to abort (return immediately) without drawing anything:

- The destination is not the screen.
- The source and destination pixel sizes are not the same.
- The widths and heights specified for the source and destination arrays are not all nonnegative. No value is returned by the function in any event.

Only the portion of the destination rectangle lying within the current clipping window is modified by this function. The source rectangle, however, is permitted to lie partially or entirely outside the clipping window, in which case the pixels lying within the source rectangle are zoomed to the destination, regardless of whether they are inside or outside the window. The applications programmer is responsible for constraining the size and position of the source rectangle to ensure that it encloses valid pixel values.

The only exception to this behavior occurs when the left or top edge of the source rectangle lies in negative screen coordinate space, in which case the function automatically clips the source rectangle to positive x-y coordinate space; in most systems, this means that the source is clipped to the top and left edges of the screen. The resulting clipped source rectangle is zoomed to the destination rectangle and justified to the lower right corner of the specified destination rectangle. Portions of the destination rectangle corresponding to clipped portions of the source are not modified.

If the desired effect is to zoom a 1-bit-per-pixel bit map to the screen and the screen pixel size is greater than 1, the zoom operation must be done in two stages. First, the *bitblt* function is called to expand the original bit map to a color pixel array contained in an off-screen buffer. Second, the *zoom_rect* function is called to zoom the expanded pixel array from the off-screen buffer to the screen.

Shrinking in the horizontal direction causes some number of horizontally-adjacent source pixels to be collapsed to a single destination pixel. Similarly, shrinking in the vertical direction causes some number of vertically adjacent rows of source pixels to be collapsed to a single row in the destination array. When several source pixels are collapsed to a single destination pixel, they are combined with each other and with the destination background pixel according to the selected pixel-processing operation code. 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 by using a *Boolean-OR* operation (at 1 bit per pixel) or a *max* operation (at multiple bits per pixel).

The function internally disables transparency during the zoom operation but restores the original transparency state prior to returning.

The *zoom_rect* function may yield unexpected results for the following pixel- processing operation codes:

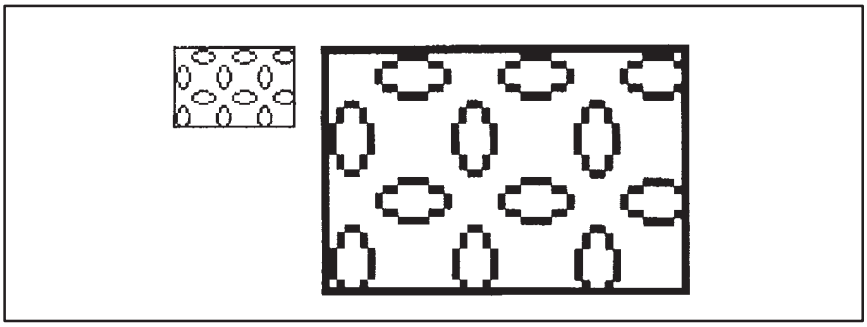
PPOP Code	Operation
7	~src AND ~dst
11	~src AND dst
13	~src OR dst
14	~src OR ~dst
15	~src

When used in conjunction with the *zoom_rect* function, selecting these operations causes the source array to be 1s complemented not once, as might be expected, but twice.

The buffer specified by the *rowbuf* argument is not used if all three of the following conditions are satisfied:

1) The pixel-processing operation code is 0 (*replace*).
2) The destination width and height are both greater than or equal to the source width and height.
3) The top of the destination rectangle does not lie above the top of the screen (in negative-y screen space).

Example Use the *zoom_rect* function to blow up an area-fill pattern for closer inspection. The image is zoomed by a factor of 3. This example includes the C header file `gsptypes.h`, which defines the PATTERN structure.



```
#include <gsptypes.h>          /* define PATTERN structure */
#define  W                48   /* width of source rectangle */
#define  H                32   /* height of source rectangle */
#define  X                12   /* left edge of source rectangle */
#define  Y                12   /* top edge of source rectangle */
#define  Z                3    /* zoom factor */
#define  DEPTH            4    /* screen pixel size */
#define  MAXBYTES   DEPTH*Z*W/8  /* zoom_rect buffer size in bytes
*/

static short tinyblobs[16] =
{                              /* 16x16 area-fill pattern */
       0x1008,  0x0C30,  0x03C0,  0x8001,  0x4002,  0x4002,  0x2004,
0x2004,
       0x2004,  0x2004,  0x4002,  0x4002,  0x8001,  0x03C0,  0x0C30,
0x1008,
};
static PATTERN fillpatn = { 16, 16, 1, (PTR)tinyblobs };
static char buf[MAXBYTES];

main()
{
    set_config(0, !0);
    clear_screen(0);
    set_patn(&fillpatn);
    patnfill_rect(W, H, X, Y);
    frame_rect(W, H, X, Y, 1, 1);
    zoom_rect(W, H, X, Y, Z*W, Z*H, X+W+10, Y, buf);
}
```

Appendix A

Data Structures

This appendix describes the data structures defined and used within the TMS340 Graphics Library. The following is a list of all the structure types to be discussed:

Structure Type	Description
BITMAP	Bit map (actually a two-dimensional pixel array)
CONFIG	Information about display system and graphics mode
ENCODED_RECT	Header for image compressed by *encode_rect* function
ENVIRONMENT	Information about drawing environment
ENVTEXT	Information about text environment
FONT	Header for a bit-mapped font
FONTINFO	Information about a bit-mapped font
MODEINFO	Information about a particular graphics mode
OFFSCREEN_AREA	Information about a particular off-screen buffer
PAGE	Information about a particular video page
PALET	Information about a particular palette entry
PATTERN	Information about a particular area-fill pattern

Certain library functions retrieve information about the graphics environment and copy that information into one of the structures listed above. The following is a list of these functions and the structures they use to present information to the application program:

Function	Structure Type
get_config	CONFIG
get_env	ENVIRONMENT
get_fontinfo	FONTINFO
get_modeinfo	MODEINFO
get_offscreen_memory	OFFSCREEN_AREA
get_palet	PALET

Note that the definitions of the structure types above may change in subsequent revisions of TIGA and the TMS340 Graphics Library. To minimize the impact of such revisions, write your application programs to refer to the ele-

ments of the structure symbolically by their field names, rather than as offsets from the start of the structure. The include files provided with the library and with TIGA will be updated, as necessary, in future revisions to track any such changes in data structure definitions.

Within the graphics library, information about the graphics environment is available in the form of global variables that are structures. The global variables and the corresponding structure types are listed below:

Global Variable	Structure Type	Global Variable Description
config	CONFIG	Information about display system and current graphics mode
DEFAULT_PALET[ ]	PALET	Default 16-color palette
env	ENVIRONMENT	Current drawing environment
envtext	ENVTEXT	Current text environment
**modeinfo*	MODEINFO	Pointer to information about current graphics mode
**offscreen*	OFFSCREEN_AREA	Pointer to array of off-screen buffers available in current graphics mode
**page*	PAGE	Pointer to array of video pages available in current graphics mode
palet[ ]	PALET	Array of current palette entries
pattern	PATTERN	Current area-fill pattern
**sysfont*	FONT	Pointer to system font

An asterisk in the left column indicates that the associated variable is a pointer; a pair of square brackets indicates that the variable is an array. These global variables are accessible to all library functions and to any proprietary library extensions added by the user. While application programs running under the TMS340 Graphics Library may also directly access these globals, applications running in the TIGA environment cannot. If emulation of the TIGA environment is important to your applications, access the library globals indirectly through inquiry functions such as *get_config* and *get_env*. This practice will enhance the portability of applications between the graphics library and TIGA.

The type PTR, which appears in several of the structure definitions that follow, is defined as follows:

```
typedef unsigned long PTR;
```

Type PTR is a 32-bit pointer to a location in the TMS340 graphics processor's memory.

A.1 BITMAP Structure Definition

The BITMAP data structure contains information describing a two-dimensional array of pixels. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    PTR addr;                /*  pixel  array  base  address
*/
    unsigned short pitch;     /* pitch in bits */
    unsigned short xext, yext;   /* x and y extents */
    unsigned short psize;     /* pixel size in bits */
} BITMAP;
```

The fields of the data structure are utilized as follows:

addr 32-bit base address of pixel array in TMS340 graphics processor's memory

pitch Difference in starting addresses between adjacent rows of pixel array

xext The extent of pixel array in x dimension (pixels per row)

yext The extent of pixel array in y dimension (number of rows)

psize The depth (number of bits per pixel) of pixel array

The BITMAP structure is used in the definition of the ENVIRONMENT structure. For more information, refer to the description of the *get_env* function in Chapter 7.

A.2 CONFIG Structure Definition

The CONFIG data structure contains information about the display system and graphics mode. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned short version_number;       /* TIGA revision number */
    unsigned long comm_buff_size;        /* undefined in graphics library */
    unsigned long sys_flags;             /* coprocessor presence flags */
    unsigned long device_rev;            /* GSP silicon revision number */
    unsigned short num_modes;            /* number of graphics modes */
    unsigned short current_mode;         /* current graphics mode */
    unsigned long program_mem_start;     /* start of program memory */
    unsigned long program_mem_end;       /* end of program memory */
    unsigned long display_mem_start;     /* start of display memory */
    unsigned long display_mem_end;       /* end of display memory */
    unsigned long stack_size;            /* max. stack size in bytes */
    unsigned long shared_mem_size;       /* undefined in graphics library */
    HPTR shared_host_addr;               /* undefined in graphics library */
    PTR shared_gsp_addr;                 /* undefined in graphics library */
    MODEINFO mode;                       /* graphics mode information */
} CONFIG;
```

The HPTR type is a 32-bit pointer to a location in the host processor's memory. The graphics library's global variable *config* is a structure of type CONFIG. This global contains information describing the display system and the current graphics mode. For more information on the CONFIG structure, refer to the description of the *get_config* function in Chapter 6.

A.3 ENCODED_RECT Structure Definition

The ENCODED_RECT data structure defines the header information at the beginning of a compressed image encoded by the *encode_rect* function. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned short magic;     /* magic number */
    unsigned long length;     /* length of data in bytes */
    unsigned short scheme;    /* encoding scheme */
    short width, height;      /* dimensions of image rectangle */
    short psize;              /* pixel size of image */
    short flags;              /* usage varies with scheme */
    unsigned long clipadj;    /* x-y clipping adjustments */
} ENCODED_RECT;
```

For more information, refer to the description of the *encode_rect* function in Chapter 7.

A.4 ENVIRONMENT Structure Definition

The ENVIRONMENT data structure specifies the values of the parameters in the drawing environment. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned long xyorigin;   /* x–y drawing origin */
    unsigned long pensize;    /* width and height of pen */
    BITMAP *srcbm, *dstbm;    /* source and destination bit maps */
    unsigned long stylemask;  /* line–style pattern */
} ENVIRONMENT;
```

The graphics library's global variable *env* is a structure of type ENVIRONMENT and contains information describing the current drawing environment. The ENVIRONMENT structure is also the format in which the *get_env* function retrieves the information describing the current drawing environment. For more information, refer to the description of the ENVIRONMENT data structure in the description of the *get_env* function in Chapter 7.

A.5 ENVTEXT Structure Definition

The ENVTEXT data structure contains information describing the current text environment. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    short installed;          /* number of fonts installed */
    short allocated;          /* number of slots allocated */
    FONT **font;           /* pointer to font table */
    FONT *selected;        /* pointer to current font */
    short align;           /* alignment (top left or baseline) */
    short charextra;          /* additional intercharacter spacing */
    long effects;          /* special effects */
    short xposn, yposn;    /* current text x-y coordinates */
} ENVTEXT;
```

The graphics library's global variable *envtext* is a structure of type ENVTEXT that describes the current text attributes and provides access to the installed fonts. The fields of the ENVTEXT structure are utilized as follows:

installed The number of fonts currently installed in the font table (in addition to font 0, which is permanently installed)

allocated The maximum number of fonts that can be installed in the font table, in addition to font 0

font A pointer to the start of the font table, which is a table of pointers to all currently installed fonts

selected A pointer to the currently selected font

align Text alignment attribute (0 = align to top left corner of first character in string, 1 = align to character origin at base line)

charextra Extra intercharacter spacing attribute (value added to default spacing specified in font structure)

effects Reserved for future use

xposn x coordinate of position at which next call to *text_outp* will start printing

yposn y coordinate of position at which next call to *text_outp* will start printing

By convention, the *allocated* field above must contain a value of 16 or greater. In other words, the font table will always be large enough to permit at least 16 fonts to be installed, in addition to font 0, which is the permanently installed system font.

A.6 FONT Structure Definition

The FONT data structure defines the header information at the start of a bit-mapped font. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned short magic; /* font type code */
    long length;          /* length of font in bytes */
    char facename[30];    /* string containing name of font */
    short deflt;          /* ASCII code of default character */
    short first;          /* ASCII code of first character */
    short last;           /* ASCII code of last character */
    short maxwide;        /* maximum character width */
    short maxkern;        /* maximum character kerning amount */
    short charwide;       /*  width  of  characters  (0  if  proportional)
*/
    short avgwide;        /* average width of characters */
    short charhigh;       /* character height */
    short ascent;         /* ascent (how far above base line) */
    short descent;        /* descent (how far below base line) */
    short leading;        /* leading (row bottom to next row top) */
    long rowpitch;        /* bits per row of char patterns */
    long oPatnTbl;        /* bit offset to pattern table */
    long oLocTbl;         /* bit offset to location table */
    long oOwTbl;          /* bit offset to offset/width table */
} FONT;
```

The graphics library's global variable *sysfont* is a pointer to the system font, which begins with a header of type FONT. The FONT structure is also used in the definition of the ENVTEXT and FONTINFO structures. For more information, refer to the discussion of the FONT structure in Chapter 5.

A.7 FONTINFO Structure Definition

The FONTINFO structure defines the format in which the *get_fontinfo* function retrieves information describing the designated font. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    char facename[30];     /* string containing name of font */
    short deflt;           /* ASCII code of default character */
    short first;           /* ASCII code of first character */
    short last;            /* ASCII code of last character */
    short maxwide;         /* maximum character width */
    short avgwide;         /* average width of characters */
    short maxkern;         /* maximum kerning amount */
    short charwide;        /* width of characters (0=proportional) */
    short charhigh;        /* character height */
    short ascent;          /* ascent (how far above base line) */
    short descent;         /* descent (how far below base line) */
    short leading;         /* leading (row bottom to next row top) */
    FONT *fontptr;         /* address of font in GSP memory */
    short id;              /* font identifier (font table index) */
} FONTINFO;
```

The FONTINFO structure is the format in which the *get_fontinfo* retrieves the information describing the specified installed font. For more information, refer to the description of the *get_fontinfo* function in Chapter 6.

A.8 MODEINFO Structure Definition

The MODEINFO data structure contains information describing a particular graphics mode. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned long disp_pitch;        /* display pitch */
    unsigned short disp_vres;        /* number of scan lines */
    unsigned short disp_hres;        /* pixels per scan line */
    short screen_wide;               /* screen width in centimeters */
    short screen_high;               /* screen height in centimeters */
    unsigned short disp_psize;          /* pixel size in bits */
    unsigned long pixel_mask;        /* pixel mask */
    unsigned short palet_gun_depth;  /* bits per gun */
    unsigned long palet_size;        /* number of palette entries */
    short palet_inset;               /* palette offset from frame */
    unsigned short num_pages;        /* number of display pages */
    short num_offscrn_areas;         /* number of offscreen buffers */
    unsigned long wksp_addr;         /* offscreen workspace address */
    unsigned long wksp_pitch;        /* offscreen workspace pitch */
    unsigned short vram_block_write; /* VRAM block write flag */
} MODEINFO;
```

The graphics library's global variable *modeinfo* is a pointer to a structure of type MODEINFO that describes the current graphics mode. The MODEINFO structure is the format in which the *get_modeinfo* function retrieves the information describing the specified graphics mode. Also, the MODEINFO structure is used in the definition of the CONFIG structure. For more information, refer to the description of the *get_modeinfo* function in Chapter 6.

A.9 OFFSCREEN_AREA Structure Definition

The OFFSCREEN_AREA structure contains information describing a particular off-screen buffer area within the display memory. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    PTR addr;                /* base address of off–screen buffer */
    unsigned short xext, yext;   /* x and y extents of buffer */
} OFFSCREEN_AREA;
```

By convention, the pitch and pixel size of an off-screen buffer are always identical to those used for the display. These values can be obtained from the *mode.disp_pitch* and *mode.disp_psize* fields of the CONFIG structure retrieved by the *get_config* function.

The graphics library's global variable *offscreen* is a pointer to an array of type OFFSCREEN_AREA that specifies all the off-screen buffers available in the current graphics mode. For more information, refer to the discussion of the *get_offscreen_memory* function in Chapter 6.

A.10 PAGE Structure Definition

The PAGE data structure contains information describing a particular video page (or frame buffer). The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned long BaseAddr;          /* base address of start of page
*/
    unsigned long DpyStart;      /* display start value of page */
} PAGE;
```

The graphics library's global variable *page* is a pointer to an array of type PAGE that describes all the video pages available in the current graphics mode. The length of this array is specified in the *num_pages* field value retrieved by the *get_config* function; the length is also available in the global structure *config*, as element *config.mode.num_pages*. The fields of the PAGE structure are utilized as follows:

BaseAddr The base address of the video page; that is, the 32-bit memory address of the pixel in the top left corner of the monitor screen

DpyStart The display start address, which is the address of the first pixel output to refresh the monitor

The *BaseAddr* and *DpyStart* fields usually contain identical values. The exception occurs in the case of a display system utilizing a TMS34070 Color Palette device in either frame-load or line-load mode. In a system with a TMS34070, the display starting value loaded into the TMS34010's DPYSTRT register or into the TMS34020's DPYSTL and DPYSTH registers does not correspond to the page's base address; it corresponds instead to the start of the palette information that precedes the page in display memory. Note that the *palet_inset* field retrieved by the *get_config* function (this value is also available in the global structure *config* as element *config.mode.palet_inset*) is specified as the difference between the *BaseAddr* and *DpyStart* values given for the page:

```
palet_inset = BaseAddr – DpyStart
```

The array of page information pointed to by global variable *page* is utilized by the *set_config* and *page_flip* functions, described in Chapter 6.

A.11 PALET Structure Definition

The PALET data structure contains the R-G-B and intensity components of a particular palette entry. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned char r;     /* red component */
    unsigned char g;     /* green component */
    unsigned char b;     /* blue component */
    unsigned char i;     /* intensity component */
} PALET;
```

The graphics library's global variable DEFAULT_PALET is an array of type PALET containing the 16 default palette entries, as described in the discussion of the *init_palet* function in Chapter 6. Library global *palette* is an array of type PALET that contains the palette currently in use. The length of this array is specified in the *palet_size* field value retrieved by the *get_config* function; the length is also available in the global structure *config*, as element *config.mode.palet_size*. For more information on the PALET structure, refer to the discussion of the *get_palet* and *set_palet* functions in Chapter 6.

A.12 PATTERN Structure Definition

The PATTERN data structure contains information describing an area-fill pattern. The following is the C *typedef* statement describing the structure:

```
typedef struct
{
    unsigned short width;     /* width of pattern */
    unsigned short height;    /* height of pattern */
    unsigned short depth;     /* depth (bits/pixel) */
    PTR data;                 /* pointer to data */
    int (*hsrv)();            /* horizontal fill routine */
    int (*srv)();             /* general fill routine */
} PATTERN;
```

The graphics library's global variable *pattern* is a structure of type PATTERN which specifies the current area-fill pattern. The *set_patn* function accepts as its sole argument a pointer to a structure of type PATTERN. (The function uses only the first four fields of the PATTERN structure pointed to.) The fields of the PATTERN data structure are utilized as follows:

width	Number of pixels per row of pixel array containing pattern
height	Number of rows in pixel array containing pattern
depth	Depth (number of bits per pixel) in pixel array containing pattern
data	Pointer to pixel array containing pattern
hsrv	Pointer to function that renders horizontal lines (sometimes called spans) of regions to be filled with patterns
srv	Pointer to function that renders two-dimensional cells having same width and height as pattern array itself

Currently, patterns are restricted to width 16, height 16, and depth 1. The *hsrv* field points to a routine that is called from assembly language. This routine renders a single horizontal line (or span) of a region to be filled, and it expects the B-file registers to be set up as they would be for a FILL XY instruction. For an example of such a routine, refer to the `patnline.asm` source file in the graphics library's `\extprims` directory.

Appendix B

Reserved Symbols

This appendix contains a list of the global symbols defined within the TMS340 Graphics Library. These symbols should be treated as reserved. In order to ensure correct operation of your application programs and proprietary library extensions, avoid defining global symbols that conflict with those in the library.

The graphics library's reserved symbols are partitioned into three lists. The symbols in the Core and Extended Primitives Libraries are listed separately below, as are the global names for the bit-mapped fonts.

The symbols in the Core and Extended Primitives Libraries that are preceded by "_dm_" are TIGA direct-mode entry points. Refer to the *TIGA-340 Interface User's Guide* for more information.

B.1 Symbols in Core Primitives Library

ILLOP_PC_IN_A0
_bottom_of_stack
_clear_frame_buffer
_clear_page
_clear_screen
_config
_cpw
_cvxyl
_DEFAULT_PALET
_default_setup
_delay
_dm_clear_frame_buffer
_dm_clear_page
_dm_clear_screen
_dm_cpw
_dm_cvxyl
_dm_get_nearest_color
_dm_gsp2gsp
_dm_init_palet
_dm_lmo
_dm_peek_breg
_dm_poke_breg
_dm_rmo
_dm_set_bcolor
_dm_set_clip_rect
_dm_set_colors
_dm_set_fcolor
_dm_set_palet_entry
_dm_set_pmask
_dm_set_ppop
_dm_set_text_xy
_dm_set_windowing
_dm_set_wksp
_dm_text_outp
_end_of_dram
_env
_envtext
_field_extract
_field_insert
_getrev
_get_colors

_get_config
_get_fontinfo
_get_modeinfo
_get_nearest_color
_get_offscreen_memory
_get_palet
_get_palet_entry
_get_pmask
_get_ppop
_get_text_xy
_get_transp
_get_vector
_get_windowing
_get_wksp
_gsp2gsp
_init_palet
_init_text
_init_video_regs
_lmo
_modeinfo
_mode_setup
_monitorinfo
_null_patn_line
_num_modes
_oemdata
_oemmsg
_offscreen
_page
_page_busy
_page_flip
_palet
_pattern
_peek_breg
_poke_breg
_rmo
_setup
_set_bcolor
_set_clip_rect
_set_colors
_set_config
_set_dpitch
_set_fcolor
_set_palet

_set_palet_entry
_set_pmask
_set_ppop
_set_text_xy
_set_vector
_set_windowing
_set_wksp
_stack_size
_start_of_dram
_sysfont
_text_out
_text_outp
_transp_off
_transp_on
_wait_scan

B.2 Symbols in Extended Primitives Library

_arcstyle
_arc_draw
_arc_fill
_arc_pen
_arc_quad
_arc_quadrant
_arc_slice
_bitblt
_decode_rect
_delete_font
_dm_bitblt
_dm_draw_line
_dm_draw_oval
_dm_draw_ovalarc
_dm_draw_piearc
_dm_draw_point
_dm_draw_polyline
_dm_draw_rect
_dm_fill_convex
_dm_fill_oval
_dm_fill_piearc
_dm_fill_polygon
_dm_fill_rect
_dm_frame_oval
_dm_frame_rect
_dm_get_pixel
_dm_move_pixel
_dm_patnfill_convex
_dm_patnfill_oval
_dm_patnfill_piearc
_dm_patnfill_polygon
_dm_patnfill_rect
_dm_patnframe_oval
_dm_patnframe_rect
_dm_patnpen_line
_dm_patnpen_ovalarc
_dm_patnpen_piearc
_dm_patnpen_point
_dm_patnpen_polyline
_dm_pen_line
_dm_pen_ovalarc

_dm_pen_piearc
_dm_pen_point
_dm_pen_polyline
_dm_put_pixel
_dm_seed_fill
_dm_seed_patnfill
_dm_set_draw_origin
_dm_set_patn
_dm_set_pensize
_dm_styled_oval
_dm_styled_ovalarc
_dm_styled_piearc
_dm_zoom_rect
_draw_eliparc
_draw_line
_draw_oval
_draw_ovalarc
_draw_piearc
_draw_point
_draw_polyline
_draw_rect
_encode_rect
_fill_convex
_fill_eliparc
_fill_oval
_fill_piearc
_fill_polygon
_fill_rect
_frame_oval
_frame_rect
_get_env
_get_pixel
_get_textattr
_install_font
_in_font
_move_pixel
_onarc
_patnfill_convex
_patnfill_oval
_patnfill_piearc
_patnfill_polygon
_patnfill_rect
_patnframe_oval

_patnframe_rect
_patnpen_line
_patnpen_ovalarc
_patnpen_piearc
_patnpen_point
_patnpen_polyline
_patn_line
_pen_eliparc
_pen_line
_pen_ovalarc
_pen_piearc
_pen_point
_pen_polyline
_put_pixel
_seed_fill
_seed_patnfill
_select_font
_set_draw_origin
_set_dstbm
_set_patn
_set_pensize
_set_srcbm
_set_textattr
_sin_tbl
_styled_line
_styled_oval
_styled_ovalarc
_styled_piearc
_swap_bm
_text_width
_trig_values
_zoom_rect

B.3 Global Font Names

_arrows25, _arrows31
_austin11, _austin15, _austin20, _austin25, _austin38, _austin50
_corpus15, _corpus16, _corpus26, _corpus29, _corpus49
_devons23, _devons28, _devons41
_fargo22, _fargo26, _fargo38
_galves12, _galves15, _galves21, _galves22, _galves28, _galves42
_houstn14, _houstn17, _houstn20, _houstn26, _houstn38, _houstn50
_lucken07
_math16, _math19, _math24, _math32, _math44, _math64
_sanant22, _sanant28, _sanant40
_sys16, _sys24
_tampa18, _tampa22, _tampa30, _tampa42
_ti_art22, _ti_art28, _ti_art41, _ti_art54, _ti_art82
_ti_bau11, _ti_bau14, _ti_bau17, _ti_bau19, _ti_bau22, _ti_bau24,
_ti_bau28, _ti_bau43, _ti_bau56
_ti_clo27, _ti_clo40
_ti_dom23, _ti_dom25, _ti_dom30, _ti_dom42, _ti_dom46
_ti_hel11, _ti_hel15, _ti_hel18, _ti_hel20, _ti_hel22, _ti_hel24,
_ti_hel28, _ti_hel32, _ti_hel36, _ti_hel42, _ti_hel54, _ti_hel82
_ti_prk15, _ti_prk18, _ti_prk21, _ti_prk23, _ti_prk25, _ti_prk28,
_ti_prk43, _ti_prk54
_ti_rom11, _ti_rom14, _ti_rom16, _ti_rom18, _ti_rom20, _ti_rom22,
_ti_rom26, _ti_rom30, _ti_rom33, _ti_rom38, _ti_rom52, _ti_rom78
_ti_typ11, _ti_typ14, _ti_typ16, _ti_typ18, _ti_typ20, _ti_typ22,
_ti_typ26, _ti_typ38

Appendix C

Glossary

A

archive file: A file which is formed by concatenating several files, often using an encoding method that compresses the original files.

area-fill pattern: A two-dimensional pattern of pixels used to fill regions bounded by lines and curves.

ascent: Font metric; the distance in pixels from the base line to the top of the highest character in the font.

ASCII character code: American Standard Code for Information Exchange. A standard code for representing both control and graphic characters. Each character is represented by a 7-bit code, or by an 8-bit code if a parity bit is included. This code is widely used for information exchange among data processing and communications systems.

B

background color: The color to which the 0s in bit maps and area-fill patterns are set; also the color in which the pixels within the rectangles surrounding bit-mapped character shapes are drawn. The pixel value corresponding to the background color is pixel-replicated and loaded into the COLOR0 register of the TMS340 graphics processor.

base line: An invisible reference line corresponding to the bottom of the characters in a font, excluding the descenders.

binary image file: A file containing the binary image of a program or a block of data exactly as it appears when loaded into memory.

binsrc: A utility program for converting a binary image file to a C or assembly language file.

BitBlt: Bit-aligned block transfer. A BitBlt operation copies a bit map from one location in memory to another. The copy operation may involve com-

bining each pair of corresponding source and destination bits according to one of 16 possible Boolean operations. (Also see *PixBlt*.)

bit map: The digital representation of an image in which bits are mapped to pixels. This is a special case of a pixel array in which the pixel size is 1 bit. (See *pixel array*.)

bit-mapped font: A digital representation of a font in which the character shapes are stored as bit maps.

blanking interval: The time during which the monitor's electron beam is extinguished during the horizontal and vertical retrace periods.

block font: A font that emulates the cell-mapped text output by older, character-ROM-driven terminals and by display adapters such as the VGA.

block write cycle: A type of video-RAM write cycle that is typically used to rapidly fill an area of the frame buffer with a particular pixel value. Block write cycles are supported by TI's TMS44C251 1-megabit video RAM. Prior to the block write cycle, a 4-bit color latch within each VRAM is loaded with 4 bits of pixel data. During the block write cycle, the level input to the VRAM on each of its 4 data pins controls whether the color latch is written to a particular region of the memory; in other words, up to 16 bits may be written within the VRAM during a single block write cycle.

Bresenham's algorithm: An algorithm first described by J. E. Bresenham that is widely used for drawing straight lines on raster displays.

bulk initialization: A method for rapidly replicating a pixel value through all or a portion of video memory. This method can be utilized only with video RAM devices that support serial-register-to-memory cycles.

C

clipping window: The rectangular region on the screen to which graphics output is restricted. No drawing is allowed to occur outside the current clipping window.

COFF: Common Object File Format. Originally defined by AT&T, this format is used for the object files created by the TMS340 Family assembler, C compiler and linker. Details are presented in the *TMS340 Family Code Generation Tools User's Guide*.

cof2bin: A utility program for converting a COFF file to a binary image file.

color lookup table: (See *palette*.)

color palette: (See *palette*.)

conditional assembly statements: Directives to the assembler that control whether particular blocks of assembly language statements are assembled or ignored.

conditional compilation statements: Directives to the compiler that control whether particular blocks of high-level-language statements are compiled or ignored.

Core Primitives Library: The TMS340 Graphics Library consists of two parts, the Extended Primitives and the Core Primitives. This is similar to TIGA. The Core Primitives include the functions for initializing and querying the graphics environment, as well as rudimentary text and graphics capabilities.

D

DAC: Digital-to-analog converter. A device that converts a digital input code to an analog output voltage or current. The output level is the analog representation of the digital value input to the device.

dearchive: The process of extracting the original files from an archive file.

descent: Font metric; the distance in pixels from the base line to the bottom of the lowest descender in the font.

display memory: The portion of memory that can be displayed. The display memory contains one or more video pages that can be output to a monitor. In a TMS34010- or TMS34020-based display system, the display memory typically consists of video RAM devices.

display page: The video page that is currently displayed on the monitor.

double buffering: A technique for achieving flickerless animation that requires two video pages (or frame buffers). While the graphics processor draws to one page, the alternate page is displayed on the monitor. When the graphics output to the drawing page is completed, the old drawing page becomes the new display page, and *vice versa*.

DRAM refresh: Dynamic random-access memory refresh. The operation of maintaining data stored in DRAM devices. Data are stored in DRAMs as electrical charges across a grid of capacitive cells, and the charge stored in a cell will leak off over time unless it is refreshed.

drawing coordinate system: The x–y coordinate system in which all drawing (graphics output) operations are specified. The drawing origin can be moved relative to the fixed screen origin. The x coordinates increase from left to right, and y coordinates from top to bottom.

drawing origin: The position of the x-y origin on the display surface, used as a reference for positioning graphics output. The drawing origin can

move relative to the screen origin, which is fixed in the top left corner of the screen.

drawing page: The video page that is currently the designated target of all graphics output commands.

E

em, em space: Font metric; equal to the square of the type size used.

Extended Primitives Library: The TMS340 Graphics Library consists of two parts, the Extended Primitives and the Core Primitives. This is similar to TIGA. The Extended Primitives comprise most of the sophisticated graphics functions, including lines, curves, fills, patterns and proportionally-spaced text.

F

field: 1) A group of contiguous bits in a register or memory that are dedicated to a particular function or represent a single entity. 2) A software-configurable data type in a TMS34010 or TMS34020 whose length can be programmed to be any value in the range 1 to 32 bits.

font: A complete assortment of characters of a particular size and typeface.

font size: The size of a bit-mapped font in the graphics library, specified in terms of the height in pixels. This height is equal to the sum of the font's ascent and descent parameters.

font table: A data structure within the TMS340 Graphics Library that contains pointers to all currently installed fonts.

foreground color: The color to which solid lines, curves, and filled areas are set; the color to which the 1s in bit maps and area-fill patterns are set; also the color in which bit-mapped character shapes are drawn. The pixel value corresponding to the foreground color is pixel-replicated and loaded into the COLOR1 register of the TMS340 graphics processor.

frame: In the case of a noninterlaced display, the screen image output to the raster display monitor during a single vertical sweep of the electron beam. In the case of an interlaced display, a frame is composed of two fields, each of which requires a separate vertical sweep.

frame buffer: A portion of memory used to buffer rasterized data to be output to a CRT display monitor. This term sometimes refers either to a video page or to the entire display memory.

frame time: The time required to display a single frame on a monitor. In the case of a noninterlaced display, the frame time corresponds to the

time required to complete a full vertical sweep. This is typically about 1/60 second.

G

graphics mode: A software-controlled configuration of a hardware display system to select a specified set of display characteristics such as pixel size, screen resolution, monitor-specific video timings, and number of video pages. Each display system to which the TMS340 Graphics Library has been ported supports one or more graphics modes.

gray scale: A scale of light intensities ranging from black to white in more or less uniform steps.

GSP: Graphics System Processor. Texas Instruments' TMS340 Family currently includes two GSPs, the TMS34010 and the TMS34020.

gspcl: A shell utility program provided with the TMS340 Family Code Generation Tools that automatically runs one or more program source files through the C compiler, assembler, or linker.

gspl: A COFF loader utility program provided for use with the TMS340 Graphics Library. The utility downloads an executable file from the PC to a TMS34010- or TMS34020-based graphics card and executes it.

gun, electron gun: The element of a cathode-ray tube that emits the electrons that form the beam that sweeps over the phosphors on the screen. The strength of the beam is modulated by the applied signal. In the case of a color monitor, the CRT typically contains separate guns to illuminate the red, green, and blue phosphors.

H

halftone: A process for converting a gray-scale image to black-and-white patterns of fine dots, the size or density of which in each small region corresponds to the intensity level of the original image in that region.

header: A preamble to a file or data structure that contains information about the contents of the file or data structure.

I

interlaced display: A raster display in which the displayed image consists of two fields of scan lines. Odd-numbered scan lines, which make up the odd field, are output at one time, and the even-numbered scan lines, which make up the even field, are output at another time. The lines of the two fields are interlaced on the display to form a single frame or image.

interrupt vector: A fixed 32-bit location in the TMS340 graphics processor's memory that contains the address of a trap or interrupt service routine. (Also see *trap vector*.)

ISR: Interrupt service routine.

K

kerning: Font metric; the amount by which a descender (such as the tail of a lower-case y) extends into the em space of the character to its left. (See *em*.)

L

leading: Font metric; the vertical spacing between the bottom of one line of text (as measured from the lowest descender in the font) to the top of the next line of text below it (as measured from the highest ascender in the font).

line-style pattern: A mask of 1s and 0s that are used to control the appearance of a styled line. As the line is drawn, the mask is rotated one bit per pixel, and the rightmost bit in the mask determines whether the next pixel is drawn in the foreground or the background color.

loader: A utility that loads an executable program or code module into a processor's memory; the loader typically causes the processor to begin executing the program.

long word: A 32-bit logical word in the TMS340 graphics processor's memory or a register.

LSB: Least significant bit. The lowest-order bit in a memory field or register.

M

magic number: The value contained in a field located at the beginning of a file or data structure that identifies the type and revision number of the file or data structure.

make utility: A utility program that automates program development. A make utility can update an executable file automatically whenever changes are made to one of its source or object files.

monospaced font: A font in which the character-to-character horizontal spacing is uniform for all characters.

MSB: Most significant bit. The highest-order bit in a memory field or register.

O

object file: A file that has been assembled or linked, and contains machine language object code that can be either relocatable or mapped to absolute addresses.

OEM: Original equipment manufacturer. A company that configures systems for resale.

off-screen memory: Frequently used to refer to the portion of the display memory that is not displayed and is therefore available for other uses, such as buffering data. The term is sometimes used to refer to non-display memory utilized for similar purposes.

off-screen workspace: A workspace in memory that has the same width and height as the screen but is only 1 bit per pixel.

on-screen memory: The portion of the display memory that is actually displayed; the memory comprising the video page or pages.

ordered dither: A halftoning algorithm, widely used in raster displays, that represents intensities as regular dot patterns of varying densities.

outcode: A 4-bit code that represents the position of a point relative to a rectangular clipping window. Refer to the user's guide for the TMS34010 or TMS34020 for details.

outline font: A computer representation of a font in which each character shape is specified as one or more filled regions bounded by curves.

P

palette: A digital lookup table used in a computer graphics display to translate the pixel values from the display memory into the red, green and blue components of the pixel as it is displayed.

pattern: (See *area-fill pattern*, *line-style pattern*.)

pen, drawing pen: A rectangular shape used within the graphics library to model a physical drawing pen or brush. In tracing the path of a line or curve, the area swept out by the pen is filled.

pitch: The difference in memory addresses from the start of one row of a two-dimensional pixel array to the next row of the array. This value is the same for each pair of adjacent rows in the array.

PixBlt: Pixel-aligned block transfer. A PixBlt operation copies a pixel array from one location in memory to another. The copy operation may involve combining each pair of corresponding source and destination pixels according to a Boolean or arithmetic operation. (Also see *BitBlt*.)

pixel: A picture element of a raster display device.
1) A physical pixel is the smallest individually controllable point of light on a CRT display.
2) In a bit-mapped display system, a logical pixel is the digital representation in memory of the attributes of the physical pixel to be displayed at the corresponding location on a CRT display.

pixel array: A two-dimensional array of pixels characterized by a base address, x and y extents, a pitch, and a depth (number of bits per pixel). A pixel array with a depth of 1 bit is often referred to as a bit map. A pixel array is sometimes referred to as a *pixel map* or *pixmap*.

pixel processing: The merging of a source pixel value with a destination pixel value according to a Boolean or arithmetic operation.

pixel-replicated format: The TMS340 graphics processor's architecture requires that the values loaded into the PMASK (plane mask), COLOR0 (background color), and COLOR1 (foreground color) registers be specified in pixel-replicated format. In this format, each *n*-bit pixel is replicated 32/*n* times through the length of the 32-bit register.

pixel size: The length of a logical pixel in bits. Also referred to as the *pixel depth*.

plane mask: A mask that specifies which bits in a pixel are protected from modification. This mask is the same number of bits in length as a pixel and affects all operations on pixels. The graphics library follows the convention that a mask bit that is a 1 designates a write-protected pixel bit, while pixel bits corresponding to 0s in the plane mask can be modified.

polyline: A set of connected lines. In the graphics library, a polyline is specified as a list of points, and a line is drawn between each pair of adjacent points in the list.

port: A device-specific implementation of a software product. All device-specific functions of the TMS340 Graphics Library are isolated in a small portion of the library software to facilitate the porting of the library to TMS34010- and TMS34020-based graphics systems.

proportionally spaced font: A font in which each character is permitted to vary in width from the others, and the spacing from one character to the next is dependent on the width of the character.

R

raster: A rectangular grid of picture elements (or pixels) whose colors and intensity levels are manipulated to represent images.

raster display: A display device with a flat surface consisting of a regular, two-dimensional grid of picture elements (or pixels), each of which is individually addressable.

rasterize: To convert a geometric shape such as a line, curve, or filled region from its mathematical representation to a set of pixels that represent the shape on a raster display. The accuracy of the representation is necessarily limited by the resolution of the display.

raster-op: Raster operation. Also referred to as a pixel processing operation. (See also *pixel processing*.)

resolution: The resolution of a raster display device, given in terms of the number of pixels (or dots) per unit length (measured in inches or millimeters). The width and height of a display monitor in pixels is frequently referred to informally as the "resolution" of the monitor.

RGB monitor: Red-green-blue monitor. This is a CRT monitor capable of displaying colors, and having separate inputs for the signals that drive the red, green, and blue guns of the CRT.

run-length encoding: An image-compression technique that encodes each scan line of an image as a series of color transitions. The color for each transition is paired with the number of pixels in the run prior to the next color transition.

S

scan line: A horizontal line output to a raster scan display. The term is also used to refer to a row of logical pixels in memory that are to be output to a particular scan line of the display.

screen coordinate system: The x–y coordinate system in which the absolute position of the pixels on the screen is specified. The screen origin is fixed in the top left corner of the screen. The x coordinates increase from left to right, and y coordinates from top to bottom.

screen origin: The x-y origin at the top left corner of the screen, which serves as an absolute reference point for referring to pixels on the screen.

screen refresh: The operation of streaming the contents of the display memory to the monitor in synchronization with the sweep of the electron

beam. In a video RAM system, a screen-refresh cycle typically occurs during the horizontal blanking interval that precedes each active scan line to download the logical pixels in the scan line to the video RAMs' serial registers.

SDB: Software Development Board. Texas Instruments provides SDBs for developing software to run on TMS34010- and TMS34020-based systems.

seed fill: A fill operation that fills a connected region of pixels on the screen with a solid color or pattern, beginning at a specified seed pixel. All pixels that are part of the connected region that includes the seed pixel are filled.

serial register: A register within a video RAM device that contains the data corresponding to a row in memory that is output from the serial port to refresh the screen. The data in the register may be serially clocked out at a rate independent of activity at the video RAM's random-access port.

source file: An ASCII text file that contains either C language or TMS340x0 assembly language source code that can be compiled or assembled to generate an object file.

stroke font: A computer representation of a font in which each character shape is specified as a series of line segments (or strokes).

styled line: A rasterized line that is drawn with a line-style pattern. Bresenham's algorithm is used to select a thin, but connected, set of pixels to represent the line. The color of each pixel in the set is governed by the corresponding bit in the line-style pattern. (See *line-style pattern*.)

symbolic debugger: A debugger utility with the capability of utilizing symbolic information retained during the compilation and linking processes.

system font: The graphics library's default font. This font is permanently installed as font 0 and is always a block font.

T

TDB: TMS34010 TIGA Development Board. This TMS34010-based PC add-in card from Texas Instruments contains a TMS34092 VGA Interface Chip, drives analog RGB monitors at resolutions of 640×480 and 1024×768, and supports pixel sizes of 1, 2, 4, and 8 bits.

TIGA, TIGA-340: Texas Instruments Graphics Architecture. A combined software and hardware standard for graphics systems defined by TI. TIGA is a standard approach to managing communications between the

PC host and the TMS340 graphics processor. At the core of TIGA is an interprocessor communications protocol that links an application or environment driver to a library of graphics functions that execute on the TMS340 processor.

TMS340: Product name for a family of graphics system processors and peripherals manufactured by Texas Instruments.

TMS34010: First-generation graphics system processor.

TMS34020: Second-generation graphics system processor.

TMS34070: A low-cost color palette device supporting displays with 4 bits per pixel.

TMS34082: Floating-point coprocessor that interfaces directly to the TMS34020 graphics processor.

TMS34092: TI's VGA Interface Chip. The TMS34092 is a memory and pixel pipeline peripheral for low-cost PC video adapters using the TMS34010. VGA pass-through capability is provided.

transparency: A pixel attribute which, when enabled, permits objects written to the screen to have transparent regions through which the original background pixels are preserved. Transparency is useful in graphics applications involving text, area-fill patterns, and pixel arrays in which only the shapes, and not the extraneous pixels surrounding them, are drawn to the screen.

trap vector: A fixed 32-bit location in the TMS340 graphics processor's memory that contains the address of a trap or interrupt service routine. (Also see *interrupt vector*.)

typeface, face: A collection of fonts that have common features such as style and weight, but which differ in size.

V

VGA: Video Graphics Array. A display adapter from IBM.

video page: The portion of display memory that contains an image that can be output to the monitor screen or other display surface.

video RAM, VRAM: Video random-access memory. A dual-ported dynamic memory device for computer graphics applications. One interface supports random read and write accesses to the memory; the other interface provides an independently clocked serial data stream to refresh a display.

W

word: A 16-bit logical word in GSP memory or a register.

workspace: (See *off-screen workspace*.)

X

XDS System: Extended Development Support System. Texas Instruments' XDS in-circuit emulation systems support the development of TMS34010- and TMS34020-based systems.

Z

zoom: To scale an image so that it is either magnified or reduced in size.

Index

R

S

T

V

X

Z

PERF

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

Technical Publications Manager
Texas Instruments Incorporated
P.O. Box 1443, MS640
Houston, Texas 77001

BINDING TAB

SCORE SCORE

(Fold and Staple Before Mailing)

BINDING TAB

REPLY CARD FOR 6.3 X 8.5 BOOK

PERF