TIGA Interface

User's Guide

September 1990 **Computer Video Products**

tiga Interface User's Guide

SPVU015C
September 1990

Printed on Recycled Paper

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.

Copyright © 1991, Texas Instruments Incorporated

Preface

Read This First

How to Use This Manual

This document contains the following chapters:

Chapter 1 **Introduction**
Introduces the TIGA Interface, its features and architecture.

Chapter 2 **Getting Started**
Contains instructions to install TIGA on a PC and to run a demonstration program.

Chapter 3 **Application Interface**
Describes the application interface and lists all TIGA functions by functional group.

Chapter 4 **Core Functions**
Describes the core functions alphabetically, showing the syntax and arguments of each function, a detailed description, and programming examples or references to related functions.

Chapter 5 **Extended Graphics Library Functions**
Describes the extended functions contained in the TIGA graphics library, showing the syntax and arguments of each function, a detailed description, and programming examples or references to related functions.

Chapter 6 **Graphics Library Conventions**
Describes the assumptions made and conventions adopted regarding coordinate systems, mapping of pixels to coordinates, operations on pixels, clipping, and the geometric figures and rendering styles supported by TIGA.

Chapter 7 **Bit-Mapped Text**
Describes the text capabilities of TIGA, the types of fonts supported, and the internal structure of the database for each font; gives an alphabetical listing of the available fonts with illustrations.

Chapter 8 **Extensibility**
Describes how to extend TIGA by adding your own functions; also describes the command processing entry points of the communication driver.

Appendix A **Data Structures**
Describes the data structures used in TIGA.

Appendix B **TIGA Reserved Symbols**
Describes the function names reserved for internal use of TIGA.

Appendix C **Debugger Support for TIGA**
Describes TIGA debugger support functions.

Appendix D **Error Messages / Error Codes**
Contains a list of error codes and messages returned by TIGA.

Appendix E **Glossary**
Contains the definitions of TIGA-specific and TIGA-related terms and acronyms.

Related Documentation

The following documents are available from Texas Instruments:

- *TMS34010 User's Guide* (literature number SPVU001).

  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.

To obtain any of TI's product literature listed above, please contact the Texas Instruments Customer Response Center at toll-free telephone number (800) 336-5236, or at (214) 995-6611 if you are calling from outside the US and Canada.

You may also find the following documentation useful:

- 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.

Style and Symbol Conventions

This document uses the following conventions:

- TIGA functions and their parameters are shown in *italic* face in regular text. For example, the TIGA function *draw_line* has parameters *x1, y1, x2, y2.* Filenames (example: *tigalnk.exe*) are also shown in *italic* face..
- Program examples and interactive display examples are shown in monospaced `program` font. Here is an example program:

```
#include <tiga.h>

main()
{
   short module;

   /*-------------------------------------------------------   */
   /* Initialize TIGA                                          */
   /*-------------------------------------------------------   */
   if (tiga_set(CD_OPEN) < 0)
   {
      printf("CD initialization error\n");
      exit(0);
   }

   if(!set_videomode(TIGA,INIT))
   {
      printf("TIGA GM initialization error\n");
      tiga_set(CD_CLOSE);
      exit(0);
   }
   /*-------------------------------------------------------   */
   /* Attempt to install module                                */
   /*-------------------------------------------------------   */
   if ((module=install_rlm("EXAMPLE")) < 0)
   {
      printf("Error %d installing EXAMPLE rlm\n", module);
      exit_tiga();
   }
   /*-------------------------------------------------------   */
   /* Main body of application appears here                   */
   /*-------------------------------------------------------   */
        :
        :
```

```
  /*-------------------------------------------------------   */
  /* Terminate TIGA                                            */
  /*-------------------------------------------------------   */
  exit_tiga();
}

exit_tiga()
{
  set_videomode(PREVIOUS, INIT);
  tiga_set(CD_CLOSE);
  exit(0);
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold** face font with parameters in *italic* face. Portions of a syntax in **bold** face (including quote marks) should be entered as shown. Portions of a syntax in *italic face* describe the type of information that you provide. Square brackets identify optional information:

mg2tiga *MG font* *TIGA font* [***"****facename****"***]

- *Italic* face is also used to emphasize a word or phrase that may be important or that is being defined or used for the first time in a chapter.

Trademarks

DGIS is a trademark of Graphic Software Systems, Inc.

GEM is a trademark of Digital Research, Inc.

GSS*CGI is a trademark of Graphic Software Systems, Inc.

High C and Metaware are trademarks of MetaWare Incorporated.

MS-Windows, PM, MS-DOS, and CodeView are trademarks of Microsoft Corp.

NDP C-386 and MicroWay are trademarks of Microway, Inc.

Phar Lap, 386|DOS-Extender, and 386|VMM are trademarks of Phar Lap Software, Inc.

PC-DOS and PGA are trademarks of IBM Corp.

TIGA is a trademark of Texas Instruments Incorporated.

Contents

1 Introduction ... **1-1**
- 1.1 Features ... 1-2
- 1.2 Architecture ... 1-3
- 1.3 Extensibility ... 1-5

2 Getting Started ... **2-1**
- 2.1 TMS340 Development Products ... 2-2
- 2.2 System Requirements ... 2-3
  - 2.2.1 TIGA Driver Development Package (TIGA-DDP) and TIGA Software Development Package (TIGA-SDP) ... 2-3
  - 2.2.2 TIGA Software Porting Package (TIGA-SPP) ... 2-3
- 2.3 Installing TIGA on Your System ... 2-4
  - 2.3.1 TIGA Driver Development Package (TIGA-DDP) Subdirectories ... 2-5
  - 2.3.2 TIGA Software Development Package (TIGA-SDP) Subdirectories ... 2-5
  - 2.3.3 TIGA Software Porting Package (TIGA-SPP) Subdirectories ... 2-5
- 2.4 Modifying Autoexec and the Environment ... 2-7
- 2.5 The TIGA Environment Variable ... 2-8
- 2.6 Running the TIGA Driver ... 2-9
- 2.7 TIGA Utility Programs ... 2-10
  - 2.7.1 cltiga Batch File ... 2-10
  - 2.7.2 mg2tiga Utility ... 2-11
- 2.8 Rebuilding Existing TIGA Applications for TIGA 2.0 ... 2-13
  - 2.8.1 TIGA 2.0 Initialization / Termination ... 2-13
  - 2.8.2 CURSOR Structure Change ... 2-13
  - 2.8.3 Return Value of set_config ... 2-14
  - 2.8.4 Elimination of Offscreen Workspace ... 2-15
  - 2.8.5 TIGA 1.1 Functions No Longer Supported ... 2-15
  - 2.8.6 New Functions Available in TIGA 2.0 ... 2-15
  - 2.8.7 Functional Differences in TIGA 2.0 ... 2-15

3 Application Interface ... **3-1**
- 3.1 Supported Development Tools ... 3-2
  - 3.1.1 Host-PC Development Tools ... 3-2
  - 3.1.2 TMS340 Development Tools ... 3-2
- 3.2 Host-PC Include Files and Libraries ... 3-3
- 3.3 TMS340 Include Files and Libraries ... 3-5

3.4 Sample TIGA Application 3-6
3.5 TIGA Functions 3-9
3.5.1 Core Functions 3-9
3.5.2 Extended Functions 3-9
3.6 Summary of Functions by Functional Group 3-10
3.6.1 Graphics System Initialization Functions 3-10
3.6.2 Clear Functions 3-10
3.6.3 Graphics Attribute Control Functions 3-11
3.6.4 Palette Functions 3-12
3.6.5 Graphics Drawing Functions 3-12
3.6.6 Poly Drawing Functions 3-13
3.6.7 Pixel Array Functions 3-14
3.6.8 Text Functions 3-15
3.6.9 Graphics Cursor Functions 3-15
3.6.10 Graphics Utility Functions 3-16
3.6.11 Handle-Based Memory Management Functions 3-16
3.6.12 Pointer-Based Memory Management Functions 3-17
3.6.13 Data Input/Output Functions 3-18
3.6.14 Extensibility Functions 3-18
3.6.15 Interrupt Handler Functions 3-19

4 Core Functions 4-1
4.1 Core Functions Reference 4-2

5 Extended Graphics Library Functions 5-1
5.1 Extended Graphics Library Functions 5-2

6 Graphics Library Conventions 6-1
6.1 Graphics Library Function Naming Conventions 6-2
6.2 Coordinate Systems 6-4
6.3 Area-Filling Conventions 6-6
6.4 Vector-Drawing Conventions 6-8
6.5 Rectangular Drawing Pen 6-10
6.6 Area-Fill Patterns 6-12
6.7 Line-Style Patterns 6-13
6.8 Operations on Pixels 6-15
6.8.1 Transparency 6-15
6.8.2 Plane Mask 6-16
6.8.3 Pixel-Processing Operations 6-16
6.9 Clipping Window 6-18
6.10 Pixel-Size Independence 6-19

7 Bit-Mapped Text 7-1
7.1 Bit-Mapped Font Parameters 7-2
7.2 Font Data Structure 7-5

7.2.1 Font Header Information 7-5
7.2.2 Font Pattern Table 7-8
7.2.3 Location Table 7-10
7.2.4 Offset/Width Table 7-10
7.3 Proportionally Spaced Versus Block Fonts 7-11
7.4 Font Table 7-12
7.5 Text Attributes 7-13
7.6 Available Fonts 7-14
7.6.1 Installing Fonts 7-15
7.6.2 Alphabetical Listing of Fonts 7-16

8 Extensibility 8-1
8.1 Dynamic Load Module 8-2
8.1.1 Relocatable Load Modules 8-2
8.1.2 Absolute Load Modules 8-2
8.2 Generating a Dynamic Load Module 8-4
8.2.1 TIGAEXT Section 8-4
8.2.2 The TIGAISR Section 8-4
8.2.3 Linking the Code and Special Sections Into an RLM 8-5
8.3 Installing a Dynamic Load Module 8-6
8.3.1 Installing a Relocatable Load Module 8-6
8.3.2 Installing an Absolute Load Module 8-7
8.4 Invoking Functions in a Dynamic Load Module 8-9
8.4.1 Command Number Format 8-9
8.4.2 Using Macros in Command Number Definitions 8-10
8.4.3 Passing Parameters to the TIGA Function 8-10
8.5 C-Packet Mode 8-12
8.5.1 The Type of Call 8-12
8.5.2 The Command Number 8-12
8.5.3 Description of Function Arguments 8-13
8.5.4 C-Packet Examples 8-14
8.5.5 Overflow of the Command Buffer 8-15
8.6 Direct Mode 8-16
8.6.1 Differences Between Microsoft C and High C/NDP Compilers 8-16
8.6.2 Standard Command Entry Point 8-17
8.6.3 Standard Command Entry Point With Return 8-18
8.6.4 Standard Memory Send Command Entry Point 8-19
8.6.5 Standard Memory Return Command Entry Point 8-20
8.6.6 Standard String Entry Point 8-21
8.6.7 Altered Memory Return Command Entry Point 8-21
8.6.8 Send/Return Memory Command Entry Point 8-21
8.6.9 Mixed Immediate and Pointer Command Entry Point 8-22
8.6.10 Mixed Immediate and Pointer Command Entry Point With Return 8-22
8.6.11 Poly Function Command 8-22

8.6.12 Immediate and Poly Data Entry Point . . . 8-25
8.7 Downloaded Function Restrictions . . . 8-28
8.7.1 Register Usage Conventions . . . 8-29
8.7.2 TIGA Graphics Manager System Parameters . . . 8-30
8.8 Using the TMS340-to-Host Callback Functions . . . 8-31
8.8.1 The Command Number . . . 8-31
8.8.2 Description of the Function Arguments . . . 8-31
8.8.3 Call-Back Examples . . . 8-32
8.8.4 Initializing the Callback Environment . . . 8-33
8.8.5 Sizing the Callback Buffer and Handling Overflow . . . 8-34
8.9 Installing Interrupts . . . 8-36
8.10 Object Code Compatibility . . . 8-39
8.10.1 Determining the Processor . . . 8-39
8.10.2 Pattern B-File Register . . . 8-40
8.10.3 Pitch Registers . . . 8-40
8.10.4 Video Timing Registers . . . 8-41
8.10.5 TM34020-Specific Instructions . . . 8-43
8.10.6 VRAM Block Mode . . . 8-43
8.11 The TIGA Linking Loader . . . 8-45
8.11.1 /ca – Create Absolute Load Module . . . 8-45
8.11.2 /cs – Create External Symbol Table . . . 8-46
8.11.3 /ec – Error Check . . . 8-46
8.11.4 /fs – Flush External Symbol Table . . . 8-47
8.11.5 /la – Load and Install an Absolute Load Module . . . 8-47
8.11.6 /lr – Load and Install a Relocatable Load Module . . . 8-47
8.11.7 /lx – Load and Execute a COFF File / Execute TIGA GM . . . 8-47

A Data Structures . . . A-1
A.1 Integral Data Types . . . A-2
A.2 CONFIG Structure . . . A-3
A.3 CURSOR Structure . . . A-5
A.4 ENVIRONMENT Structure . . . A-7
A.5 FONTINFO Structure . . . A-8
A.6 MODEINFO Structure . . . A-9
A.7 OFFSCREEN Structure . . . A-13
A.8 PALET Structure . . . A-14
A.9 PATTERN Structure . . . A-15

B TIGA Reserved Symbols . . . B-1
B.1 Reserved Functions . . . B-2
B.2 TIGA Core Functions Symbols . . . B-3
B.3 TIGA Extended Graphics Library Symbols . . . B-6

C Debugger Support for TIGA . . . C-1
C.1 Debugger Functions . . . C-2

C.2 TIGA / Debugger Interface C-12
C.3 Compatibility Functions C-14

D Error Messages / Error Codes D-1
D.1 Error Messages D-2
D.2 Error Codes D-3
D.3 Communication Driver (CD) Errors D-7

E Glossary E-1

Figures

1–1. Block Diagram 1-3
1–2. Function Configuration Options 1-5
4–1. Outcodes for Lines Endpoints 4-12
6–1. Screen Coordinates and Drawing Coordinates 6-4
6–2. Mapping of Pixels to Coordinate Grid 6-5
6–3. A Filled Rectangle 6-6
6–4. A Filled Polygon 6-7
6–5. An Outlined Polygon 6-9
6–6. A Line Drawn by a Pen 6-10
6–7. A 16-by-16 Area-Fill Pattern 6-12
6–8. Three Connected Styled Lines 6-14
7–1. Bit-Mapped Font Parameters 7-4
7–2. Data Structure for Bit-Mapped Fonts 7-5
7–3. Bit-Mapped Font Representation 7-9
8–1. Command Number Format 8-9
8–2. Data Structure of dm_cmd 8-18
8–3. Data Structure of dm_psnd 8-20
8–4. Data Structure of dm_poly 8-23

Tables

3–1. Include Files for PC Development . . . 3-3
3–2. AI Libraries Development Tools . . . 3-4
3–3. Include Files for TIGA Extended Function Development . . . 3-5
3–4. Graphics System Initialization Functions . . . 3-10
3–5. Clear Functions . . . 3-11
3–6. Graphics Attribute Control Functions . . . 3-11
3–7. Palette Functions . . . 3-12
3–8. Graphics Drawing Functions . . . 3-12
3–9. Poly Drawing Functions . . . 3-14
3–10. Pixel Array Functions . . . 3-14
3–11. Text Functions . . . 3-15
3–12. Graphics Cursor Functions . . . 3-15
3–13. Graphics Utility Functions . . . 3-16
3–14. Handle-Based Memory Management Functions . . . 3-17
3–15. Pointer-Based Memory Management Functions . . . 3-17
3–16. Data Input /Output Functions . . . 3-18
3–17. Extensibility Functions . . . 3-18
3–18. Interrupt Handler Functions . . . 3-19
4–1. Pixel-Processing Operations . . . 4-43
4–2. Pixel-Processing Operations . . . 4-120
6–1. Geometric Types . . . 6-2
6–2. Rendering Styles . . . 6-3
6–3. Checklist of Available Geometric Types and Rendering Styles . . . 6-3
6–4. Boolean Pixel-Processing Operation Codes . . . 6-17
6–5. Arithmetic Pixel-Processing Operation Codes . . . 6-17
7–1. Text-Related Functions . . . 7-2
7–2. Font Database Summary . . . 7-14
7–3. Installable Font Names . . . 7-15
8–1. Keyword Equivalent Types . . . 8-13
8–2. Trap Vectors . . . 8-36
8–3. Interrupt Service Routines . . . 8-37
8–4. Linking Loader Options . . . 8-45

Examples

8–1. Installation of the RLM example.rlm 8-6
8–2. Creation of an ALM From EXAMPLE.RLM 8-7
8–3. TMS340 Shell Routine With dm_poly 8-24
8–4. C Code to Determine the TMS340 Processor Type 8-39
8–5. Assembly Code to Determine the TMS340 Processor Type 8-40
8–6. Initialization of the CONVSP Register 8-41
8–7. Initialization of the Video Timing I/O Register Pointers 8-42
8–8. Use of TMS34020-Specific Instructions 8-43
8–9. Use of the VFILL Instruction 8-44

Chapter 1

Introduction

This user's guide describes TIGA (Texas Instruments Graphics Architecture), a software interface that standardizes communication between application software and all TMS340 family-based hardware for IBM-compatible personal computers. TIGA divides tasks between the TMS340 processor and the 80x86 host to improve application performance.

The TIGA interface standard simplifies the development of portable applications and application drivers for the diverse range of TMS340-based systems. TIGA's function set can be easily extended and customized by software developers for an application's specific needs. In addition, hardware developers can customize TIGA to take advantage of any value-added features available on the target TMS340-based board.

TIGA contains a low-level communication interface designed so that other standards such as MS-Windows, Presentation Manager (PM), DGIS, GEM, CGI, and PGA can run through the interface with no performance penalty. Essentially, TIGA replaces custom communication routines in other software interfaces with a single standard set of host-to-TMS340 communication routines.

Topics in this chapter include

Section **Page**

1.1 Features 1-2

1.2 Architecture 1-3

1.3 Extensibility 1-5

1.1 Features

These are the key application-related features of the TIGA interface standard:

Applications run faster TIGA provides the application developer with a dual-processor environment. This environment partitions the tasks in the application to run in parallel between the host and the TMS340 processors. The TIGA interface is optimized to provide high-speed communications between the host and the TMS340 family processors and to minimize the overhead in the processing of TIGA commands.

TMS340 family support TIGA supports the complete line of TMS340x0 graphics system processors from Texas Instruments. TIGA's functions take advantage of any available enhanced TMS340 processor instructions.

Dual-mode support TIGA supports both real- and protected-mode DOS applications.

Easy to use TIGA provides applications with a base set of graphics functions and with all the support required for the graphics subsystem. TIGA is compatible with a variety of popular DOS and extended DOS development tools.

Extensible When an application requires graphics functions that are not available in the TIGA base set of functions, you can develop user-extended functions by using TMS340 C, assembly language, or a mixture of the two. These extended graphics library functions can be downloaded at runtime during the application initialization.

Hardware independent Inquiry functions enable the application to determine the resolution, pixel size, etc., of the graphics subsystem and to adapt itself to the TMS340-based board on which it runs.

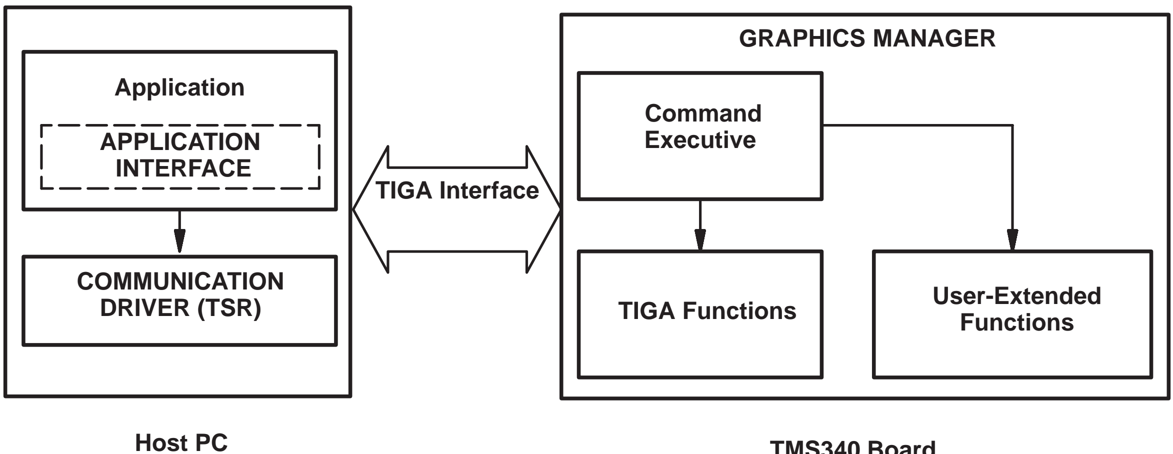
1.2 Architecture

Figure 1–1 shows a block diagram of the TIGA interface, illustrating the communication between the host routines and the TMS340 family processor routines.

As Figure 1–1 shows, the TIGA standard consists of four components:

1) Application Interface (AI)
a) Communication Driver (CD)
b) Graphics Manager (GM)
c) TIGA Extensions

Figure 1–1. Block Diagram



The application interface (AI) provides the communication path between a TIGA application and the TIGA communication driver. The AI consists of header files that reference TIGA function and type definitions, which may be used in the application, and of a library that the application links to when it is created. The AI does not actually contain the routines that interface to the TMS340 processor; these routines are contained in the communication driver.

The communication driver (CD) is a terminate-and-stay-resident (TSR) program that runs on a host PC. The CD is specific to the TMS340 board and is ported to it by a board manufacturer. A manufacturer ships the CD with the board; the CD is in a file called *tigacd.exe*. This file can be invoked directly from the command line or placed in the *autoexec.bat* file to be executed at startup. The CD contains the functions used to communicate between the host and the TMS340 board. The application invokes these functions via calls in the AI. These communication functions control the host side hardware-dependent

portion of TIGA, including whether the TMS340 board is memory-mapped or I/O-mapped.

The graphics manager (GM) is the portion of TIGA that runs on the TMS340 board and is specific to the board that it resides on. It consists of a command executive that controls the TMS340 side of the communications with the host, and of a set of core functions that provide memory management, palette support, and low-level graphics support. The GM typically resides in RAM on the TMS340 board (although this is not a requirement) and therefore must be loaded onto the board after power-up. The task of loading the GM is handled automatically by TIGA.

In addition to its core functions,TIGA also provides a set of functions to perform a wide range of graphics drawing operations. TIGA's functions can be extended by downloading additional, user-developed functions onto the TMS340 system. These downloaded functions may be written with either the TMS340 C or assembly language. Downloading functions can decrease the amount of processing required by the host and thus improve the performance of the application.

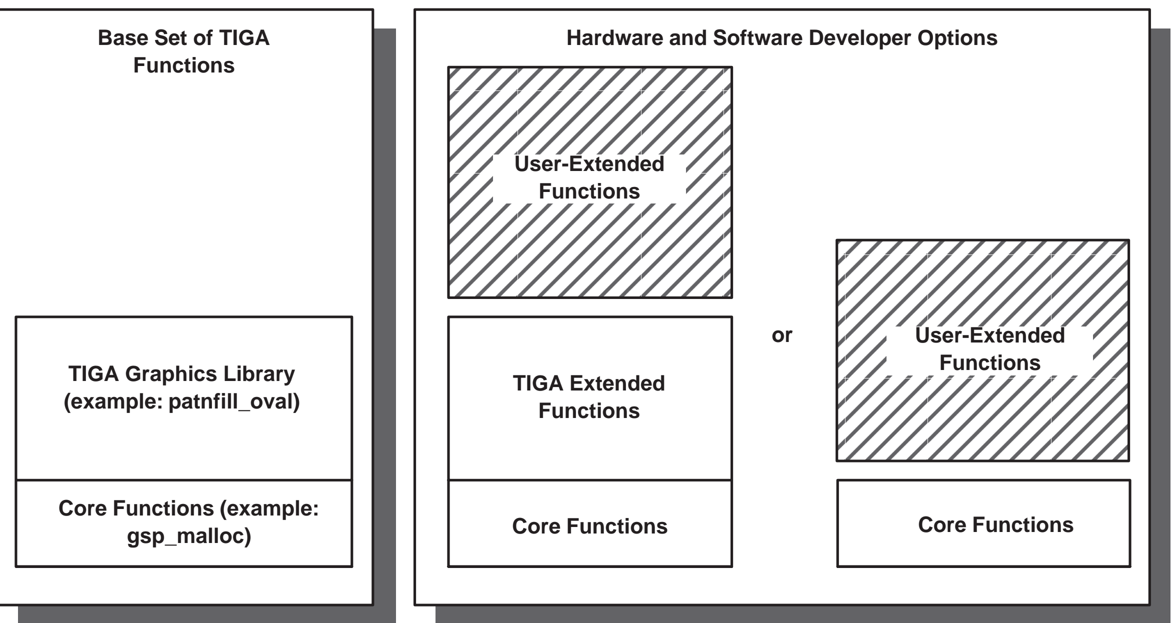
The host application invokes most of the TIGA functions on the TMS340 processor by downloading the parameters of the function, along with a command number, into one of several communication buffers. The command number is an identifier for the function to be executed. The command executive, which forms part of the GM, determines which function is to be invoked and calls it with the parameters that have been passed to it. Because there are several buffers, the host downloads data into one buffer while the TMS340 is executing data from another. This parallelism produces significant speed improvement over the host performing the graphics manipulation directly.

1.3 Extensibility

Graphics standards that existed prior to TIGA limited software development by providing a fixed set of graphics drawing functions. In the rapidly changing graphics market, a fixed set of functions is unacceptable.

TIGA's functions can be extended by adding or manipulating its user library collection of C-callable routines. Figure 1–2 shows the configuration options for TIGA functions.

Figure 1–2. Function Configuration Options



TIGA-compatible applications can be developed by using the base set of functions provided by TIGA (as shown in the left-hand side of Figure 1–2). These TIGA functions include the core functions, which are always available to the application, and the TIGA graphics library functions, which can be loaded if the application requires them.This set of graphics functions and the TMS340 processor give many applications an acceptable level of graphics performance. However, downloading user-extended functions can improve this performance still further. The user-extended functions can be downloaded to be used in addition to or instead of the TIGA graphics library (as shown on the right-hand side of Figure 1–2).

The same concept of adding functions can be implemented in hardware. For example, if you develop a TMS340-based graphics system and incorporate hardware in addition to the TMS340 processor, you can provide access to this

hardware through the TIGA interface by developing a set of user-extended functions that use the additional hardware functionality. Thus, the TIGA interface becomes a standard programming platform for the software written with these user-extended functions.

Chapter 2

Getting Started

This chapter contains instructions for installing TIGA on your system. Topics in this chapter include

Section		Page
2.1	TIGA Development Products	2-2
2.2	System Requirements	2-3
2.3	Installing TIGA on Your System	2-4
2.4	Modifying Autoexec and the Environment	2-7
2.5	The TIGA Environment Variable	2-8
2.6	Running the TIGA Driver	2-9
2.7	TIGA Utility Programs	2-10
2.8	Rebuilding Existing TIGA Applications for TIGA 2.0	2-13

2.1 TMS340 Development Products

Three TMS340 software development products are available from Texas Instruments:

- **DDK: TMS340 Driver Developer's Kit** (TI part number TMS340DDK-PC)

  The DDK is intended for software developers writing TIGA-compatible applications and drivers, for which no TIGA extension development is required. The TIGA Driver Development Package (TIGA-DDP), a subassembly of the DDK, contains the TIGA-related files and information.

- **SDK: TMS340 Software Developer's Kit** (TI part number TMS340SDK-PC)

  The SDK is intended for software developers writing TIGA-compatible applications and drivers, for which TIGA extension development is required. The TIGA Software Development Package (TIGA-SDP), a subassembly of the SDK, contains the TIGA-related files and information. Other subassemblies included with the SDK are the TMS340 Code Generation Tools and the TMS340 Graphics Library.

- **SPK: TMS340 Software Porting Kit** (TI part number TMS340SPK-PC)

  The SPK is intended for OEMs porting TIGA to their TMS340-based video hardware. The TIGA Software Porting Package (TIGA-SPP), a subassembly of the SPK, contains the TIGA-related files and information. Other subassemblies included with the SPK are the TMS340 Code Generation Tools and the TMS340 Graphics Library.

2.2 System Requirements

To ensure proper installation and operation of TIGA, your system must meet certain software and hardware minimum requirements. Consult the following sections for a list of these requirements, depending on the TIGA package you are installing.

2.2.1 TIGA Driver Development Package (TIGA-DDP) and TIGA Software Development Package (TIGA-SDP)

- IBM PC, XT, AT, or 100% compatible (hard disk required)
- 640K RAM
- TMS340-based TIGA 2.0-compliant video board with TIGA 2.0 driver
- MS-DOS or PC-DOS, version 2.13 or higher
- Microsoft Macro Assembler, version 5.0 or higher (if developing assembler-based applications/drivers)
- Microsoft C Compiler, version 5.0 or higher (if developing DOS real-mode C applications)
- MetaWare High C compiler, version 1.5 or 1.7, or Microway NDP C compiler, version 2.0 or higher, and Phar Lap 386 development tools, version 2.2 (if developing DOS protected-mode applications/drivers).
- TMS340 Family Code Generation Tools, version 4.0 or higher (if writing user-extended functions)

2.2.2 TIGA Software Porting Package (TIGA-SPP)

- IBM PC, XT, AT, or 100% compatible (hard disk required)
- 640K RAM
- TMS340-based video board (TIGA 2.0 target platform)
- MS-DOS or PC-DOS, version 2.13 or higher
- Microsoft Macro Assembler, version 5.0 or higher
- Microsoft C Compiler, version 5.0 or higher
- TMS340 Family Code Generation Tools, version 4.0 or higher

Note:

The TIGA-SPP provides complete porting sources to build and run TIGA 2.0 on a Texas Instruments TMS34010 or TMS34020 software development board, or on the TMS34010-based TIGA Development Board (TDB). In addition, all software necessary to port TIGA to a different TMS340 board is included in the TIGA-SPP; consult the Porting Guide in the file *\tiga\docs\portguid.doc* for information.

2.3 Installing TIGA on Your System

Note:

If you have an earlier version of TIGA on your system, be aware that the TIGA installation procedure overwrites same-named files in the *tiga* directory. For this reason, you should back up files of previous versions of TIGA, if needed, before proceeding with the new TIGA installation.

All TIGA development packages have an automated installation program with identical procedures to aid in installing TIGA on your system.

Follow these instructions to install your TIGA kit:

Step 1: Make backup copies of the product diskettes.

Step 2: Place diskette #1 (DDP #1, SDP #1, or SPP #1) of your TIGA kit into drive A.

Step 3: If A is not your current drive, at the MS-DOS prompt enter

```
A: ↵
```

Step 4: Make sure you are at the root directory of A. If you are not sure, enter this at the MS-DOS prompt:

```
cd\ ↵
```

Step 5: Enter

```
setup ↵
```

Step 6: Follow the instructions displayed on the screen to complete installation.

Note:

After installing TIGA, consult the *\tiga\docs\readme.1st* file for the latest information not included in this user's guide.

The installation of your TIGA development package creates a number of subdirectories on your destination drive. Consult one of the following three sections (depending on the TIGA development package you installed) for information describing these subdirectories and the files contained within them.

2.3.1 TIGA Driver Development Package (TIGA-DDP) Subdirectories

Installing the TIGA-DDP on your system creates the following subdirectories:

Subdirectory	Description
\tiga	TIGA root directory, TIGA drivers, system files, and utility programs
\tiga\demos	TIGA-compatible example programs
\tiga\docs	Additional TIGA-related documentation
\tiga\fonts	TIGA-compatible fonts
\tiga\include	Include files
\tiga\libs	Application interface libraries

2.3.2 TIGA Software Development Package (TIGA-SDP) Subdirectories

Installing the TIGA-SDP on your system creates the following subdirectories:

Subdirectory	Description
\tiga	TIGA root directory, TIGA drivers, system files, and utility programs
\tiga\demos	TIGA-compatible example programs
\tiga\docs	Additional TIGA-related documentation
\tiga\fonts	TIGA-compatible font files
\tiga\gm\extprims	TIGA-extended graphics library files
\tiga\include	Include files
\tiga\libs	Application interface libraries

The *\tiga\gm\extprims* directory contains the self-extracting archive file *extprims.exe*. This archive contains source for every extended graphics library function available within TIGA. It enables you to choose the extended functions you need, link them with your specific user extensions, and create a custom TIGA dynamic load module with the TMS340 functions that your application or driver requires. To extract the source files contained in this archive, enter *extprims* ↵ from within this directory.

2.3.3 TIGA Software Porting Package (TIGA-SPP) Subdirectories

Installing the TIGA-SPP on your system creates the following subdirectories:

Subdirectory	Description
\tiga	TIGA root directory, TIGA drivers, system files, and utility programs
\tiga\cd	Common TIGA communication driver (CD) files
\tiga\cd\cd34010	TMS34010 common CD files
\tiga\cd\cd34020	TMS34020 common CD files
\tiga\cd\tdb10	TIGA Development Board-specific CD files
\tiga\cd\sdb10	TMS34010 SDB-specific CD files

Subdirectory	Description
\tiga\cd\sdb20	TMS34020 SDB-specific CD files
\tiga\demos	TIGA-compatible example programs/drivers
\tiga\docs	Additional TIGA-related documentation
\tiga\fonts	TIGA-compatible font files
\tiga\gm	TIGA Graphics Manager (GM) files
\tiga\gm\corprims	TIGA core function files
\tiga\gm\extprims	TIGA-extended graphics library files
\tiga\gm\tdb10	TIGA Development Board-specific GM files
\tiga\gm\sdb10	TMS34010 SDB-specific GM files
\tiga\gm\sdb20	TMS34020 SDB-specific GM files
\tiga\include	Include files
\tiga\libs	Application interface (AI) libraries

Refer to the file *portguid.doc* in the *\tiga\docs* directory for detailed instructions on how to port TIGA to your TMS340-based video board.

2.4 Modifying Autoexec and the Environment

After installing your TIGA package, you will need to make a few modifications and/or additions to your *autoexec.bat* or comparable batch file. Note that these instructions use C: to identify the hard disk drive. Replace C: with the designator for the drive where you installed your particular TIGA package:

1) Append *C:\tiga* to the MS-DOS path:

```
PATH=existing PATH;C:\tiga ↵
```

2) If you plan to develop TIGA-compatible applications using the Microsoft C Compiler, append *C:\tiga\include* to the Microsoft C compiler environment variable INCLUDE:

```
set INCLUDE=existing INCLUDE;C:\tiga\include ↵
```

If you do not currently have an INCLUDE environment variable in your *autoexec.bat* file, this command adds it.

3) If you have the TMS340 Family Code Generation Tools installed on your system, then append *C:\tiga\include* to the existing A_DIR and C_DIR environment variables:

```
set A_DIR=existing A_DIR;C:\tiga\include ↵
set C_DIR=existing C_DIR;C:\tiga\include ↵
```

Again, if these environment variables currently do not exist, these commands add them.

4) Add the following TIGA environment variable:

```
set TIGA= –mC:\tiga –lC:\tiga –i0x60 ↵
```

See Section 2.5 for a complete description of the TIGA environment variable.

5) After modifying your *autoexec.bat* file, run it or reboot your PC.

2.5 The TIGA Environment Variable

TIGA uses the environment variable *TIGA* to get information about the location of TIGA system files, dynamic load modules, and the desired interrupt level. Set the TIGA environment variable by using the following syntax:

set TIGA = [*options*] [*string*] [*options*] [*string*]

set TIGA is the command that sets the environment

options valid options include

- **–m** specifies the path for TIGA system files
- **–l** specifies the path for TIGA dynamic load user modules
- **–i** specifies the host interrupt level used by the TIGA communication driver

The option string cannot contain the character '–'. In addition, there should be no spaces between the option and the option string. For example:

`set TIGA=–mc:\tiga` is correct

`set TIGA=–m c:\tiga` is incorrect

Also, a space is required between options. For example:

`set TIGA=–mc:\tiga –lc:\tiga` is correct

`set TIGA=–mc:\tiga–lc:\tiga` is incorrect

When TIGA is initially installed, all TIGA system files are placed in the TIGA directory of the destination drive. Specify this path with the *–m* option of the TIGA environment variable.

Any dynamic load modules loaded from a TIGA application must be located in either

- the current directory from which the TIGA application is called, or
- the path specified by the *–l* option in the TIGA environment variable.

By default, TIGA's communication driver uses interrupt level 0x7F to communicate with an application. Use the *–i* option followed by the interrupt level (in hex format) in the TIGA environment variable to specify an alternate interrupt level.

Note:

The TIGA interrupt level must be set below 0x70 for the TIGA CD to operate properly with the Phar Lap utilities (that is, applications linked with the MetaWare High C or NDP C compilers). TI recommends using 0x60.

For example, assume that all TIGA system files are located in *C:\tiga*, that user dynamic load modules are in *D:\dlm*, and that the desired interrupt level to use is 0x60. Set the corresponding TIGA environment variable:

```
set TIGA=–mc:\tiga –ld:\dlm –i0x60 ↵
```

2.6 Running the TIGA Driver

This section provides general instructions on how to load the TIGA communication driver. Consult the TIGA software installation instructions that accompany your TIGA video board for specific loading information.

To load TIGA, enter this at the MS-DOS prompt:

tigacd [*options*]

tigacd is the command that invokes the TIGA communication driver (CD).

options valid options include

- **–i** Reinstalls the TSR. This option forces a new copy of the TIGA communication driver to be loaded in memory, thereby superseding any previously installed CD. Note that reinstalling the TSR with the *–i* option forces reloading of the TIGA graphics manager.
- **–u** Uninstalls the TSR. This option causes the previously installed TIGA CD to be released from memory, disabling TIGA. To re-enable TIGA, enter *tigacd* ↵ once again.

The following options control TIGA's debugger facilities. For detailed information on how to debug a TIGA application, refer to the *TMS340 Family C Source Debugger* user's guide.

- **–d0** Disables the TIGA debug facility.
- **–d1** Enables the emulator mode of TIGA's debug facility. This option, used in conjunction with the emulator version of the TMS340 C source debugger and the XDS500 emulator, provides C source level debug capability for TIGA applications.
- **–d2** Enables the development board mode of the TIGA debug facility. This option, used in conjunction with the serial link configuration of the TMS340 C source debugger, provides C-source-level debug capability for TIGA applications.

After the TIGA CD is loaded, TIGA is ready to use; however, the TMS340 side of TIGA has not yet been initialized. This is accomplished by an application calling *set_videomode(TIGA,INIT)* to check whether the TIGA graphics manager (GM) is loaded and running on the TMS340 side. If so, both the host and TMS340 sides of TIGA are ready. If not, the GM is loaded, executed, and initialized before returning from the *set_videomode* function.

After you load the host and TMS340 sides of TIGA, your application is free to call TIGA's core functions.

2.7 TIGA Utility Programs

The following TIGA utility programs are in TIGA's root directory *\tiga* to simplify porting and/or applications development:

TIGA Utility	Description
cltiga.bat	Batch file that uses Microsoft C tools to compile and link a TIGA application.
hcc.bat	Batch file that invokes the MetaWare High C Compiler, compiling the specified source file.
hcl.bat	Batch file that invokes the Phar Lap linker, linking the object code and optional user library with the MetaWare High C version of the TIGA AI library (*hcai.lib*).
make.exe	Texas Instruments program maintenance utility. It is fully compatible with the Microsoft *make.exe* utility and has additional features. Consult the file *\tiga\docs\make.doc* for detailed usage information.
oldap.exe	This utility restores the TIGA environment back to a known, stable state. There is always the potential of a TIGA 1.1 application prematurely aborting and not properly restoring the TIGA environment. Run *oldap.exe* anytime you encounter timeout errors while running TIGA applications under TIGA 2.0.
mg2tiga.exe	Utility to convert TMS340 math/graphics fonts to TIGA-compatible fonts.
tigamode.exe	Utility to query available modes and select default mode.

2.7.1 cltiga Batch File

The *cltiga.bat* batch file easily compiles and links a TIGA-compatible application (contained in a single C source file) to the TIGA application interface library *ai.lib,* using the Microsoft C compiler. It also supports symbolic debugging through Microsoft's CodeView debugger. The syntax for *cltiga* is

cltiga [–d] *filename*

where:

cltiga is a batch file to compile and link a TIGA application,

–d is an option that specifies symbolic debug processing, and

filename is the name of the C file to be processed. No extension should be specified on the filename.

Note:

The TIGA application interface library *ai.lib* is independent of the Microsoft C model. However, the *cltiga* batch file uses the large model by default. You can override the default by modifying the *cltiga.bat* file(consult the Microsoft C reference manual for details).

2.7.2 mg2tiga Utility

The *mg2tiga* utility converts fonts compatible with the TMS340 Graphics Library to a format compatible with the TIGA text functions. To invoke the *mg2tiga* utility, enter

mg2tiga *MG font* *TIGA font* [*"facename"*]

mg2tiga is the command to invoke the *mg2tiga.exe* utility.

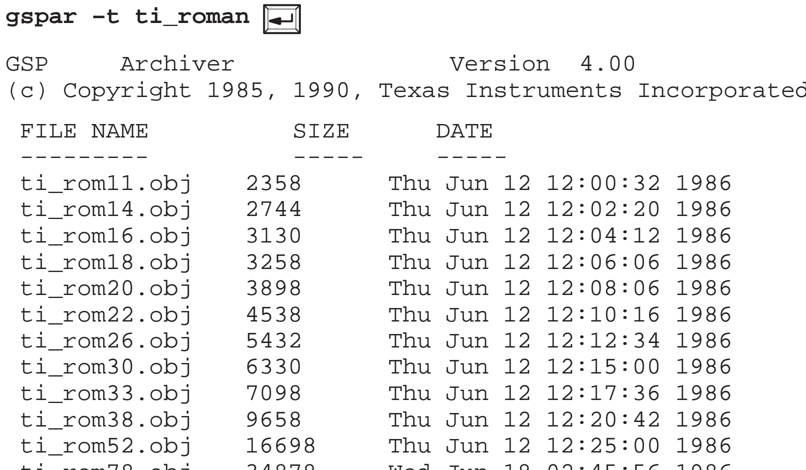
MG font is a binary or COFF object image of a math/graphics compatible font.

TIGA font is the filename under which the converted font is saved.

facename is an optional name of the font (up to 29 characters long) enclosed within double quotes. If this parameter is not specified on the command line, *mg2tiga* prompts you for it.

Here is an example of converting the TI Roman 18-point font from the math/graphics font library to TIGA format.

1) Locate the library that contains TI Roman fonts. As supplied, this library is called *ti_roman.lib* and contains 12 fonts. To display a table of contents of this library, enter



```
gspar -t ti_roman ↵

GSP     Archiver               Version  4.00
(c) Copyright 1985, 1990, Texas Instruments Incorporated

 FILE NAME          SIZE      DATE
 ---------          -----     -----
 ti_rom11.obj    2358      Thu Jun 12 12:00:32 1986
 ti_rom14.obj    2744      Thu Jun 12 12:02:20 1986
 ti_rom16.obj    3130      Thu Jun 12 12:04:12 1986
 ti_rom18.obj    3258      Thu Jun 12 12:06:06 1986
 ti_rom20.obj    3898      Thu Jun 12 12:08:06 1986
 ti_rom22.obj    4538      Thu Jun 12 12:10:16 1986
 ti_rom26.obj    5432      Thu Jun 12 12:12:34 1986
 ti_rom30.obj    6330      Thu Jun 12 12:15:00 1986
 ti_rom33.obj    7098      Thu Jun 12 12:17:36 1986
 ti_rom38.obj    9658      Thu Jun 12 12:20:42 1986
 ti_rom52.obj    16698     Thu Jun 12 12:25:00 1986
 ti_rom78.obj    34878     Wed Jun 18 02:45:56 1986
```

2) Extract the desired font — in this case, *ti_rom18.obj*.

   Example: `gspar x ti_roman ti_rom18.obj` ↵

3) Now use `mg2tiga` to convert it to TIGA format.

   Example: `mg2tiga ti_rom18.obj roman18.fnt` ↵

   At this point, *mg2tiga* prompts you to enter a facename for the font. This facename can be up to 29 characters long and should be the name of the font.

Example:

```
MGFL to TIGA font converter
Enter facename (29 chars max): TI ROMAN ↵
```

After you have entered the facename, *mg2tiga* displays the MG font header and then the new TIGA font header. A prompt to press ↵ follows each of these displays. After you enter this information, the conversion is complete.

```
         [ -------- Old Font Header -------- ]
fonttype:  9000
firstchar: 0000
lastchar:  00ff
widemax:   0010
kernmax:   0000
ndescent:  fffd
charhigh:  0011
owtloc:    046a
ascent:    000e
descent:   0003
leading:   0002
rowwords:  0033
[ Press return ]->
         [ -------- New Font Header -------- ]
magic:     8040
length:    00000b18
facename:  TI ROMAN
first:     0000
last:      00ff
deflt:     0000
maxwide:   0010
maxkern:   0000
charwide:  0000
avgwide:   0008
charhigh:  0011
ascent:    000e
descent:   0003
leading:   0002
rowpitch:  00000330
oPatnTbl:  00000250
oLocTbl:   00003880
oOwTbl:    000048a0
[ Press return ]->
```

2.8 Rebuilding Existing TIGA Applications for TIGA 2.0

Any TIGA application or driver linked with the TIGA 1.1 Application Interface (AI) library will run unchanged under TIGA 2.0. However, if you relink your existing TIGA 1.1 compatible application or driver with one of the TIGA 2.0 AI libraries, several modifications to your source code will be required. This section outlines the modifications required to upgrade existing TIGA 1.1 applications to link with TIGA 2.0 libraries.

2.8.1 TIGA 2.0 Initialization / Termination

A new method of initializing and terminating a TIGA application has been added to TIGA 2.0. This was required to support TIGA's ability to run both real- and protected-mode DOS applications.

TIGA 1.1 initialization was accomplished by calling one of the following two functions:

- cd_is_alive()
- set_videomode(TIGA, style)

Both establish communications with the TIGA communication driver. In addition, the *set_videomode* function loads the TIGA graphics manager if specified to do so by the *style* argument.

A new function, *tiga_set*, has been added to TIGA 2.0 for properly initializing and terminating the TIGA environment from an application. This function replaces the *cd_is_alive* function formerly used in TIGA 1.1. In addition, the TIGA 2.0 version of the *set_videomode* function only switches video modes and does not initialize communications.

TIGA 2.0 applications must call *tiga_set(CD_OPEN)* before calling any other TIGA function and must call *tiga_set(CD_CLOSE)* before returning to DOS. This will insure that the TIGA environment is properly maintained. Refer to the sample TIGA 2.0 application listing in Section 3.4, page 3-6, and to the *tiga_set* function description in Chapter 4, page 4-136, for more information.

2.8.2 CURSOR Structure Change

TIGA's CURSOR structure has been modified to support two-color cursors. TIGA 1.1 allowed only the cursor shape color to be modified. TIGA 2.0 now allows both the cursor mask and shape colors to be specified. Any existing source code using the TIGA CURSOR structure must be modified to conform to the TIGA 2.0 CURSOR structure definition.

The TIGA 1.1 CURSOR structure was defined as

```
typedef struct
{
   short hot_x;                /* hotspot offset from top                    */
   short hot_y;                /* left-hand corner                           */
   unsigned short width;       /* cursor width (in bits)                     */
   unsigned short height;      /* cursor height (lines)                      */
   unsigned short pitch;       /* pitch of cursor data                       */
   unsigned long color;        /* cursor shape color                         */
   unsigned short mask_rop;    /* cursor mask rop                            */
   unsigned short shape_rop;   /* cursor shape rop                           */
   PTR data;                   /* pointer to cursor data in TMS340 memory    */
}CURSOR;
```

The TIGA 2.0 CURSOR structure has an additional field, *mask_color*, along with transparency support added to the rop fields:

```
typedef struct
{
   short hot_x;                /* hotspot offset from top                    */
   short hot_y;                /* left-hand corner                           */
   unsigned short width;       /* cursor width (in bits)                     */
   unsigned short height;      /* cursor height (lines)                      */
   unsigned short pitch;       /* pitch of cursor data                       */
   unsigned long color;        /* cursor shape color                         */
   unsigned short mask_rop;    /* cursor mask rop                            */
   unsigned short shape_rop;   /* cursor shape rop                           */
   unsigned long mask_color;   /* cursor mask color                          */
   PTR data;                   /* pointer to cursor data in TMS340 memory    */
}CURSOR;
```

Consult the *set_curs_shape* and *set_cursattr* function descriptions in Chapter 4 for additional information.

2.8.3 Return Value of set_config

The TIGA 2.0 Graphics Manager (GM) is relocatable to support varying memory maps. Specifying a particular graphics mode with the *set_config* function may cause the TMS340 memory map to change, which in turn could force the GM to be reloaded. This may have undesirable results, since reloading the GM causes all downloaded extensions to be flushed and all allocated memory to be freed.

The return value of the TIGA 2.0 *set_config* function can be used by an application to query whether the GM was reloaded. If so, the application can take the appropriate steps necessary to return the TMS340 environment back to a known state. However, it is recommended that any application that calls *set_config* does so before downloading any TIGA extensions or allocating any memory. See the *set_config* function description, page 4-100, for more information.

2.8.4 Elimination of Offscreen Workspace

The TIGA 1.1 *fill_polygon* and *patnfill_polygon* functions both required the use of a one-bit-per-pixel offscreen workspace to operate properly. In TIGA 2.0, these functions have been modified to remove this restriction. You may wish to remove existing code used to initialize the offscreen workspace area; it is no longer required by any function in TIGA 2.0.

2.8.5 TIGA 1.1 Functions No Longer Supported

Only one function previously available in TIGA 1.1 is no longer supported in TIGA 2.0. It is the *cd_is_alive* function. Consult subsection 2.8.1 for additional information.

2.8.6 New Functions Available in TIGA 2.0

TIGA 2.0 provides a variety of new functions, which are listed in this section. Also, many existing TIGA 1.1 functions have been modified to provide additional functionality. For further information, consult the appropriate function's description in Chapter 4, for core functions, and in Chapter 5, for extended graphics library functions.

- New TIGA 2.0 core functions:

aux_command	gsph_init
cvxyl	gsph_maxheap
flush_module	gsph_memtype
get_text_xy	gsph_realloc
gm_idlefunction	gsph_totalfree
gsph_alloc	setup_hostcmd
gsph_calloc	set_cursattr
gsph_compact	set_module_state
gsph_deref	set_text_xy
gsph_falloc	sym_flush
gsph_fcalloc	text_outp
gsph_findhandle	tiga_busy
gsph_findmem	tiga_set
gsph_free	

- New TIGA 2.0 extended graphics library functions:

decode_rect	put_pixel
encode_rect	styled_oval
get_pixel	styled_ovalarc
in_font	styled_piearc
move_pixel	

2.8.7 Functional Differences in TIGA 2.0

get_pmask — The TIGA 2.0 implementation returns the value of the plane mask register right-justified (and zero-extended) in

	the *N* LSBs of the return value, where *N* is the current pixel size. The TIGA 1.1 implementation of *get_pmask* simply returned the current 32-bit value in the plane mask register.
gsp_minit	The stack size argument supplied to the *gsp_minit* function is ignored in TIGA 2.0 because of limitations imposed by the new TIGA 2.0 memory manager. Note however, that an argument must still be specified for the function.
page_flip	The TIGA 1.1 implementation of *page_flip* returned zero if an invalid drawing or display page argument was specified, and nonzero if the specified arguments were valid. The TIGA 2.0 implementation of *page_flip* does not return a value and treats invalid display and/or drawing page arguments as if the function were called as *page_flip(0,0).*
set_pmask	The TIGA 2.0 implementation of *set_pmask* automatically replicates the right-justified *N* LSBs (where N is the current pixel size) of the mask argument, throughout the 32-bit plane mask register. The TIGA 1.1 implementation expected the mask argument to be already replicated.
graphics cursors	The TIGA 2.0 cursor generator automatically saves and restores the current PMASK register, then clears it to zero (enables all bit planes) before saving the cursor background, drawing the cursor, and restoring the original background. Earlier versions of the cursor generator simply used whatever value was currently set in PMASK.

Chapter 3

Application Interface

TIGA consists of a set of functions that a host-PC application can invoke to perform a variety of graphics-related tasks (a host-PC application is defined as that portion of the application that is running on the host-PC processor). These functions may run entirely on the host-PC or the TMS340 board, or they may execute in parallel on both processors.

TIGA also gives the application developer the capability to create functions that are downloaded to the TMS340-based target board by the host-PC application. Because these functions are not part of the standard TIGA core functions, and because they reside and run on a TMS340-based board, they are commonly referred to as *extended functions*. Once these extended functions are loaded, the host-PC application may call them at any time. In addition, an extended function may call a core function, a previously loaded extended function, or even a function residing on the host-PC side of the application. Detailed information on how to create, load, and call TMS340 extended functions is presented in Chapter 8.

This chapter discusses basic information required to develop a TIGA-compatible application. It also lists the TIGA functions in their functional groups.

Topics in this chapter include

Section **Page**

3.1 Supported Development Tools 3-2
3.2 Host-PC Include Files and Libraries 3-3
3.3 TMS340 Include Files and Libraries 3-5
3.4 Sample TIGA Application 3-6
3.5 TIGA Functions 3-11
3.6 Summary of Functions by Functional Group 3-12

3.1 Supported Development Tools

The following development tools are currently supported by TIGA 2.0.

Note:

Consult the *\tiga\docs\readme.1st* file for information describing any additional development tools supported by TIGA 2.0.

3.1.1 Host-PC Development Tools

Compiler/Assembler	Version	Operating System
Microsoft C Compiler	5.0 or higher	DOS (real mode)
Microsoft Macro Assembler	5.0 or higher	DOS (real mode)
MetaWare High-C Compiler	1.5 or 1.7	Phar Lap DOS Extender v 2.2
Microway NDP C-386 Compiler	2.0 or higher	Phar Lap DOS Extender v 2.2

Note:

The examples in this user's guide are intended to be built with the Microsoft C Compiler. Minor modifications may be required to build the examples with any of the other supported development tools listed above.

3.1.2 TMS340 Development Tools

The TMS340 Family Code Generation Tools, version 3.0 or higher, are used to build TMS340 extended functions. You will need these tools only if you are planning on writing your own extended functions.

3.2 host-PC Include Files and Libraries

Three types of TIGA includes files are used for developing host-PC TIGA-compatible applications:

tiga.* Contains TIGA constants commonly required by a TIGA application and core function references. Every TIGA host-PC application *must* include and specify this file before any other TIGA include file.

extend.* Contains references used for the TIGA 2-D Graphics Library. Any application calling a function from this TMS340 extended library must include this file.

typedefs.* Contains references for TIGA structures. Any application using a TIGA structure type must include this file.

These include files are located in the *\tiga\include* directory. Different versions of each include file type are provided for each set of supported development tools. Table 3–1 summarizes the include files provided in TIGA 2.0 for host-PC code development:

Table 3–1. Include Files for PC Development

	Microway NDP C-386	**MetaWare High C 386**	**Microsoft C (all models)**	**Microsoft Assembler**
tiga.*	tiga.ndp	tiga.hch	tiga.h	tiga.inc, tiga_sm.inc
extend.*	extend.pl	extend.pl	extend.h	extend.inc
typedefs.*	typedefs.pl	typedefs.pl	typedefs.h	typedefs.inc

After you compile and/or assemble the host-PC source program, you must link the derivative object file(s) with the appropriate TIGA Application Interface (AI) library. The functions within this library provide the communications interface between the application and the TIGA Communication Driver (CD).

All TIGA AI libraries are located in the *\tiga\libs* directory. Different versions of the AI library are provided for each set of supported development tools. Table 3–2 summarizes the AI libraries provided in TIGA 2.0 for linking host-PC developed code. In addition,Table 3–2 lists the code/data referencing type, memory models, and compatible development tools for each TIGA AI library.

Table 3–2. AI Libraries Development Tools

TIGA AI Library	Description	Development Tools	Memory Models
ai.lib	Far code references, Far data references	Microsoft C 5.0 or higher, Microsoft Macro Assembler	small, medium, compact, large, huge
ai_com.lib	Near code references, Near data references	Microsoft C 5.0 or higher, Microsoft Macro Assembler	.com
hcai.lib	Near code references, Near data references	MetaWare High C	not applicable
ndpai.lib	Near code references, Near data references	Microway NDP C	not applicable

3.3 TMS340 Include Files and Libraries

A standard set of include files supports development of extended functions for TIGA. TIGA extensions may be developed in C or TMS340 assembly code. See Chapter 8 for more information on how to develop TMS340 extensions to TIGA.

The TIGA TMS340 include files and libraries are identified by the leading characters *gsp* (Graphics System Processor). In addition to the three types of include files described in Section 3.2, three other include file types are provided for developing TIGA extended functions:

gspglobs.* Contains references to TIGA global variables, arrays, and structures. Include this file if your TMS340 extended functions reference any TIGA variable, array, or structure.

gspreg.* Contains equated constants for all TMS34010 and TMS34020 processor registers.

gspmac.lib Contains macros useful for developing TMS340 assembly code. Consult the *\tiga\docs\gspmac.doc* file for a description of the macros in this library.

Table 3–3 summarizes the include files and libraries provided in TIGA 2.0 for TMS340 extended function development:

Table 3–3. Include Files for TIGA Extended Function Development

	TMS340 C	**TMS340 Assembler**
gsptiga.*	gsptiga.h	gsptiga.inc
gspextnd.*	gspextnd.h	gspextnd.inc
gsptypes.*	gsptypes.h	gsptypes.inc
gspglobs.*	gspglobs.h	gspglobs.inc
gspreg.*	gspreg.h	gspreg.inc
gspmac.lib	–	gspmac.lib

3.4 Sample TIGA Application

This section describes the basic components of a TIGA application. In general, all TIGA applications contain the following:

- Initialization
- Load TIGA extended functions (optional)
- Main body
- Termination

The initialization establishes the communications link between the application and the TIGA communication driver. Also, correct operation of the TIGA graphics manager is verified. Initialization is extremely important to ensure a stable TIGA environment for your application.

After initialization is complete, the application may call any of TIGA's core functions. However, this set of functions may be insufficient for your application. Additional required functions are normally loaded into TIGA following initialization. These extended functions may include TIGA's graphics library and/or any other required extended function library.

Once all functions are loaded into TIGA, the application can call them at any time throughout the main body of the application.

Finally, the TIGA environment must be properly terminated. This step restores the video environment to the state it was in before you executed the TIGA application.

The following example TIGA application illustrates how to set up the TIGA environment, load extended functions, and then terminate the TIGA environment.

```
/*******************************************************************************/
/*     Sample TIGA 2.0 application                                             */
/*                                                                             */
/*     This example illustrates the basic components of a TIGA app:            */
/*                                                                             */
/*         - Initialization                                                    */
/*         - Loading extended functions                                        */
/*         - Main body                                                         */
/*         - Termination                                                       */
/*                                                                             */
/*     This example is intended to be built with the Microsoft 'C'             */
/*     compiler.                                                               */
/*******************************************************************************/
#include <tiga.h>              /* All apps MUST include this file              */
#include <extend.h>            /* We are going to call an ext func             */
#include <typedefs.h>          /* We may want to ref a TIGA struct             */

short oldmode;                 /* Storage for old videomode                    */
```

```
/*-----------------------------------------------------------------------------*/
/*     term_tiga                                                               */
/*                                                                             */
/*     syntax void term_tiga(void)                                             */
/*                                                                             */
/*     This function properly terminates a TIGA application by                 */
/*     restoring the previous video mode and closing the TIGA CD.              */
/*     It must be    called prior to returning to DOS.                         */
/*-----------------------------------------------------------------------------*/
void term_tiga()
{
   printf("Press any key to return to DOS...");
   getch();
   set_videomode(oldmode,INIT);  /* Return mode to prev state                  */
   tiga_set(CD_CLOSE);           /* Close the TIGA CD                          */
   exit(0);                      /* Exit back to DOS                           */
}
/*-----------------------------------------------------------------------------*/
/*     init_tiga                                                               */
/*                                                                             */
/*     syntax    void init_tiga(load_graphics_lib)                             */
/*               short load_graphics_lib;                                      */
/*                                                                             */
/*     This function properly initializes the TIGA environment                 */
/*     and loads the graphics library functions if argument                    */
/*     load_graphics_lib is non-zero. This function should be                  */
/*     called prior to calling any other TIGA function.                        */
/*-----------------------------------------------------------------------------*/
void init_tiga(load_graphics_lib)
short load_graphics_lib;
{
   short v;
   long lv;
   /*------------------------------------------------------------------------- *
/
   /*     Open TIGA Communications Driver                                      */
   /*------------------------------------------------------------------------- *
/
   if((lv = tiga_set(CD_OPEN)) < 0L)
   {
      printf("TIGA CD error: %ld\n", lv);
      exit(0);                   /* Exit back to DOS                           */
   }
```

```
   /*---------------------------------------------------------------------------- *
/
   /*     Go into TIGA mode                                                      */
   /*---------------------------------------------------------------------------- *
/
   oldmode = get_videomode();    /* Save current videomode for later            */
   if(!(v = set_videomode(TIGA,INIT | CLR_SCREEN)))
   {
      printf(”TIGA GM error: %d\n”, v);
      tiga_set(CD_CLOSE);     /* Be sure to close the open TIGA CD              */
      exit(0);                /* before exiting to DOS                          */
   }
   /*---------------------------------------------------------------------------- *
/
   /* Load graphics library functions if specified to do so                  */
   /*---------------------------------------------------------------------------- *
/
   if(load_graphics_lib && (v=install_primitives()) < 0)
   {
      printf( ”Graphics Library load error: %d\n”, v );
      term_tiga();
   }
}
```

```
/*-----------------------------------------------------------------------------------------*/
/*     Main program.                                                                       */
/*-----------------------------------------------------------------------------------------*/
main()
{
   CONFIG config;                /* Storage for TIGA config struct                */
   short width,height,xleft,ytop;    /* fill_rect() arguments
*/

   /*---------------------------------------------------------------------------- *
/
   /* Call init_tiga() to initialize the TIGA environment. Also,                 */
   /* load the Graphics Library functions.                                       */
   /*---------------------------------------------------------------------------- *
/
   init_tiga(1);

   /*---------------------------------------------------------------------------- *
/
   /* Load extended functions section (OPTIONAL)                                 */
   /*                                                                            */
   /* At this point, the TIGA environment has been properly                      */
   /* initialized. This is a good time to load any other extended                */
   /* functions required by the application.                                     */
   /*---------------------------------------------------------------------------- *
/
                    :
                    :
   /*---------------------------------------------------------------------------- *
/
   /* Main body                                                                  */
   /*                                                                            */
   /* All functions are now loaded. We can now safely call any of                */
   /* these functions.                                                           */
   /*                                                                            */
   /* As an example, let's draw a blue, solid filled rectangle,                  */
   /* half the size of the screen, centered in the screen.                       */
   /*---------------------------------------------------------------------------- *
/

   get_config(&config);                    /* Get info on current mode          */
   width = config.mode.disp_hres >> 1;  /* Width 1/2 screen width             */
   height = config.mode.disp_vres >> 1; /* Height 1/2 screen height           */
   xleft = width >> 1;                  /* Center rect in middle              */
   ytop = height >> 1;                  /* of screen                          */
   set_fcolor(BLUE);                    /* Set foreground color               */
   fill_rect(width,height,xleft,ytop);  /* Fill the rectangle                 */
                    :
                    :
                    :
   term_tiga();                         /* Properly terminate TIGA            */
}
```

All example programs provided with the function descriptions in this user's guide use similar methods to properly initialize and terminate the TIGA environment. You will notice that, to simplify the source code listings, each example makes calls to the functions *init_tiga* and *term_tiga*.

The above source code listings of *init_tiga* and *term_tiga* are an example of how to properly initialize and terminate the TIGA environment from a TIGA

application. Your initialization and termination code may differ slightly from the one presented here but should provide similar functionality.

3.5 TIGA Functions

A TIGA function falls into one of two classes:

- Core functions
- Extended functions

3.5.1 Core Functions

TIGA's core functions are always available to an application following proper initialization of the TIGA environment as shown in the example in Section 3.4. The majority of core functions can be called by host-PC applications and any extended function. There are, however, a few core functions that can be called only by a host-PC application and may not be called by an extended function. These functions are referred to as *host-only* core functions and are identified as such in Chapter 4.

3.5.2 Extended Functions

Extended functions are not a part of TIGA's core function set and must be explicitly loaded by an application before the application can call them.

TIGA includes an extended graphics library that contains a comprehensive set of 2-D drawing functions. The functions in this library are examples of extended functions because they must be loaded into TIGA via the *install_primitives()* function before being available to the application.

3.6 Summary of Functions by Functional Group

3.6.1 Graphics System Initialization Functions

The graphics system initialization functions perform the initializing, terminating, and inquiring of the TIGA environment. Before a TIGA function is called, the TIGA environment must be properly initialized. Similarly, once the TIGA application has completed, the graphics environment, which existed prior to running the TIGA application, must be properly restored. These initialization and termination tasks are handled by the functions shown in Table 3–4.

Table 3–4. Graphics System Initialization Functions

Function	Description	Type
aux_command	Execute auxiliary command	Host
function_implemented	Return if function is implemented	Host
get_config	Return hardware configuration information	Core
get_modeinfo	Return graphics mode information	Host
get_videomode	Return current video mode	Host
gm_idlefunction	Enable/disable GM idle function	Core
gsp_execute	Execute a COFF program	Host
install_primitives	Install extended graphics library functions	Host
install_usererror	Install user error handler	Host
loadcoff	Load COFF file	Host
set_config	Set hardware configuration	Host
set_timeout	Set timeout delay value	Host
set_videomode	Set video mode	Host
setup_hostcmd	Initialize callback environment	Host
synchronize	Synchronize host and TMS340 communications	Host
tiga_busy	Determine if TIGA is busy	Host
tiga_set	Open/close/query communication driver	Host

3.6.2 Clear Functions

The clear functions, shown in Table 3–5, provide different ways to clear the screen. They all attempt to use any special memory functions (such as shift-register transfers), that the board or the memory chips themselves may have, to perform as quickly as possible.

Table 3–5. Clear Functions

Function	Description	Type
clear_frame_buffer	Clear frame buffer	Core
clear_page	Clear current drawing page	Core
clear_screen	Clear screen	Core

3.6.3 Graphics Attribute Control Functions

The graphics attribute control functions, shown in Table 3–6, are used to modify and query graphics attributes used by the TIGA drawing functions when drawing to the screen. See Chapter 6, *Graphics Library Conventions*, for additional information concerning graphics attributes.

Table 3–6. Graphics Attribute Control Functions

Function	Description	Type
cpw	Compare point to clipping window	Core
get_colors	Return foreground and background colors	Core
get_env	Return graphics environment information	Ext
get_pmask	Return plane mask	Core
get_ppop	Return pixel-processing operation code	Core
get_transp	Return transparency flag	Core
get_windowing	Return window-clipping mode	Core
set_bcolor	Set background color	Core
set_clip_rect	Set clipping rectangle	Core
set_colors	Set foreground and background colors	Core
set_draw_origin	Set drawing origin	Ext
set_fcolor	Set foreground color	Core
set_patn	Set current pattern address	Ext
set_pensize	Set pen size	Ext
set_pmask	Set plane mask	Core
set_ppop	Set pixel-processing operation code	Core
set_transp	Set transparency mode	Core
set_windowing	Set window-clipping mode	Core
transp_off	Turn transparency off	Core
transp_on	Turn transparency on	Core

3.6.4 Palette Functions

The palette functions, shown in Table 3–7, provide a board-independent way to modify and query palette values on the target TMS340-based board. See the example TIGA program in the directory *\tiga\demos\msc\tigademo* for more information on color and palette management with TIGA.

Table 3–7. Palette Functions

Function	Description	Type
get_nearest_color	Return nearest color in palette	Core
get_palet	Read entire palette	Core
get_palet_entry	Return single palette entry	Core
init_palet	Initialize palette	Core
set_palet	Set multiple palette entries	Core
set_palet_entry	Set single palette entry	Core

3.6.5 Graphics Drawing Functions

The graphics drawing functions, shown in Table 3–8, are self-explanatory. For further details concerning the drawing functions, see Chapter 6.

Table 3–8. Graphics Drawing Functions

Function	Description	Type
draw_line	Draw straight line	Ext
draw_oval	Draw ellipse outline	Ext
draw_ovalarc	Draw ellipse arc	Ext
draw_piearc	Draw ellipse pie arc	Ext
draw_point	Draw single pixel	Ext
draw_polyline	Draw list of lines	Ext
draw_rect	Draw rectangle outline	Ext
fill_convex	Draw solid convex polygon	Ext
fill_oval	Draw solid ellipse	Ext
fill_piearc	Draw solid ellipse pie slice	Ext
fill_polygon	Draw solid polygon	Ext
fill_rect	Draw solid rectangle	Ext

Table 3–8. Graphics Drawing Functions (Continued)

Function	Description	Type
frame_oval	Draw oval border	Ext
frame_rect	Draw rectangular border	Ext
patnfill_convex	Fill convex polygon with pattern	Ext
patnfill_oval	Fill oval with pattern	Ext
patnfill_piearc	Fill pie slice 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
patnpen_polyline	Draw polyline with pen and pattern	Ext
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
put_pixel	Assign value to pixel	Ext
seed_fill	Fill region with color	Ext
seed_patnfill	Fill region with pattern	Ext
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

3.6.6 Poly Drawing Functions

The TIGA communication driver functions pass the arguments of all the TIGA functions into a communication buffer for the TIGA graphics manager to use. Nearly all TIGA functions have fixed-size arguments that fit easily into the communication buffer. This is not the case with the poly drawing functions, shown in Table 3–9, which have a point list parameter that can be of any length. It is

easy for the function to overflow the buffer, destroying the TIGA environment. The application can either check the size of the data that it is sending, against the communication buffer size in the CONFIG structure, or it can use alternate entry points (with an *_a* appended to the function name), which use a buffer allocated from the dynamic heap pool, to store the data. However, these alternate entry points are slower.

Table 3–9. Poly Drawing Functions

Function	Description	Type
draw_polyline	Draw polyline	Ext
fill_convex	Draw solid convex polygon	Ext
fill_polygon	Fill polygon	Ext
patnfill_convex	Pattern fill convex polygon	Ext
patnfill_polygon	Pattern fill polygon	Ext
patnpen_polyline	Pattern pen polyline	Ext
pen_polyline	Pen polyline	Ext

3.6.7 Pixel Array Functions

The pixel array functions, shown in Table 3–10, operate on rectangular pixel arrays. TIGA contains two bitmaps —the source and destination bitmaps— used as implied operands for most of these functions.

The source bitmap is ignored by all functions except *bitblt*, *swap_bm*, and z*oom_rect*. The destination bitmap is used as an implied operand for all drawing functions. If it is set to anything other than the screen, all drawing functions (other than *bitblt*) abort. In the future, linear drawing capability may be added to each drawing function to enable drawing into a linear bitmap.

Table 3–10. Pixel Array Functions

Function	Description	Type
bitblt	Transfer bit-aligned block	Ext
decode_rect	Decode rectangular image	Ext
encode_rect	Encode rectangular image	Ext
set_dstbm	Set destination bitmap	Ext
set_srcbm	Set source bitmap	Ext
swap_bm	Swap source and destination bitmaps	Ext
zoom_rect	Zoom source rectangle	Ext

3.6.8 Text Functions

The text functions, shown in Table 3–11, provide the bitmap text-handling capabilities available in TIGA. Additional information concerning these capabilities can be found in Chapter 7, *Bit-Mapped Text.*

Table 3–11. Text Functions

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

3.6.9 Graphics Cursor Functions

The graphics cursor functions, shown in Table 3–12, provide graphics cursor support. Consult the example program in the *set_curs_shape* function description in Chapter 4, page 4-103, for additional information.

Table 3–12. Graphics Cursor Functions

Function	Description	Type
get_curs_state	Return current cursor state	Core
get_curs_xy	Return current cursor position	Core
set_curs_shape	Set current cursor shape	Core
set_curs_state	Make cursor visible/invisible	Core
set_curs_xy	Set current cursor position	Core
set_cursattr	Set current cursor attributes	Core

3.6.10 Graphics Utility Functions

The graphics utility functions, shown in Table 3–13, are a group of miscellaneous graphics functions, most of which require no explanation other than what is given with the individual functions.

Table 3–13. Graphics Utility Functions

Function	Description	Type
cvxyl	Convert x-y address to linear address	Core
get_pixel	Return pixel value	Ext
get_wksp	Return workspace information	Core
lmo	Return leftmost one bit number	Core
peek_breg	Read from B-file register	Core
poke_breg	Write to B-file register	Core
rmo	Return rightmost one bit number	Core
set_wksp	Set workspace information	Core

3.6.11 Handle-Based Memory Management Functions

The handle-based memory management functions, shown in Table 3–14, provide TIGA with a handle-based memory management system. Handle-based memory management is preferred over pointer-based memory management because of its ability to reduce the amount of memory fragmentation. Memory fragmentation occurs when numerous allocations and deletions are made during an application. The fragmentation results in a reduction in the number of large, free memory blocks available to the application.

Table 3–14. Handle-Based Memory Management Functions

Function	Description	Type
gsph_alloc	Allocate memory block	Core
gsph_calloc	Allocate and clear memory	Core
gsph_compact	Invoke memory compaction routine	Core
gsph_deref	Return pointer to memory block referenced by handle	Core
gsph_falloc	Allocate memory block with associated function	Core
gsph_fcalloc	Allocate and clear memory with associated function	Core
gsph_findhandle	Return handle to specified memory address	Core
gsph_findmem	Return type of memory	Core
gsph_free	Free block of memory	Core
gsph_init	Initialize all user memory and compact all segments	Core
gsph_maxheap	Return size of largest alloc without compaction	Core
gsph_memtype	Set characteristics of memory block	Core
gsph_realloc	Reallocate block of memory	Core
gsph_totalfree	Return size of largest block with compaction	Core

3.6.12 Pointer-Based Memory Management Functions

The pointer-based memory management functions, shown in Table 3–15, provide TIGA with a pointer-based memory management system. The functions *gsp_malloc*, *gsp_free*, *gsp_calloc*, and *gsp_realloc* should be familiar to most C programmers. They operate in a manner similar to that of the memory management functions provided in the Microsoft C runtime library.

Table 3–15. Pointer-Based Memory Management Functions

Function	Description	Type
get_offscreen_memory	Return offscreen memory blocks	Core
gsp_calloc	Clear and allocate TMS340 memory	Core
gsp_free	Free TMS340 memory from allocation	Core
gsp_malloc	Allocate TMS340 memory	Core
gsp_maxheap	Return largest free block	Core
gsp_minit	Reinitialize dynamic memory pool	Core
gsp_realloc	Reallocate TMS340 memory	Core

3.6.13 Data Input/Output Functions

The data input/output functions, shown in Table 3–16, transfer data between host and TMS340 memory spaces, on between the TMS340 and its coprocessor.

Table 3–16. Data Input/Output Functions

Function	Description	Type
cop2gsp	Copy from coprocessor memory to TMS340 memory	Core
field_extract	Extract field from TMS340 memory	Core
field_insert	Insert field into TMS340 memory	Core
gsp2cop	Copy from TMS340 memory to coprocessor memory	Core
gsp2gsp	Transfer data within TMS340 memory	Core
gsp2host	Move data from TMS340 memory to host memory	Host
gsp2hostxy	Copy rectangular memory area from TMS340 to host	Host
host2gsp	Move data from host memory to TMS340 memory	Host
host2gspxy	Copy rectangular memory area from host to TMS340	Host

3.6.14 Extensibility Functions

The extensibility functions, shown in Table 3–17, are described in detail in Chapter 8.

Table 3–17. Extensibility Functions

Function	Description	Type
create_alm	Create absolute load module	Host
create_esym	Create external symbol table file	Host
flush_esym	Flush external symbol table file	Host
flush_extended	Flush all user extensions	Host
flush_module	Remove module from TMS340 memory	Core
get_isr_priorities	Return interrupt service routine priorities	Core
install_alm	Install absolute load module	Host
install_primitives	Install extended drawing functions	Host
install_rlm	Install relocatable load module	Host
set_module_state	Set state of loaded module	Core
sym_flush	Flush relocatable load module symbols	Core

3.6.15 Interrupt Handler Functions

The interrupt handler functions, shown in Table 3–18, provide interrupt handler support. For more information of interrupt handling in TIGA, refer to Section 8.9, *Installing Interrupts,* page 8-36.

Table 3–18. Interrupt Handler Functions

Function	Description	Type
get_vector	Return address at TMS340 trap vector	Core
page_busy	Return status of page flipping	Core
page_flip	Flip display and drawing pages	Core
set_interrupt	Set interrupt handler	Core
set_vector	Set contents of TMS340 trap vector	Core
wait_scan	Wait for scan line	Core

Chapter 4

Core Functions

This chapter discusses the core functions alphabetically. Each discussion

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

The examples in this chapter use the functions *init_tiga* and *term_tiga* to initialize and terminate the TIGA environment. Although the *init_tiga* and *term_tiga* functions are not actually TIGA functions, they do make calls to various TIGA functions. The *init_tiga* function initializes the TIGA environment and is called before any other TIGA function. The *term_tiga* function terminates a TIGA application by restoring the previous video mode and closing the TIGA communication driver. Refer to Section 3.4, page 3-6, for a sample TIGA application that illustrates the *init_tiga* and *term_tiga* functions.

4.1 Core Functions Reference

The following is an alphabetical table of contents for functions reference.

Function .... Page
aux_command 4-4
clear_frame_buffer 4-7
clear_page 4-8
clear_screen 4-10
cop2gsp 4-11
cpw 4-12
create_alm 4-14
create_esym 4-15
cvxyl 4-16
field_extract 4-17
field_insert 4-18
flush_esym 4-19
flush_extended 4-20
flush_module 4-21
function_implemented 4-22
get_colors 4-23
get_config 4-24
get_curs_state 4-26
get_curs_xy 4-27
get_fontinfo 4-28
get_isr_priorities 4-30
get_modeinfo 4-31
get_nearest_color 4-34
get_offscreen_memory 4-36
get_palet 4-38
get_palet_entry 4-40
get_pmask 4-42
get_ppop 4-43
get_text_xy 4-45
get_transp 4-46
get_vector 4-47
get_videomode 4-48
get_windowing 4-49
get_wksp 4-50
gm_idlefunction 4-51
gsp2cop 4-52
gsp2gsp 4-53
gsp2host 4-54
gsp2hostxy 4-55
gsp_calloc 4-56
gsp_execute 4-57
gsp_free 4-58
gsp_malloc 4-59
gsp_maxheap 4-60
gsp_minit 4-61
gsp_realloc 4-62
gsph_alloc 4-63
gsph_calloc 4-64
gsph_compact 4-65
gsph_deref 4-66
gsph_falloc 4-67
gsph_fcalloc 4-68
gsph_findhandle 4-69
gsph_findmem 4-70
gsph_free 4-71
gsph_init 4-72
gsph_maxheap 4-73

gsph_memtype 4-74
gsph_realloc 4-75
gsph_totalfree 4-76
host2gsp 4-77
host2gspxy 4-78
init_palet 4-79
init_text 4-80
install_alm 4-81
install_primitives 4-82
install_rlm 4-83
install_usererror 4-85
lmo 4-87
loadcoff 4-88
page_busy 4-89
page_flip 4-91
peek_breg 4-93
poke_breg 4-94
rmo 4-95
set_bcolor 4-96
set_clip_rect 4-97
set_colors 4-99
set_config 4-100
set_curs_shape 4-103
set_curs_state 4-108
set_curs_xy 4-109
set_cursattr 4-110
set_fcolor 4-111
set_interrupt 4-112
set_module_state 4-113
set_palet 4-115
set_palet_entry 4-117
set_pmask 4-118
set_ppop 4-120
set_text_xy 4-122
set_timeout 4-123
set_transp 4-124
set_vector 4-125
set_videomode 4-126
set_windowing 4-128
set_wksp 4-129
setup_hostcmd 4-130
sym_flush 4-131
synchronize 4-132
text_out 4-133
text_outp 4-134
tiga_busy 4-135
tiga_set 4-136
transp_off 4-138
transp_on 4-139
wait_scan 4-140

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void far *aux_command(command_index, command_buffer)
    unsigned short command_index;
    void far *command_buffer;
```

Type

Host-only

Description

The *aux_command* function invokes hardware-specific extended functions provided by the board OEM. Currently, two auxiliary commands are defined and callable on every TIGA-compliant board. Information pertaining to any additional auxiliary commands is supplied by the respective TIGA board manufacturer.

The argument *command_index* specifies the auxiliary command to be executed. The argument *command_buffer* is a far pointer to arguments that may be required by the specified command. If no arguments are required, *command_buffer* is NULL. The format of the far pointer is dependent on the compiler being used, as follows:

Compiler	Pointer Format
Microsoft	DX = Segment, AX = Offset
MetaWare	DX = Selector, EAX = Offset
Microway	EAX = Selector :: 16 bit Offset

For additional examples of correct far pointer handling, refer to the TEST_AUX example program for the various compilers in the *\tiga\demos* directory.

Note:

When the host PC is in protected mode, this command may pass back a data readable code selector as the 16 MSBs of the pointer. Any attempt to write to this segment results in a memory protection fault.

Command indices 0 – 99 are reserved for the exclusive use of Texas Instruments. Command indices 100 – 65535 are available for OEM use.

The following command indices have been defined and are supported on all TIGA compliant boards:

- command_index = 0, command_buffer = NULL

  Auxiliary command 0 with a NULL *command_buffer* pointer returns a far pointer to a null terminated string identifying the TMS340-based board on which the TIGA driver is operating. A null pointer may be returned by this command.

- command_index = 1, command_buffer = NULL

Auxiliary command 1 with a NULL command_buffer pointer returns a far pointer to the following AUX_CONFIG structure:

```
typedef struct
{
  unsigned long   base;
  unsigned long   range;
}MEM_MAP;

typedef struct
{
  unsigned short  base;
  unsigned short  range;
}IO_MAP;

typedef struct
{
  unsigned short  alt_video;
  unsigned short  emulation;
  short           mem_mapped;
  MEM_MAP         memmap[8];
  short           io_mapped;
  IO_MAP          iomap[8];
}AUX_CONFIG;
```

The fields of the MEM_MAP structure are defined as follows:

base	A 32-bit field that specifies the paragraph address corresponding to the start of the memory-mapped host area. For example, an origin value of D700 specifies the actual address of D700:0 or D7000.
range	A 32-bit field that specifies the size in bytes, of the memory-mapped area, beginning at the paragraph address specified by the origin.

The fields of the IO_MAP structure are defined as follows:

base	A 16-bit field that specifies the starting I/O address of a range I/O address.
range	A 16-bit field that specifies the range, in bytes, of the I/O-mapped area beginning at the I/O address specified by base. For example, a base address of 0x280 with range of 0x10 specifies the I/O address of 0x280 to 0x290.

The fields of the AUX_CONFIG structure are defined as follows:

alt_video	Alternate video capabilities. This 16-bit field describes the alternate non-TMS340 video capabilities of the TMS340-based board.
Bits 0	MDA capability (0=no / 1=yes)
Bits 1	CGA capability (0=no / 1=yes)

Bits 2	HERC capability (0=no / 1=yes)
Bits 3	EGA capability (0=no / 1=yes)
Bits 4	VGA capability (0=no / 1=yes)
Bits 5	8514/A capability (0=no / 1=yes)
Bits 6–12	Reserved
Bit 13	Shared VRAM buffer (0=no / 1=yes)
Bits 14–15	Monitor config (00=Single / 01=Dual)

Capability is defined as the ability of the TMS340-based board to display output in the indicated format, either through onboard chip support, pass-through, or TMS340 software emulation.

emulation — Alternate video emulation flags. This 16-bit field tells whether the capabilities described in the *alt_video* field are emulated in software by the TMS340 processor.

Bits 0	MDA emulated (0=no / 1=yes)
Bits 1	CGA emulated (0=no / 1=yes)
Bits 2	HERC emulated (0=no / 1=yes)
Bits 3	EGA emulated (0=no / 1=yes)
Bits 4	VGA emulated (0=no / 1=yes)
Bits 5	8514/A emulated (0=no / 1=yes)
Bits 6–15	Reserved

Emulation is defined as the entire or partial support that the TMS340 provides through software emulation, processor of the indicated video format. Emulation also indicates that the alternate video mode and TIGA modes share the same video buffer and/or program RAM area.

mem_mapped — A16-bit field that specifies the number of memory-mapped host address areas used by the board.

memmap — An array structure of type MEM_MAP, containing the base address and *range* of each memory-mapped area. The number of areas is specified by the *mem_mapped* field.

io_mapped — A 16-bit field that specifies the number of I/O-mapped host address areas used by the board.

iomap — An array of structure of type IO_MAP, containing the base address and range of each I/O-mapped area. The number of areas is specified by the *io_mapped* field.

Syntax

```
#include <tiga.h>

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

Type

Core

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. (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.

Offscreen 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 offscreen 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.

```
#include <tiga.h>

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

Syntax

```
#include <tiga.h>

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

Type Core

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 memory. In other words, the pixel value is replicated 32/N times within each 32-bit long word in the area of display memory corresponding to 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.)

The *clear_page* 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 offscreen as well as onscreen areas of the display memory. Data stored in offscreen areas may be lost as a result. The *clear_screen* function is similar in operation but does not affect data contained in offscreen 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 current graphics mode 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*.

```
#include <tiga.h>

#define RADIUS 64          /* radius of rotating text        */

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

  init_tiga(0);
  drawpage = 0;
  disppage = 1;
  x = (long)RADIUS << 16;
  y = 0;
  do
  {
    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");
  }while(!kbhit());
  getch();
  term_tiga();
}
```

Syntax

#include <tiga.h>

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

Type

Core

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 memory. In other words, the pixel value is replicated 32/N times within each 32-bit long word 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 screen to the current background color. (To clear the screen to all 1s when the pixel size is 32 bits, set the background color to 0xFFFFFFFF and call the c*lear_screen* function with an argument of –1.)

The *clear_screen* function does not affect data contained in offscreen areas of the display memory. The *clear_page* function is similar in operation but may affect data contained in offscreen 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 before printing the text “Hello world” on the screen.

```
#include <tiga.h>

main()
{
   init_tiga(0);
   clear_screen(-1);
   text_out(10, 10, ”Hello world.”);
   term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void cop2gsp(copid, copaddr, gspaddr, length)
    short copid;
    PTR copaddr;
    PTR gspaddr;
    long length;
```

Type

Core

Description

The *cop2gsp* function copies data from the address space of the TMS34082 coprocessor with id *copid* (a number from 0 to 7, with 4 being broadcast, as defined in the TMS34020 specification) into TMS340 memory. The data to be transferred is in 32-bit long words.

Syntax

```
#include <tiga.h>

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

Type

Core

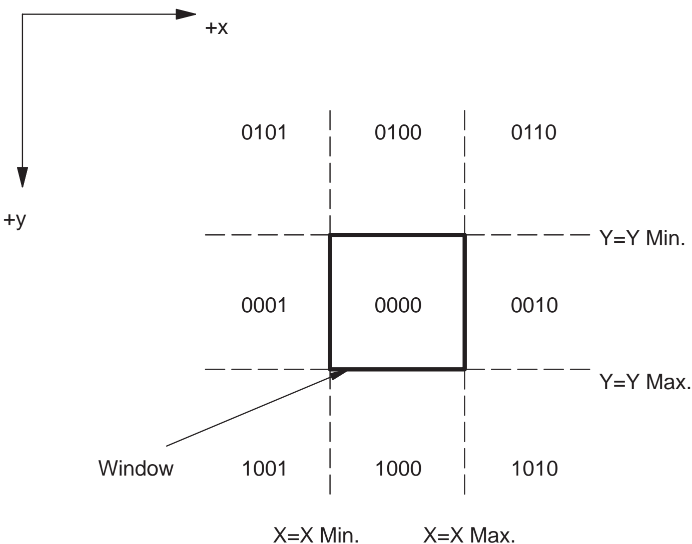
Description

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 clipping 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 4–1, the area surrounding the clipping window is partitioned into 8 regions.

Figure 4–1. Outcodes for Lines 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.

```
#include <tiga.h>
#define WCLIP 130          /* width of clipping window       */
#define HCLIP 100          /* height of clipping window      */

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

  init_tiga(0);
  clear_screen(-1);
  set_clip_rect(WCLIP, HCLIP, 0, 0);
  vx = 2;
  vy = 1;
  x = y = 0;
  do
  {
    text_out(x, y, "*");
    if (cpw(x, 0))
      vx=-vx;
    if (cpw(0, y))
      vy=-vy;
    wait_scan(HCLIP);
    text_out(x, y," ");
  }while(!kbhit());
  getch();
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short create_alm(rlm_name, alm_name)
    char *rlm_name;
    char *alm_name;
```

Type

Host-only

Description

The *create_alm* function creates an Absolute Load Module (ALM). ALMs were required before version 2.0 of TIGA because the downloading of a user extension to TIGA was done by calling the linking loader. This is not the case in versions 2.0 onward, so ALMs are now redundant. This function is included purely to maintain compatibility with TIGA drivers written before version 2.0 of TIGA.

The *create_alm* function converts the relocatable load module (specified by *rlm_name*) into an absolute load module and saves it under the filename specified by *alm_name*. If no file extension is supplied for the RLM, then an extension of .RLM is used. If no extension is supplied for the ALM, then an extension of .ALM is used. If no path information is specified, this function looks first in the current directory and then in the directory specified by the *–l* option of the TIGA environment variable.

If the ALM file already exists, the procedure does nothing. This saves time by creating the ALM file only once. If a new ALM file is desired, the old one must be deleted explicitly. For more details on extensibility and an example of the use of this function, refer to subsection 8.3.2.

If the function terminates correctly, zero is returned; if an error occurs, a negative error code is returned. The function returns these error codes:

Error Code	Description
–3	**Out of Memory** — Not enough TMS340 memory to load the ALM.
–6	**Error Accessing RLM** — Unable to open RLM for reading. Either the spelling of the RLM filename does not match the RLM filename in the current directory, or the *–l* option of the TIGA environment variable is not set up correctly.
–8	**Error Accessing ALM** — Unable to open ALM for writing; check the specified file name.
–10	**Symbol Reference —** An unresolved symbol was refernced by the RLM. Determine the name of the symbol, either by producing a link map for the RLM or by invoking TIGALNK from the command line via the *–ec* option.
–14	**Error Loading COFF File** — An error was obtained in the loading of the COFF file. Recreate the RLM and try again.
–15	**Out of Symbol Memory** — Not enough TMS340 memory to store the symbols of the RLM.

Syntax short create_esym()

Type Host-only

Description

The *create_esym* function does not need to be called. It is provided for downward compatibility with prior versions of TIGA, which provided stored symbol table information in a host file.

This function creates a new symbol table in TMS340 memory. After creation, the table contains only the global (or external) symbols that were contained in the TIGA graphics manager file. During subsequent installations of relocatable load modules, this table is used to resolve external references. Also, any external symbols contained in the RLM are added to table during the installation process so that those symbols can be referenced by other RLMs.

The symbol table is flushed automatically at the start of each application or you can do it explicitly by calling the *flush_esym* or the *flush_extended* functions. For more details on extensibility, refer to Chapter 8.

If an error occurs, a negative error code is returned. If the function terminates normally, zero is returned.

This function can return the following error codes:

Error Code	Description
–3	**Out of Memory** — Not enough TMS340 memory to create the symbol table.
–9	**Error Accessing TIGAGM.OUT** — Could not access the graphics manager COFF file to read in the symbols. The file is either missing, or the *–m* option of the TIGA environment variable is not set up correctly.
–14	**Error Loading COFF File** — An error was obtained in the load of the *tigagm.out* COFF file. Recopy this file from the installation disk.
–15	**Out of Symbol Memory** — Not enough TMS340 memory to store the symbol table.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

Description

The *cvxyl* function returns the 32-bit linear 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 offscreen 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 the current graphics mode. 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 <tiga.h>
#include <typedefs.h>

main()
{
  short x, y, n;
  char s[16];
  CONFIG cfg;
  FONTINFO fntinf;

  init_tiga(0);
  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);
    sprintf(s, "0x%lx", cvxyl(0,0));
    y+=fntinf.charhigh;
    page_flip(0, 0);
    text_out(x, y, s);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

unsigned long field_extract(gptr,fs)
   PTR gptr;              /* TMS340 memory pointer          */
   unsigned short fs;     /* field size                     */
```

Type

Core

Description

The *field_extract* function returns the contents of a field located in TMS340 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 of 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.

```
#include <tiga.h>
#define CONTROL 0xC00000B0 /*TMS340 CONTROL reg. address    */
#define XOR 10             /*PPOP = XOR                     */

main()
{
  unsigned short ppop;
  char s[16];

  init_tiga(0);
  clear_screen(-1);
  set_ppop(XOR);      /* load PPOP field                    */
  synchronize();      /* wait for set_ppop to complete      */
  ppop=field_extract(CONTROL+10, 5); /* read it back        */
  sprintf(s, "PPOP = %d", ppop);
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void field_insert(gptr, fs, val)
   PTR gptr;              /* TMS340 memory pointer     */
   short fs;              /* field size                */
   unsigned long val;     /* data to be inserted       */
```

Type

Core

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 of 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.

```
#include <tiga.h>
#define CONTROL 0xC00000B0      /* I/O register address      */
#define NOT_OR 13               /* NOT src OR dst --> dst    */

main()
{
   char s[16];

   init_tiga(0);
   clear_screen(-1);
   synchronize();   /* wait for clear_screen to complete      */
   field_insert(CONTROL+10, 5, NOT_OR);  /* load PPOP field  */
   sprintf(s, "PPOP = %d", get_ppop());
   text_out(10, 10, s);
   term_tiga();
}
```

Syntax

```
#include <tiga.h>

short flush_esym()
```

Type

Host-only

Description

The *flush_esym* function does not need to be called by the user. It is provided for backwards compatibility with older versions of TIGA, which provided stored symbol table information in a host file.

This function deletes all unsecured installed modules and flushes the module symbols from the symbol table, except for those in the TIGA graphics manager.

Currently, there are no error codes returned by this function.

For more details on extensibility, refer to Chapter 8.

Syntax

```
#include <tiga.h>

void flush_extended()
```

Type Host-only

Description

The *flush_extended* function deletes all unsecured installed modules and flushes the module symbols from the symbol table, except for those in the TIGA graphics manager.

Note that if TIGA is initialized with a call to *set_videomode* with an INIT style parameter, this function is executed automatically because all unsecured heap is initialized.

For more details on extensibility and the use of this function, see Chapter 8.

Syntax

```
#include <tiga.h>

short flush_module(module_id)
    short module_id;        /* module identifier                 */
```

Type

Host-only

Description

The *flush_module* function flushes the code and symbols of the module specified by the argument *module_id*, which is returned by the *install_rlm* and *install_alm* functions. Use the constant GRAPHICS_LIB_ID defined in the *tiga.h* file to specify the module corresponding to the extended graphics library module.

The function returns the following values:

1 = Module flushed succesfully
0 = Error, caused by one of the following:

- Invalid module id
- Specified module is not loaded
- Specified module is locked

Syntax

```
#include <tiga.h>

short function_implemented(function_code)
   short function_code;
```

Type Host-only

Description

The *function_implemented* function queries whether a function is implemented or not. Functions in TIGA have an associated *function_code*; some functions may not be implemented on every board.

The following functions are not likely to be implemented on all boards and should be queried with *function_implemented* before being invoked:

cop2gsp	get_palet_entry
set_palet	set_transp
get_palet	gsp2cop
set_palet_entry	init_palet

The function codes themselves are contained in the *tiga.h*, *tiga.hch*, and *tiga.ndp* include files, which contain the type and function declarations. The function codes are *#defined* to be the same as the function name, but in upper case. Thus, the syntax to inquire if *set_palet* is implemented is

```
if(function_implemented(SET_PALET))
{
 .....
}
```

Example

```
#include <tiga.h>
#include <typedefs.h>

CONFIG config;

main()
{
   unsigned long green_index;

   init_tiga(0);
   /* if it is possible to set the palet entry value, */
   /* set it to bright green         */
   if (function_implemented(SET_PALET_ENTRY))
   {
      green_index = 1;
      set_palet_entry(green_index, 0, 0xFF, 0, 0);
   }
   else
   {
      /* if it is not possible to set the palet entry, */
      /* (as in the case of a ROM–based palette)   */
      /* then get the index of the brightest green */
      green_index = get_nearest_color(0, 0xFF, 0, 0);
   }
   /* use index to clear the screen to   */
   clear_screen(green_index);
   term_tiga();
}
```

Syntax

#include <tiga.h>

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

Type

Core

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 is zero-extended.

Example

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

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

main()
{
  FONTINFO fontinfo;
  unsigned long fcolor, bcolor;
  short x, y;
  char s[20];

  init_tiga(0);
  clear_screen(-1);
  get_fontinfo(0, &fontinfo);
  get_colors(&fcolor, &bcolor);     /* retrieve colors       */
  x = y = 10;
  sprintf(s, "white = %ld", fcolor);
  text_out(x, y, s);
  y += fontinfo.charhigh;
  sprintf(s, "black = %ld", bcolor);
  text_out(x, y, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

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 that contains parameters describing the currently selected graphics mode.

The CONFIG structure contains the following fields:

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

The MODEINFO structure contains the following fields:

```
typedef struct
{
  unsigned long  disp_pitch;
  unsigned short disp_vres;
  unsigned short disp_hres;
  short          screen_wide;
  short          screen_high;
  unsigned short disp_psize;
  unsigned long  pixel_mask;
  unsigned short palet_gun_depth;
  unsigned long  palet_size;
  short          palet_inset;
  unsigned short num_pages;
  short          num_offscrn_areas;
  unsigned long  wksp_addr;
  unsigned long  wksp_pitch;
  unsigned short silicon_capability;
  unsigned short color_class;
  unsigned long  red_mask;
  unsigned long  blue_mask;
  unsigned long  green_mask;
  unsigned short x_aspect;
  unsigned short y_aspect;
  unsigned short diagonal_aspect;
}MODEINFO;
```

Refer to the CONFIG and MODEINFO structure descriptions in Appendix A for detailed descriptions of each field.

Note that the CONFIG and MODEINFO structures 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 <tiga.h>
#include <typedefs.h>      /* defines CONFIG structure       */

main()
{
  CONFIG cfg;
  char s[20];

  init_tiga(0);
  clear_screen(-1);
  get_config(&cfg);
  sprintf(s,"Pixel size = %d", cfg.mode.disp_psize);
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short get_curs_state()
```

Type Core

Description The *get_curs_state* function returns true (nonzero) if the graphics cursor is enabled, false (zero) otherwise.

Example See cursor manipulation in *set_curs_shape*.

Syntax

```
#include <tiga.h>

void get_curs_xy(px, py)
    short *px;
    short *py;
```

Type

Core

Description

The *get_curs_xy* returns the pixel coordinates of the graphics cursor hotspot. Note that the coordinates are relative to the upper left corner of the screen, not to the drawing origin.

> **Note:**
>
> Because the cursor is moved by an interrupt service routine driven by the display interrupt of the TMS340 processor, it is possible to set a new position and then execute a *get_curs_xy* to return this new position ***before*** the interrupt service routine has actually moved the cursor. Thus, the function returns the new position either where the cursor is, if the display interrupt has been invoked, or where it will be, once the display interrupt has been invoked for the current frame.

Example

See cursor manipulation in *set_curs_shape*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

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 TIGA. 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, it rerutns 0.

The FONTINFO structure contains the following fields:

```
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;
  PTR    fontptr;
  short  id;
}FONTINFO;
```

Refer to the FONTINFO structure description in appendix A for detailed descriptions of each field.

Note that the FONTINFO structure 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 display the three font parameters on the screen.

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

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

  init_tiga(0);
  clear_screen(–1);
  get_fontinfo(0, &fntinf);
  x=y=10;
  text_out(x, y, fntinf.facename);
  y += fntinf.charhigh;
  sprintf(s, ”character width = %d”, fntinf.charwide);
  text_out(x, y, s);
  y+=fntinf.charhigh;
  sprintf(s, ”character height = %d”, fntinf.charhigh);
  text_out(x, y, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

void get_isr_priorities(numisrs, ptr)
    short numisrs;          /* number of isrs                */
    short *ptr;             /* pointer to array of shorts    */
```

Type

Core

Description

The *get_isr_priorities* function returns the priorities assigned when installing interrupt service routines (ISRs) using the *install_rlm* or *install_alm* functions. The calling function must ensure that enough space is allocated to hold all returned priority information.

There is a one-to-one correspondence between an ISR and its associated priority. The first priority returned corresponds to the first ISR installed, and so on.

After this function is called, all priority data is flushed internally within TIGA, thereby enabling new priority data to be collected the next time *install_alm* or *install_rlm* is called to install an ISR.

For more details on extensibility and the use of this function, see Chapter 8.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Host-only

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 MODEINFO structure contains the following fields:

```
typedef struct
{
  unsigned long  disp_pitch;
  unsigned short disp_vres;
  unsigned short disp_hres;
  short          screen_wide;
  short          screen_high;
  unsigned short disp_psize;
  unsigned long  pixel_mask;
  unsigned short palet_gun_depth;
  unsigned long  palet_size;
  short          palet_inset;
  unsigned short num_pages;
  short          num_offscrn_areas;
  unsigned long  wksp_addr;
  unsigned long  wksp_pitch;
  unsigned short silicon_capability;
  unsigned short color_class;
  unsigned long  red_mask;
  unsigned long  blue_mask;
  unsigned long  green_mask;
  unsigned short x_aspect;
  unsigned short y_aspect;
  unsigned short diagonal_aspect;
}MODEINFO;
```

Refer to the MODEINFO structure description in Appendix A for detailed descriptions of each field.

Note that the MODEINFO structure 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 <tiga.h>
#include <typedefs.h>      /* MODEINFO, CONFIG & FONTINFO    */

main()
{
  MODEINFO modinf;
  FONTINFO fntinf;
  char s[80];
  short x, y, mode;

  init_tiga(0);
  clear_screen(-1);
  get_fontinfo(0, &fntinf);
  x = y = 10;
  for (mode = 0; get_modeinfo(mode, &modinf); mode++)
  {
    sprinf(s, "mode = %d: %d-by-%d", mode,
            modinf.disp_hres, modinf.disp_vres);
    text_out(x, y, s);
    y += fntinf.charhigh;
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

Description

The *get_nearest_color* function 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 of 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 pixel 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 functions library to render the wheel. The wheel is partitioned into the following six regions of color transition:

- red to yellow
- yellow to green
- green to cyan
- cyan to blue
- blue to magenta
- magenta to red

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

```
#include <tiga.h>
#include <extend.h>

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;

  init_tiga(1);
  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        */
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

Description

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

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

The OFFSCREEN_AREA structure contains the following fields:

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

Refer to the OFFSCREEN_AREA structure description in Appendix A for detailed descriptions of each field.

An offscreen 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 offscreen 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 offscreen buffer is used as the offscreen 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 offscreen 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 offscreen 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 offscreen buffers available in the current graphics mode. Use the *text_out* function to print on the screen the x and y extent of each buffer .

```
#include <tiga.h>
#include <typedefs.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, nbufs;
  char s[80];

  init_tiga(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++)
    {
      sprintf(s, ”Buffer %d: xext = %d, yext = %d”, i,
            offscrn[i].xext, offscrn[i].yext);
      text_out(x, y, c);
      y += fntinf.charhigh;
    }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

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. The PALET structure contains the following fields:

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

Refer to the PALET structure description in Appendix A for detailed descriptions of each field.

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 of 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 palette 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 <tiga.h>
#include <typedefs.h>      /* PALET, CONFIG and FONTINFO      */
#define MAXSIZE 8          /* max. LUT entries to print       */

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

  init_tiga(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)
  {
    sprintf(s, "Color %d: r = %d, g = %d, b = %d, i = %d",
             k, lut[k].r, lut[k].g, lut[k].b, lut[k].i);
    text_out(x, y, s);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

Description

The *get_palet_entry* function 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 lookup 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 of 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 <tiga.h>
#include <typedefs.h>  /* CONFIG and FONTINFO struct's       */
#define MAXSIZE 8      /* max. LUT entries to print          */

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

  init_tiga(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);
    sprintf(s, "Color %d: r = %d, g = %d, b = %d",
            i = %d", k, r, g, b, i);
    text_out(x, y, s);
  }
  term_tiga();
}
```

Syntax

#include <tiga.h>

unsigned long get_pmask()

Type

Core

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 is 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. 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. Use the *text_out* function to print the default plane mask value to the screen.

```
#include <tiga.h>

main()
{
  unsigned long pmask;
  char s[16];

  init_tiga(0);
  clear_screen(-1);
  sprintf(s, "Plane mask = 0x%lx, get_pmask());
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

#include <tiga.h>

unsigned short get_ppop()

Type

Core

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 is zero-extended.

Legal PPOP codes are in the range of 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 4–1, Boolean operations are in the range of 0 to 15, and arithmetic operations are in the range of 16 to 21.

Table 4–1. 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. 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). Use the *text_out* function to print the default PPOP code to the screen.

```
#include <tiga.h>

main()
{
  char s[16];

  init_tiga(0);
  clear_screen(-1);
  sprintf(s, "PPOP code = %d", get_ppop());
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

#include <tiga.h>

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

Type

Core

Description

The *get_text_xy* function retrieves the x-y coordinates of 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 g*et_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 <tiga.h>
#include <typedefs.h>

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

   init_tiga(0);
   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;
   }
   term_tiga();
}
```

Syntax

#include <tiga.h>

short get_transp()

Type

Core

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 on the TMS34010, and is modified according to the transparency mode selected on the TMS34020. 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 default. Use the *text_out* function to print the value returned by the *get_transp* function to the screen.

```
#include <tiga.h>

main()
{
  char s[20];

  init_tiga(0);
  sprintf(s, "Transparency = %s", get_transp() ? "ON" :
            "OFF");
  clear_screen(-1);
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

#include <tiga.h>
#include <typedefs.h>

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

Type

Core

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 from –32768 to 32767 for a TMS34020, and from 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.

```
#include <tiga.h>

main()
{
  char s[32];

  init_tiga(0);
  clear_screen(–1);
  sprintf(s, "trap 0 vector = 0x%8lx", get_vector(0));
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short get_videomode();
```

Type Host-only

Description

The *get_videomode* function returns the current video mode. Possible video modes are discussed in the *set_videomode* function.

This function is normally used to obtain the video mode before entering TIGA mode. Once the TIGA application has completed, the previous video mode can be easily restored.

Syntax

```
#include <tiga.h>

short get_windowing()
```

Type

Core

Description

The *get_windowing* function returns a 2-bit code designating the current window-checking mode.

There are four windowing modes:

1) 00_2 Window clipping disabled

2) 01_2 Interrupt request on write to pixel inside window

3) 10_2 Interrupt request on write to pixel outside window

4) 11_2 Clip to window

TIGA's graphics library 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 if calls are made to the extended graphics library functions.

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, the system is configured in windowing mode 3, which is the default.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

Description

The *get_wksp* function retrieves the parameters that define the current offscreen workspace. None of the current TIGA core or extended functions 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 offscreen workspace.

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

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

The offscreen workspace is a 1-bit-per-pixel bitmap of the same width and height as the screen. If the display hardware provides sufficient offscreen memory, the workspace can be allocated statically. 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

```
#include <tiga.h>

short gm_idlefunction(mn,fn)
    short mn;               /* Module number of desired idle function     */
    short fn;               /* function number of desired idle function   */
```

Type

Core

Description

The *gm_idlefunction* function enables the function specified by the module number argument *mn* and by the function number argument *fn* to be called whenever the TIGA graphics manager (GM) is idle, that is, is not servicing a TIGA function call.

The argument *mn* corresponds to the module identifier returned by the *install_rlm* function. The argument *fn* corresponds to the index of the desired function within the specified module.

To ensure correct operation, the function specified to *gm_idlefunction* must conform to the following requirements to ensure correct operation:

- Be currently loaded into TMS340 memory (by calling the *install_rlm* function).
- The idle function, when called by the TIGA GM, is passed one 32-bit argument on the program stack (STK). This argument informs the idle function whether or not a TIGA function is next to be executed. Valid arguments are:

  0 = No TIGA command is pending.

  1 = A TIGA command is pending (this is the last time the idle function is called before servicing the TIGA function).

  The idle function must conform to the standard TMS340 C-calling conventions.
- Save and restore all registers that it uses (except for A8).
- Not modify the sign or sign extent of field 1.

To disable the GM idle function calling feature, specify a module number of –1.

If no function corresponding to the specified module and function numbers is currently loaded in TMS340 memory, 0 is returned. Otherwise, 1 is returned.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void gsp2cop(copid,gspaddr,copaddr,length)
    short copid;
    PTR gspaddr;
    PTR copaddr;
    long length;
```

Type

Core

Description

The *gsp2cop* function copies data from TMS340 space into the TMS34082 coprocessor space with ID *copid* (a number from 0 to 7, with 4 being broadcast, as defined in the *TMS34020 User's Guide*). The size of the data to be transferred is in 32-bit long words.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

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

Type

Core

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 adjusts 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.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void gsp2host(gptr, hptr, length, swizzle)
    PTR gptr;                   /* TMS340 memory pointer      */
    char far *hptr;             /* host memory pointer        */
    unsigned short length;      /* length in bytes            */
    short swizzle;              /* data SWIZZLE flag          */
```

Type

Host-only

Description

The *gsp2host* function copies *length* number of bytes from TMS340 memory pointed to by *gptr* to host memory at *hptr*. If *swizzle* is nonzero, the data is swizzled before it is written to the host (that is, the order of the bits in each byte is reversed). *gptr* is a pointer to TMS340 memory. It must be byte-aligned (that is, 3 LSBs must be 0). *hptr* is a pointer to host memory. It must be declared as a long pointer (for example, *segment:offset* format).

All data being passed to the host must fit in the segment specified by *hptr*. No segment boundary checking is performed by this function.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void gsp2hostxy(saddr, sptch, daddr, dptch, sx, sy, dx, dy,
                xext, yext, psize, swizzle)
  PTR saddr;
  long sptch;
  char far *daddr;
  long dptch;
  short sx;
  short sy;
  short dx;
  short dy;
  short xext;
  short yext;
  short psize;
  short swizzle;
```

Type

Host-only

Description

The *gsp2hostxy* function transfers a rectangular area from TMS340 to host memory. The area is extracted from the source bitmap, starting at address *saddr* in TMS340 memory and is *xext* by *yext* pixels, with the pixel size being *psize*. The area starts at coordinates (*sx, sy*) in the source bitmap and is transferred to coordinates (*dx, dy*) of the destination bitmap. Because the host memory address is restricted to be byte-address aligned, the rectangular area sent is always padded on every side (if necessary) to ensure that the data sent is aligned to the nearest byte boundary. The source pitch, *sptch*, is the difference in the linear addresses between two pixels in the same column and adjacent rows of the bitmap in TMS340 memory. *dptch* is the same for host memory.

If *swizzle* is nonzero, the data is swizzled before it is written to the host (that is, the order of the bits in each byte is reversed).

This function has three restrictions placed upon it:

- The source pitch (on the TMS340 side), *sptch*, though a long variable, must be a multiple of 16 and less than 16 bits long. The source address, *saddr*, must also be a multiple of 16.
- All data in the host array must be accessible from the segment address of daddr; that is, none of the data being transferred must have a host address that crosses segment boundaries.
- If data is being swizzled, it is transferred from TMS340 to host and then transferred back again. The integrity of the data is preserved only if it is transferred back to the same address it came from. Otherwise, the data may be garbled.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

PTR gsp_calloc(nmemb, size)
   long nmemb;        /* number of items to allocate           */
   long size;         /* size (in bytes) of each item          */
```

Type

Core

Description

The *gsp_calloc* function allocates a packet of TMS340 memory large enough to contain *nmemb* objects of the specified *size* and returns a pointer. If it cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function also initiates the allocated memory to all zeros. This function is used in conjunction with *gsp_free*, *gsp_malloc*, g*sp_minit*, and *gsp_realloc*.

Syntax

#include <tiga.h>

void gsp_execute(entry_point)
  unsigned long entry_point;

Type

Host-only

Description

The *gsp_execute* function is not of general use to a TIGA application but is included here because the capability to load the graphics manager is an integral part of TIGA. As a side effect of this, TIGA provides a portable COFF loader across all TMS340 boards. This function executes a program that has been loaded by the loadcoff function. The parameter *entry_point* specifies the starting TMS340 address of the program. On most TMS340 boards, this address loads into the NMI vector, and an NMI is performed.

Example

```
#include <tiga.h>

main(argc, argv)
int argc;
char *argv[];
{
  unsigned long entry;

  if (argc == 2)
  {
    if (tiga_set(CD_OPEN) >= 0)
    {
      if (entry = loadcoff(argv[1]))
        gsp_execute(entry);
      else
        printf("Error in load\n");
      tiga_set(CD_CLOSE);
    }
    else printf("TIGACD not running\n");
  }
  else printf("Usage: %s <coff filename>\n", argv[0]);
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

short gsp_free(ptr)
   PTR ptr;
```

Type

Core

Description

The *gsp_free* function deallocates a packet of TMS340 memory (pointed to by *ptr*) previously allocated by *gsp_malloc*, *gsp_calloc*, or *gsp_realloc*. The function takes no action and returns false (zero) when an attempt is made to free a packet not previously allocated. This function returns true (nonzero) if the function sucessfully frees a valid TMS340 pointer.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

PTR gsp_malloc(size)
    long size;              /* size (in bytes) of block            */
```

Type

Core

Description

The *gsp_malloc* function allocates a packet of TMS340 memory of a specified *size* and returns a pointer. If *gsp_malloc* is unable to allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates. This function is used in conjunction with *gsp_free*, *gsp_minit*, and *gsp_realloc*.

gsp_maxheap

Syntax

```
#include <tiga.h>

unsigned long gsp_maxheap()
```

Type

Core

Description

The *gsp_maxheap* function returns the size of the largest contiguous block of program memory for heap allocation. It can be used at the start of an application to determine the total size of the available memory for heap allocation. If called during an application, it informs the application of the largest available block to an object; for example, a bitmap can be downloaded.

Syntax

```
#include <tiga.h>

void gsp_minit(stack_size)
   long stack_size;
```

Type

Core

Description

The *gsp_minit* function reinitializes and frees all unsecured memory blocks in the TMS340 dynamic memory heap pool. Any previously allocated blocks are no longer allocated, and all pointers to such blocks become invalid after this procedure is called.

In previous versions of TIGA, this function could modify the size of the system stack by using argument *stack_size*. TIGA 2.0 no longer supports this feature; however, to maintain downward compatibility, you should still specify an argument of type *long* for this function.

Be careful when you use this function: all unsecured allocated blocks of memory are freed, including the background save area for the cursor (if stored in heap). Disable the cursor before calling this function and install a new cursor via a call to *set_curs_shape* afterward. If the workspace set by the *set_wksp* function was previously allocated in heap, it will have to be reset before using it. The *gsp_minit* function also frees data associated with downloaded extensions and interrupt service routines. Therefore, any required extensions or interrupt handlers must be reloaded after calling *gsp_minit*. This function is used in conjunction with *gsp_free*, *gsp_malloc*, and *gsp_realloc*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

PTR gsp_realloc(ptr, size)
   PTR ptr;                /* pointer to object to change      */
   unsigned long size;     /* new size (in bytes) of packet    */
```

Type

Core

Description

The *gsp_realloc* function changes the size of the allocated data area pointed to by the first argument, *ptr*, to the *size* specified by the second argument. It returns a pointer to the space allocated because the packet and its contents may have to be moved to expand. Any memory freed by this operation is deallocated. If an error occurs, the function returns a zero. This function is used in conjunction with *gsp_calloc*, *gsp_free*, *gsp_malloc*, and *gsp_minit*.

Syntax

```
#include <tiga.h>

unsigned short gsph_alloc(size)
    unsigned long size;
```

Type

Core

Description

The *gsph_alloc* function allocates a memory block of the size (in bytes) specified by the argument *size* and returns a handle identifying this memory block. The handle is used in subsequent handle-based memory functions to specify the memory block. The memory block's type defaults to moveable and undeletable. The function *gsp_memtype* may be used to modify the memory block's characteristics. If insufficient memory is available, NULL (0) is returned.

See also *gsph_memtype*.

Syntax

```
#include <tiga.h>

unsigned short gsph_calloc(ecount, esize)
    unsigned long ecount;
    unsigned long esize;
```

Type Core

Description

The *gsph_calloc* function allocates an area of memory whose size is specified by the arguments *ecount* and *esize*. Argument *ecount* specifies the number of blocks to be allocated, and argument *esize* specifies the size, in bytes, of each block. The allocated memory is then cleared (set to zero).

A handle to the allocated memory area is returned. If an error occurs, NULL (0) is returned.

Syntax

```
#include <tiga.h>

void gsph_compact(purge)
 short purge;
```

Type

Core

Description

The *gsph_compact* function invokes TIGA's memory manager compaction routine. This routine attempts to reorganize allocated memory blocks to create larger, contiguous memory areas for further allocation. The argument *purge* specifies if memory blocks marked as deletable are to be purged during compaction. If *purge* is set to true (1), then deletable blocks are purged.

See also *gsph_memtype*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

PTR gsph_deref(handle)
    unsigned short handle;
```

Type

Core

Description

The *gsph_deref* function returns a void pointer to the memory block (in TMS340 processor memory) referenced by the argument *handle*. This is useful if an application requires direct access to the memory block address. The argument *handle* must be a valid handle returned by either the *gsph_alloc* or *gsph_falloc* functions. If the handle passed is invalid, NULL (0) is returned by this function.

See also *gsph_alloc* and *gsph_falloc*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

unsigned short gsph_falloc(size, func, flags)
    unsigned long size;
    PTR func;
    unsigned char flags;
```

Type

Core

Description

The *gsph_falloc* function allocates a memory block of the size (in bytes) specified by argument *size* and returns a handle identifying this memory block. The handle is used in subsequent handle-based memory functions to specify the memory block.

The argument *func* is a pointer to a TMS340-resident function of type void, which is called if this memory block is ever moved or deleted by TIGA's memory manager. Note that the memory block's flags must be set accordingly to enable this function to be called.

The argument *flags* defines the memory block's initial characteristics. *flags* is composed of the following manifest constants:

BLK_FUNCDELETE	Call specified function when block is deleted
BLK_FUNCMOVE	Call specified function when block is moved

The memory block's type defaults to moveable and undeletable. However, if the BLK_FUNCDELETE flag is specified, then the block is marked as deletable. The function *gsph_memtype* may also be used to modify the memory block's characteristics. If insufficient memory is available, NULL is returned.

See also *gsph_memtype*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

unsigned short gsph_fcalloc(ecount, esize, func, flags)
    unsigned long ecount;
    unsigned long esize;
    PTR func;
    unsigned char flags;
```

Type

Core

Description

The *gsph_fcalloc* function allocates an area of memory whose size is specified by arguments *ecount* and *esize*. Argument *ecount* specifies the number of blocks to be allocated, and argument *esize* specifies the size, in bytes, of each block. The allocated memory is then cleared (set to zero).

A handle to the allocated memory area is returned. If an error occurs, NULL (0) is returned.

The argument *func* is a pointer to a TMS340-resident function of type *void*, which is called if this memory block is ever moved or deleted by TIGA's memory manager. Note that the memory block's flags must be set accordingly to allow this function to be called.

The argument *flags* defines the memory block's initial characteristics. It is composed of the following manifest constants:

BLK_FUNCDELETE	Call specified function when block is deleted
BLK_FUNCMOVE	Call specified function when block is moved

The memory block's type defaults to moveable and undeletable. However, if the BLK_FUNCDELETE flag is specified, then the block is marked as deletable. The function *gsph_memtype* may also be used to modify the memory block's characteristics.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

unsigned short gsph_findhandle(memptr)
   PTR memptr;
```

Type

Core

Description

The *gsph_findhandle* function returns the handle of the memory block specified by the TMS340 memory address in argument *memptr*. If the argument does not point to a valid memory block, the function returns NULL (0).This function performs the inverse of the *gsph_deref* function.

See also *gsph_deref*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

short gsph_findmem(memptr, handle, flags)
    PTR memptr;
    unsigned short *handle;
    unsigned char *flags;
```

Type

Core

Description

The *gsph_findmem* function returns information about the memory block containing the TMS340 address specified by the argument *memptr*.

This function returns, in argument *handle*, the handle of the memory block specified by the memory address in argument *memptr*; whereas the *gsph_deref* function returns a pointer to the memory block pointed to by the handle. The value of *handle* is NULL (0) if the memory address is not part of an allocated block.

Argument *flags* contains information about the memory block's status. Valid flags include:

BLK_INUSE	This memory block is currently in use.
BLK_DELETABLE	This memory block may be deletable.
BLK_LOCKED	This memory block must remain at its current address. It may not be moved during compaction by TIGA's memory manager.
BLK_SECURED	Secured system memory block (will not be purged by a *gsp_minit()* or *gsph_init()* function call).
BLK_FUNCMOVE	When the block is moved by TIGA's memory manager, a call is made to the previously installed memory function.
BLK_FUNCDELETE	When the block is purged by TIGA's memory manager, a call is made to the previously installed memory function.

The function returns

0	if the memory address supplied marks the start of a memory block managed by the memory manager.
1	if the memory address is contained in a memory block managed by the memory manager.
–1	if the memory lies outside that managed by the memory manager.

See also *gsph_deref* and *gsph_memtype*.

Syntax

```
#include <tiga.h>

void gsph_free(handle)
    unsigned short handle;
```

Type

Core

Description

The *gsph_free* function frees the memory associated with the memory block specified by the argument *handle*. This memory block is returned to the memory pool and is then available for further allocation. The argument *handle* must be a valid handle returned by either the *gsph_alloc* or *gsph_falloc* functions. No value is returned by this function.

See also *gsph_alloc* and *gsph_falloc*.

Syntax

```
#include <tiga.h>

void gsph_init(void)
```

Type

Core

Description

The *gsph_init* function frees all allocated nonsecured memory blocks and performs a compaction on this freed memory.

Syntax

#include <tiga.h>

unsigned long gsph_maxheap()

Type

Core

Description

The *gsph_maxheap* function returns the size, in bytes, of the largest memory block that can be allocated with the *gsph_alloc* or *gsph_falloc* functions. Note that the size returned by this function assumes that the memory block will be allocated from the current memory state. In other words, it is functionally equivalent to the *gsp_maxheap* function. This function is similar to the *gsph_totalfree* function, with the exception of the memory state assumption.

See also *gsph_alloc*, *gsph_falloc*, and *gsph_totalfree*.

Syntax

```
#include <tiga.h>

void gsph_memtype(handle,flags)
    unsigned short handle;
    unsigned char flags;
```

Type

Core

Description

The *gsph_memtype* function allows the characteristics of the memory block identified by the argument *handle* to be modified. The argument *handle* must be a valid handle returned by either the *gsph_alloc* or *gsph_falloc* functions. The argument *flags* specifies the new characteristics of the specified memory block and is composed of the following manifest constants, which are defined in *tiga.h*:

BLK_DELETABLE	This memory block may be deleted.
BLK_LOCKED	This memory block must remain at its current address. It may not be moved during compaction by TIGA's memory manager.
BLK_SECURED	Secured system memory block (will never be moved or purged by compaction or by a call to *gsph_init* or *gsp_minit*).
BLK_FUNCMOVE	When the block is moved by TIGA's memory manager, a call will be made to the previously installed memory function.
BLK_FUNCDELETE	When the block is purged by TIGA's memory manager, a call will be made to the previously installed memory function.

See also *gsph_alloc*, *gsph_falloc*, and *gsph_maxheap*.

Syntax

```
#include <tiga.h>

unsigned long gsph_realloc(handle, size)
    unsigned short handle;
    unsigned long size;
```

Type

Core

Description

The *gsph_realloc* function allows the size of a previously allocated memory block to be modified. The argument *handle* specifies the memory block whose size is to be adjusted. The argument *handle* must be a valid handle returned by either the *gsph_alloc* or *gsph_falloc* functions. The argument *size* specifies the new size of the memory block in bytes. The new size of the memory block (in bytes) is returned. If the requested size could not be allocated, then the amount that was actually allocated is returned.

See also *gsph_alloc* and *gsph_falloc*.

Syntax

```
#include <tiga.h>

unsigned long gsph_totalfree()
```

Type

Core

Description

The *gsph_totalfree* function returns the size, in bytes, of the largest contiguous memory block potentially available after compaction. Note that you will have to call the function *gsph_compact* before allocating a memory block of the size returned by the *gsph_totalfree* function.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void host2gsp (hptr, gptr, length, swizzle)
    char far *hptr;          /* host memory pointer      */
    PTR gptr;                /* TMS340 memory pointer    */
    unsigned short length;   /* length in bytes          */
    short swizzle;           /* data SWIZZLE flag        */
```

Type

Host-only

Description

The *host2gsp* function copies *length* number of bytes from the host memory pointed to by *hptr*, to TMS340 memory at *gptr*. If *swizzle* is nonzero, the data is swizzled before it is written to the TMS340 (that is, the order of the bits in each byte is reversed). Argument *hptr* is a pointer to host memory and must be declared as a long pointer (that is, segment:offset format). Argument *gptr* is a pointer to TMS340 memory. It must be byte-aligned (that is, 3 LSBs must be 0).

All data being passed from the host must fit in the segment specified by *hptr*. No segment-boundary checking is performed by this function.

Example

See *get_offscreen_memory*.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void host2gspxy(saddr, sptch, daddr, dptch, sx, sy, dx, dy,
                xext, yext, psize, swizzle)
    char far *saddr;
    long sptch;
    PTR daddr;
    long dptch;
    short sx;
    short sy;
    short dx;
    short dy;
    short xext;
    short yext;
    short psize;
    short swizzle;
```

Type

Host-only

Description

The *host2gspxy* function transfers a rectangular area from host to TMS340 memory. The area is extracted from the source bitmap, starting at address *saddr* in host memory, and is *xext* by *yext* pixels, with the pixel size being *psize*. The area starts at coordinates (*sx, sy*) in the source bitmap and is transferred to coordinates (*dx, dy*) of the destination bitmap. Because the host memory address is restricted to be byte-address aligned, the rectangular area sent is always padded on every side (if necessary) to ensure that the data sent is aligned to the nearest byte boundary.The source pitch, *sptch*, is the difference in the linear addresses between two pixels in the same column and adjacent rows of the bitmap in host memory. The destination pitch *dptch* is the same for TMS340 memory. Note that *dptch* and *daddr* must be multiples of 16.

If *swizzle* is nonzero, the data is swizzled before it is written to the TMS340 (that is, the order of the bits in each byte is reversed).

Syntax

#include <tiga.h>

void init_palet()

Type

Core

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 per pixel, palette indices 0–3 are assigned the default colors black, green, red, and white. At 1 bit per pixel, palette indices 0 and 1 are assigned the default colors black and white. If the palette is nonprogrammable, the function has no effect.

The palette is also initialized to the default colors any time the drawing environment is initialized. Initialization occurs when the *set_videomode* function is called with the *style* argument specified as INIT or INIT_GLOBALS, or when *set_config* is called with the *init_draw* argument set to *true*.

Example

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

```
#include <tiga.h>

main()
{
  short i;

  init_tiga(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        */
  term_tiga();
}
```

Syntax

#include <tiga.h>

void init_text()

Type

Core

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_videomode* and *set_config* functions also initialize the font table and text attributes as part of their 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 TIGA graphics library are used to install and select a proportionally spaced font.

Note that the function *loadinst_font* is called to load and install a TIGA font from a font file. The *loadinst_font* is not a TIGA function but does make calls to various TIGA functions to load a font. Refer to the *install_font* function description in Chapter 5 for a complete source listing of the *loadinst_font* function.

```
#include <tiga.h>
#include <typedefs.h> /* defines FONT & FONTINFO struct.    */
#include <extend.h>

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

  init_tiga(1);
  clear_screen(–1);
  x = y = 10;
  index = loadinst_font(”ti_rom16.fnt”);
  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   */
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short install_alm(alm_name)
    char far *alm_name;              /* load module filename             */
```

Type

Host-only

Description

The *install_alm* function installs an absolute load module (ALM). ALMs were required before version 2.0 of TIGA because the downloading of a user extension to TIGA was done by calling the linking loader. This is not the case in versions 2.0 and higher; so ALMs are now redundant. This function is included purely to maintain compatibility with TIGA drivers written before version 2.0 of TIGA.The *install_alm* function installs the absolute load module (specified by argument *alm_name*) into theTIGA graphics manager and returns a module identifier that is used to invoke the extensions specified in the TIGAEXT section.

If the module contains interrupt service routines, they are installed into TIGA, and the priority information associated with each can be retrieved, once installation is complete, with a call to *get_isr_priorities*, which returns a priority list for last block of ISRs installed. For more details on extensibility and the use of this function, refer to Chapter 8.

If an error occurs, a negative error code number is returned. Otherwise, a module identifier is returned. This module identifier should be used whenever a routine contained within this module is specified, by joining the identifier with the function number and command type using the bitwise-OR operator (|). The function returns these error codes:

Error Code	Description
–3	**Memory Error** – Either there is not enough TMS340 memory to load the ALM, or the address at which the ALM was linked to when it was created does not match the address at install time. Check this by trying to install the ALM by invoking TIGALNK from the command line with the *la* option.
–8	**Error Accessing ALM** – Unable to open ALM for reading, or the contents of the file were in unexpected format. The spelling of the ALM filename does not match the ALM filename in the current directory, or the *–l* option of the TIGA environment variable is not set up correctly, or alternatively, the file has been corrupted, in which case it will need to be recreated.

Example

See subsection 8.3.2 on page 8-7.

Syntax

#include <tiga.h>

short install_primitives()

Type

Host-only

Description

The *install_primitives* function is similar to a call to *install_rlm* and is used to download the TIGA extended graphics library functions such as *draw_line* etc. These functions must be loaded before an application can call them.

If the extended graphics library functions are currently installed when *install_primitives* is called, no action is performed.

The function returns the module id of the installed graphics library functions if successful. If an error occurred, a negative error code is returned as follows:

Error Code	Description
–3	**Out of Memory** — Not enough TMS340 memory to load the extended graphics library functions.
–6	**Error Accessing RLM** — Unable to open extended functions (primitives) RLM for reading. Either the file *extprims.rlm* is missing from the main TIGA directory, or the *–m* option of the TIGA environment variable is not set up correctly.
–10	**Symbol Reference** — An unresolved symbol was referenced by the extended graphics library. To determine the name of the symbol, invoke TIGALNK from the command line by using the *–ec* option.
–14	**Error Loading COFF File** — An error was obtained in the loading of the *extprims.rlm* COFF file. Recopy the *extprims.rlm* file from the installation disk.
–15	**Out of Symbol Memory** — Not enough TMS340 memory to store the symbols of the extended graphics library.

Example

```
#include <tiga.h>
#include <typedefs.h>
#include <extend.h>

main()
{
  CONFIG cfg;

  init_tiga(1);       /* Init TIGA and load graphics lib.   */
  get_config(&cfg);
  draw_line(0, 0, cfg.mode.disp_hres, cfg.mode.disp_vres);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short install_rlm(rlm_name)
   char *rlm_name;          /* load module filename        */
```

Type

Host-only

Description

The *install_rlm* function installs the relocatable load module (specified by the argument *rlm_name*) into TIGA and returns a module identifier that is used to invoke the extensions specified in the TIGAEXT section.

If the module contains interrupt service routines, they are installed into the TIGA graphics manager. The priority information associated with each can be retrieved, once installation is complete, with a call to the function *get_isr_priorities*, which returns a priority list for the last block of ISRs installed.

If an error occurs, a negative value is returned. Otherwise, a module identifier is returned. Whenever a routine contained within this module is specified, join the module identifier with the routine number and command type by using the bitwise-OR operator (|).

The *rlm_name* argument can contain the special flag *%f* appended to the end of the name (for example, *install_rlm("myrlm %f")*). This flag flushes the symbols so that they are not loaded into TMS340 memory. If this flag is absent, the global symbols in the RLM are stored in TMS340 memory so that subsequently installed RLMs can reference these symbols. If this is not required — that is, if there are no RLMs to be installed that reference symbols in this RLM,— then using the *%f* flag enables the TMS340 memory to be used by the application's heap pool.

For more details on extensibility and the use of this function, see Chapter 8.

The function returns these error codes:

Error Code	Description
–3	**Out of Memory** — Not enough TMS340 memory to load the RLM.
–6	**Error Accessing RLM** — Unable to open RLM for reading. Either the spelling of the RLM filename does not match the RLM filename in the current directory, or the *–l* option of the TIGA environment variable is not set up correctly.
–10	**Symbol Reference** — An unresolved symbol was referenced by the RLM. Determine the name of the symbol, either by producing a link map for the RLM or by invoking TIGALNK from the command line using the *–ec* flag.

Error Code	Description (continued)
–14	**Error Loading RLM COFF File** — An error was obtained in the load of the RLM COFF file. Recreate the RLM and try again.

–15 **Out of Symbol Memory** — Not enough TMS340 memory to store the symbols of the RLM.

Example See subsection 8.3.1 on page 8-6.

Syntax

#include <tiga.h>

void install_usererror(function_name)
  void (*function_name)();

Type

Host-only

Description

The *install_usererror* function installs a host-resident user error function that is called when an error is encountered in the host communications. The default user error function simply prints a message to the screen when an error occurs. You can install another function to trap these errors and handle them accordingly. The user error function expects the following parameters:

```
usererror(command_number, error_code)
unsigned short command_number;
short error_code;
```

These error codes are passed to the error function:

1 Timeout with TMS340 communication on trying to load new function; that is, a previous function has not completed.

2 Timeout on waiting for TMS340 function to complete; that is, the function just invoked has not completed.

3 Parameter allocation failure; not enough memory to allocate a buffer to download data from the current function.

The return value from the installed handler tells TIGA whether to keep trying to determine if the TMS340 processor has completed (if nonzero) or to abort (if zero).

Example

```
#include <tiga.h>
#include <typedefs.h>
#include <extend.h>

#define nofpts 4000
short lotofpts[nofpts*2];

short far usererror(command_number, error_code)
unsigned short command_number;
short error_code;
{
  printf("TIGA error code of %d in command number %4x\n",
         error_code, command_number);
  return(1);
}
```

```
main()
{
  short i, x, y;
  CONFIG config;

  init_tiga(1);
  get_config(&config);
  install_usererror(usererror);
  /* initialize nofpts points to some value                 */
  x = y = 0;
  for (i = 0; i < nofpts*2; )
  {
    lotofpts[i++] = x;
    lotofpts[i++] = y;
    if (i % 4)
    {
      if (x++ > config.mode.disp_hres)
        x = 0;
    }
    else
    {
      if (y++ > config.mode.disp_vres)
        y = 0;
    }
  }
  /* set timeout value to 1 second                          */
  set_timeout(1000);
  set_pensize(64,64);
  /* tie up the TMS340 to get timeout since many points     */
  /* are being downloaded, use parameter alloc entry        */
  /* points to allocate a temporary command buffer          */
  /* for the data transfer from host to TMS340              */
  pen_polyline_a(nofpts, lotofpts);
  /* wait for TMS340 side to finish (to produce timeouts)   */
  synchronize();
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

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 of 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 <tiga.h>

main()
{
   long x;
   char s[80];

   init_tiga(0);
   clear_screen(-1);
   x = 1234;
   sprintf(s, "The leftmost 1 in %ld is bit number %d",
             x, lmo(x));
   text_out(10, 10, s);
   term_tiga();
}
```

Syntax

```
#include <tiga.h>

unsigned long loadcoff(filename)
    char *filename;
```

Type

Host-only

Description

The *loadcoff* function is not of general use to a TIGA application but is included here because the capability to load the graphics manager is an integral part of TIGA. With this function, TIGA provides a portable COFF loader across all TMS340 boards. This function loads the COFF file specified by argument *filename*. It returns false (0) if an error occurs during the load; otherwise, it returns the entry point address of the program. The entry point can be passed to the *gsp_execute* function to execute the COFF file.

Example

See *gsp_execute*.

Syntax

#include <tiga.h>

short page_busy()

Type

Core

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 monitor. When the processor has finished drawing, the new page is ready to be displayed on the screen in place of the old page. 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 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 3-D 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, you risk 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.

```
#include <tiga.h>
#define RADIUS 60      /* radius of circle of rotation       */
#define N 4            /* angular increment = 1>>N radians   */

main()
{
  short disp = 0, draw = 1;
  long x, y;

  init_tiga(0);
  x = (long)RADIUS << 16;
  y = 0;
  do
  {
    page_flip(disp, draw);
    x -= y >> N;
    y += x >> N;
    while (page_busy())
      ;
    clear_page(-1);
    text_out((x>>16)+RADIUS, (y>>16)+RADIUS, "*");
    disp ^= 1;
    draw ^= 1;
  }while(!kbhit());
  getch();
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

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 the 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 monitor. When the processor has finished drawing, the new page is ready to be displayed on the screen in place of the old page. 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 functions 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.

```
#include <tiga.h>
#include <extend.h>
#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 = 0, draw = 1;
  long x, y;

  init_tiga(1);
  set_ppop(XOR);
  x = (long)RADIUS << 16;
  y = 0;
  do
  {
    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);
    disp ^= 1;
    draw ^= 1;
  }while(!kbhit());
  getch();
  term_tiga();
}
```

Syntax

#include <tiga.h>

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

Type

Core

Description

The *peek_breg* function returns the contents of a TMS340 B-file register. Argument *breg* is a number in the range of 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.

```
#include <tiga.h>

main()
{
  char s[32];

  init_tiga(0);
  clear_screen(-1);
  sprintf(s, "COLOR1 = 0x%lx", peek_breg(9));
  text_out(10, 10, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

Description

The *poke_breg* function loads a 32-bit value into a B-file register. Argument *breg* is a number in the range of 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 TIGA graphics 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.

```
#include <tiga.h>
#include <extend.h>

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

Syntax

#include <tiga.h>

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

Type

Core

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 TIGA graphics library to draw the vertical lines.

```
#include <tiga.h>
#include <extend.h>

main()
{
  unsigned long i;
  short n;

  init_tiga(1);
  clear_screen(-1);
  for (i = 1; i < 128; i++)
  {
    n = rmo(i);
    fill_rect(1, 8*n, 10+i, 10);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

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.

```
#include <tiga.h>

main()
{
   unsigned long fcolor, bcolor;

   init_tiga(0);
   clear_screen(-1);
   get_colors(&fcolor, &bcolor);
   set_fcolor(bcolor);
   set_bcolor(fcolor);
   text_out(10, 10, "Swap COLOR0 and COLOR1.");
   term_tiga();
}
```

Syntax

```
#include <tiga.h>

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        */
```

Type

Core

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 TIGA graphics 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, the clipping window is set to its default limits, which is 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 that 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).

```
#include <tiga.h>
#include <extend.h>

main()
{
  short i;
  long x, y;

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

Syntax

```
#include <tiga.h>

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

Type

Core

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.

```
#include <tiga.h>

main()
{
    long white, black;

    init_tiga(0);
    clear_screen(-1);
    get_colors(&white, &black);
    set_colors(black, white);
    text_out(8, 8, "Black text on white background.");
    term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Host-only

Description

The *set_config* function configures the display system in the specified graphics mode. Both the display hardware and graphics software environment are initialized. Note that calling the *set_videomode* function, with the *mode* argument set to TIGA and the *style* argument set to INIT or INIT_GLOBALS, performs identical initialization for the default graphics mode as described here.

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 *set_config* function returns a 16-bit value, encoded as follows:

Bit	0:	Status	(0=Error, 1=OK)
Bit	1:	GM reloaded	(0=No, 1=Yes)
Bits	2–15:	0	

If an invalid *graphics_mode* argument is specified, zero is returned. Otherwise, bit 0 is set to 1.

Changing the graphics mode may change the memory map of the TMS340 side of TIGA. If this memory map change results in the alteration of the graphics manager (GM) load address, then *set_config* automatically reloads the GM and sets bit 1 of the return value to 1. If no reload is necessary, bit 1 is set to 0. Note that if the GM is reloaded, all downloaded extensions are flushed and allocated memory is freed. Therefore, it is recommended that an application use the *set_config* function before loading any TIGA extensions or allocating any TMS340 memory.

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 offscreen 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)
- The default graphics cursor, an arrow, is installed and selected.

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 <tiga.h>
#include <typedefs.h>
#include <extend.>
#define NFRAMES 120        /*delay in frames between modes   */

main()
{
  CONFIG cfg;
  char s[80];
  short mode, i, w, h;

  init_tiga(1);
  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);
    sprintf(s, "Graphics mode %d: %d-by-%d", mode, w, h);
    text_out(10, 10, s);
    for (i = NFRAMES; i; i--)  /* delay loop                 */
      wait_scan(h);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void set_curs_shape(shape);
   PTR shape;
```

Type

Core

Description

The *set_curs_shape* function selects a new cursor shape. Argument *shape* is a pointer to the CURSOR structure in TMS340 memory of the desired cursor. Before this function is called, both the cursor shape and mask data, and the cursor structure must be loaded into TMS340 memory using the *gsp_malloc* and *host2gsp* functions. The TMS340 memory address of the cursor shape data must be assigned to the data element of the cursor structure before the structure is loaded. The TMS340 memory address of the cursor structure can then be passed to this routine to select the cursor. A default cursor shape (an arrow) is installed with the graphics manager and is available until this routine is called to download a user cursor. The default cursor shape can be restored by invoking *set_curs_shape* with a *shape* argument of 0.

If the cursor is disabled when a call to *set_curs_shape* is made, the new cursor shape is not loaded immediately. Instead, the new cursor shape is loaded on the next call to s*et_curs_state(1*). For this reason, it is extremely important that the cursor shape and structure data resident in TMS340 memory not be freed or moved while the new cursor is being used.

On the other hand, if the cursor is enabled when *set_curs_shape* is called, the cursor shape is loaded immediately, and the new cursor shape and structure data may be removed safely from TMS340 memory (assuming this cursor shape will never need to be selected again by the application).

In the *set_curs_xy* function, (x, y) is the position of the top-left pixel of the cursor if *hot_x* and *hot_y* are zero. These values are subtracted from the current cursor position to give the top-left hand corner of the cursor's starting drawing point. For example, in a simple crosshairs cursor of width16 pixels and height 12 pixels, the *hot_x* is set to width/2, that is, 8; and similarly, *hot_y* is set to 6. If the current cursor position is (320, 240), the rectangle defining the cursor is drawn with its top left hand corner at 320 – *hot_x* and 240 – *hot_y*, that is (312, 236). This puts the center of the crosshair cursor at position (320, 240), the desired cursor position.

The data that defines the cursor consists of (1) cursor mask data, and (2) cursor shape data. This data defining the cursor shape must be contiguous; that is, the cursor shape data must immediately follow the cursor mask data. The pitch of this cursor data is indicated by the pitch element in the CURSOR structure.

Two raster operators, *mask_rop* and *shape_rop*, determine how the cursor mask data and cursor shape data, respectively, are expanded onto the screen. For the mask and shape data, the background color is always 0. The foreground color of the shape and mask are specified by the *color* and *mask_color* fields, respectively, of the CURSOR structure.

An example of cursor data follows. The mask data consists of an array width by height with 0s where the cursor is located and 1s elsewhere. The raster op for this data is AND(1), no transparency. The shape data is an array width by height with 1s where the cursor is located and 0s elsewhere. The raster op for the shape data is OR(8), no transparency. Typically, the shape of the cursor in the mask data is one pixel wider than that of the shape data. This enables the cursor outline to be seen when placed over a background of the same color as the cursor shape.

Example Example masks for a simple crosshair cursor:

```
11111111111
11110001111
11110001111
11110001111
10000000001
10000100001
10000000001
11110001111
11110001111
11110001111
11111111111
```

MASKDATA

```
00000000000
00000000000
00000100000
00000100000
00000100000
00111011100
00000100000
00000100000
00000100000
00000000000
00000000000
```

SHAPEDATA

```
#include <tiga.h>
#include <typedefs.h>
#include <extend.h>

#include <dos.h>
#include <conio.h>

#define ESC 0x1b

CONFIG config;

/* Shape data for pencil cursor                                      */
char far PencilData[] =
{
0xFF,0x87,0x03,0x00,0xFF,0x03,0x03,0x00,0xFF,0x03,0x02,0x00,0xFF,0x01,0x02,0x00,
0xFF,0x01,0x03,0x00,0xFF,0x00,0x03,0x00,0xFF,0x80,0x03,0x00,0x7F,0x80,0x03,0x00,
0x7F,0xC0,0x03,0x00,0x3F,0xC0,0x03,0x00,0x3F,0xE0,0x03,0x00,0x1F,0xE0,0x03,0x00,
0x1F,0xF0,0x03,0x00,0x0F,0xF0,0x03,0x00,0x0F,0xF8,0x03,0x00,0x07,0xF8,0x03,0x00,
```

```
0x07,0xFC,0x03,0x00,0x03,0xFC,0x03,0x00,0x03,0xFE,0x03,0x00,0x01,0xFE,0x03,0x00,
0x01,0xFF,0x03,0x00,0x00,0xFF,0x03,0x00,0x80,0xFF,0x03,0x00,0xC0,0xFF,0x03,0x00,
0xE0,0xFF,0x03,0x00,0xF0,0xFF,0x03,0x00,0xF8,0xFF,0x03,0x00,0xFD,0xFF,0x03,0x00,
0x00,0x00,0x00,0x00,0x00,0x78,0x00,0x00,0x00,0xF8,0x00,0x00,0x00,0xFC,0x00,0x00,
0x00,0x7C,0x00,0x00,0x00,0x72,0x00,0x00,0x00,0x26,0x00,0x00,0x00,0x39,0x00,0x00,
0x00,0x11,0x00,0x00,0x80,0x10,0x00,0x00,0x80,0x08,0x00,0x00,0x40,0x08,0x00,0x00,
0x40,0x04,0x00,0x00,0x20,0x04,0x00,0x00,0x20,0x02,0x00,0x00,0x10,0x02,0x00,0x00,
0x10,0x01,0x00,0x00,0x08,0x01,0x00,0x00,0x88,0x00,0x00,0x00,0x84,0x00,0x00,0x00,
0x44,0x00,0x00,0x00,0x4E,0x00,0x00,0x00,0x3E,0x00,0x00,0x00,0x1E,0x00,0x00,0x00,
0x0E,0x00,0x00,0x00,0x06,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};

/* Pencil cursor structure                                    */
CURSOR far pencil =
{
0x0000, 0x001B, 0x0011, 0x001C, 0x0020, 0x0FL, 1, 8, 0x0FFFFFFFFL, 0x0L
};

/* Shape for Arrow2 cursor                                    */
unsigned short far Arrow2Data[] =
{
0x0010, 0x0018, 0x001C, 0xFFFE, 0xFFFF, 0xFFFE, 0x001C, 0x0018, 0x0010,
0x0000, 0x0000, 0x0008, 0x000C, 0x7FFE, 0x000C, 0x0008, 0x0000, 0x0000
};

/* Arrow2 cursor structure                                    */
CURSOR far Arrow2 =
{
00x0000, 0x0004, 0x0010, 0x0009, 0x0010, 0x0FL, 0x0020, 0x0020, 0x0FFFFFFFFL, 0x0L
};

struct
{
  short x,y;          /* coordinates                          */
  short left, right;  /* buttons                              */
  short x1,y1, x2,y2; /* boundary                             */
}mouse;

union REGS regs;

/* this function checks if a mouse driver is installed       */
check_mouse()
{
  regs.x.ax = 0;
  int86(0x33,&regs,&regs);
  return(regs.x.ax);
}

mouse_driver()
{
  /* get mouse coordinates                                    */
  regs.x.ax = 11;
  int86(0x33,&regs,&regs);
  mouse.x += regs.x.cx;
  mouse.y += regs.x.dx;


  /* ensure the mouse stays within the screen boundary        */
  if (mouse.x < mouse.x1)
    mouse.x = mouse.x1;
  if (mouse.x > mouse.x2)
    mouse.x = mouse.x2;
  if (mouse.y < mouse.y1)
    mouse.y = mouse.y1;
```

```
    if (mouse.y > mouse.y2)
      mouse.y = mouse.y2;

    /* Tell the TMS340 cursor                                        */
    set_curs_xy(mouse.x, mouse.y);

    /* get the mouse buttons                                         */
    regs.x.ax = 3;
    int86(0x33,&regs,&regs);
    mouse.left = regs.h.bl & 1;
    mouse.right = (regs.h.bl & 2) >> 1;
  }

  void install_cursor(type)
  short type;           /* 0=default(arrow), 1=user(pencil),
                                2=Arrow2                             */
  {
    static PTR gpUserCurs1 = 0L; /* Address of user cursor1
                                    in TMS340 mem                    */
    static PTR gpUserCurs2 = 0L; /* Address of user cursor2
                                    in TMS340 mem                    */

    CURSOR *hpCursStruct;
    void *hpCursData;
    PTR *gpCursStruct;

    switch(type)
    {
      case 1:           /* Pencil cursor                             */
        hpCursStruct = &pencil;
        hpCursData = (void *)PencilData;
        gpCursStruct = (PTR *)&gpUserCurs1;
        break;
      case 2:           /* Arrow2 cursor                             */
        hpCursStruct = &Arrow2;
        hpCursData = (void *)Arrow2Data;
        gpCursStruct = (PTR *)&gpUserCurs2;
        break;
      default:
        set_curs_shape((PTR)0);/* Default cursor                     */
        return;
    }

    if(*gpCursStruct == 0L)
    {
      unsigned short num_bytes;

      /* download cursor shape data to TMS340                       */
      num_bytes = ((hpCursStruct->height *
                    hpCursStruct->pitch) << 1) >> 3;
      hpCursStruct->data = (PTR)gsp_malloc(num_bytes);
      host2gsp(hpCursData, hpCursStruct->data, num_bytes, 0);

      /* download cursor structure to TMS340                        */
      num_bytes = sizeof(CURSOR);
      *gpCursStruct = (PTR)gsp_malloc(num_bytes);
      host2gsp(hpCursStruct, *gpCursStruct, num_bytes, 0);
    }
    set_curs_shape( *gpCursStruct );
  }

  main()
  {
    char key;
```

```
    short CursorType = 1, fgc = 0;

    if(!check_mouse())
    {
      printf("Mouse driver needs to be installed to run this
                       example\n");
      exit(0);
    }
    init_tiga(1);
    printf("Press...\n");
    printf(" ESC to quit\n");
    printf(" SPACE to toggle cursor shapes\n");
    printf(" LEFT mouse button to draw points\n");
    printf(" C to change cursor color\n");

    /* assign a new cursor shape                             */
    install_cursor(CursorType);

    /* initialize mouse to the center of the screen          */
    get_config(&config);
    mouse.x = config.mode.disp_hres>>1;
    mouse.y = config.mode.disp_vres>>1;
    set_curs_xy(mouse.x, mouse.y);

    /* intialize mouse boundary                              */
    mouse.x1 = mouse.y1 = 0;
    mouse.x2 = config.mode.disp_hres - 1;
    mouse.y2 = config.mode.disp_vres - 1;

    /* Turn on cursor                                        */
    set_curs_state(1);

    for(;;)
    {
      /* move the cursor with the mouse                      */
      mouse_driver();
      /* if left button pressed draw a point                 */
      if (mouse.left)
        draw_point(mouse.x, mouse.y);
      if(kbhit())
        switch(getch())
        {
          case ' ' :
            if(++CursorType > 2)
              CursorType = 0;
            install_cursor(CursorType);
            break;
          case 'c' :
          case 'C' :
            if(++fgc == config.mode.palet_size)
              fgc = 1;
            if(CursorType==2)
              set_cursattr(fgc,0x0FFFFFFFFL,0x0020,0x0020);
            else
              set_cursattr(fgc,0x0FFFFFFFFL,8,1);
            break;
          case ESC :
            set_curs_state(0);/* Turn cursor off             */
            term_tiga();
        }
    }
}
```

Syntax

```
#include <tiga.h>

void set_curs_state(enable)
    short enable;
```

Type

Core

Description

The *set_curs_state* function enables (displays) the cursor (if *enable* is non-zero) or disables it (if *enable* is zero).

Example

See *set_curs_shape*.

Syntax

```
#include <tiga.h>

void set_curs_xy(x, y)
    short x;
    short y;
```

Type

Core

Description

The *set_curs_xy* function modifies the pixel coordinates of the cursor's hotspot.The cursor coordinates (arguments *x* and *y*) are *not* relative to the drawing origin; they are always relative to the top left-hand corner of the screen.

Example

See *set_curs_shape*.

Syntax

```
#include <tiga.h>

void set_cursattr(shape_c, mask_c, shape_a, mask_a)
    unsigned long shape_c;      /* shape (foregnd.) curs. color    */
    unsigned long mask_c;       /* mask (backgnd.) cursor color    */
    unsigned short shape_a;     /* cursor shape attributes         */
    unsigned short mask_a;      /* cursor mask attributes          */
```

Type

Core

Description

The *set_cursattr* function changes the display attributes of the current active cursor. Only the attributes for the cursor currently selected are modified.

The *shape_c* and *mask_c* arguments specify the shape (foreground) and mask (background) colors, respectively. The values specified by these arguments are replicated by the current pixel size before use by the cursor routines. The *mask_a* and *shape_a* arguments define the raster op and transparency modes used when drawing the cursor on the screen. Each is a 16-bit value with bit fields defined as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										te	ppop				

bits	0–4	ppop (see *TMS34010 User's Guide*, p. 6–13)
bit	5	transparency enable (0=disable,1=enable)
bits	6–15	reserved for future use

Example

See *set_curs_shape*.

Syntax

```
#include <tiga.h>

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

Type

Core

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.

```
#include <tiga.h>

main()
{
  unsigned long fcolor, bcolor;

  init_tiga(0);
  clear_screen(-1);
  get_colors(&fcolor, &bcolor);
  set_fcolor(bcolor);
  set_bcolor(fcolor);
  text_out(10, 10, "Swap COLOR0 and COLOR1.");
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short set_interrupt(level, priority, enable, scan_line)
    short level;
    short priority;
    short enable;
    short scan_line;
```

Type

Core

Description

The *set_interrupt* function enables/disables a previously installed interrupt service routine. The routine must have been installed via the *install_rlm* function or the combination of *create_alm* and *install_alm*.

Argument *level* indicates the interrupt level where the interrupt routine was installed. When the interrupt is installed, the *priority* is returned by the *get_isr_priorities* function to distinguish between different interrupt service routines on the same interrupt level. If *enable* is true (nonzero), the interrupt is enabled; otherwise, it is disabled.

Argument *scan_line* is valid only for display interrupts (interrupt level 10). It is used to set the line at which the interrupt occurs. Argument *scan_line* is specified in the range of 0 to VTOTAL–1, where VTOTAL is the total number of lines in the frame. Note that a *scan_line* value of 0 does not necessarily correspond to the top line of the visible screen. For further information, consult the video timing chapter in the TMS34010 or TMS34020 user's guide. If the *scan_line* argument is –1, then the value for the *scan_line* is taken to be that passed in the previous invocation of *set_interrupt*. This allows the interrupt to be enabled/disabled without respecifying the *scan_line* parameter.

The *set_interrupt* function returns true (nonzero) if the interrupt is set correctly; it returns false (zero) otherwise.

For more details on extensibility and the use of this function, see Chapter 8.

Example

See Section 8.9.

Syntax

```
#include <tiga.h>

short set_module_state(module_id, flags)
    short module_id;   /* Module identifier                 */
    short flags;       /* State flags                       */
```

Type

Core

Description

The *set_module_state* function is used to set the state of the module specified by the argument *module_id*.

The module identifier returned from the *install_rlm* and *install_alm* functions is used as the *module_id* argument for this function. If you wish to modify the state of the TIGA graphics library module after loading it via the *install_primitives* function (which does not return a module identifier), use the constant GRAPHICS_LIB_ID provided in the *tiga.h, tiga.hch, and tiga.ndp* include files as the *module_id* argument.

If an invalid module identifier is passed, or if the module corresponding to the specified *module_id* is currently not loaded, then the *set_module_state* function returns FALSE (0). Otherwise, it returns TRUE, indicating no errors.

The argument *flags* contains state information to be assigned to the specified module. These flags are currently supported:

- Bit 0 0=unlock module / 1=lock module

  Locking a downloaded module protects it from being flushed whenever the TIGA memory management system is initialized. This initialization occurs when
  - The *set_videomode()* function is called with either the INIT or INIT_GLOBALS style argument specified, or
  - The *gsp_minit()* function is called.

  Conversely, to enable flushing of a previously locked module, the module must first be unlocked by clearing bit 0 of the *flags* argument and calling the *set_module_state* function.

- Bits 1–15 Reserved for future use

Example

```
#include <tiga.h>
#include <extend.h>

main()
{
  /*-------------------------------------------------------------------------- *
/
  /* Initialize TIGA environment and load extended graphics library            */
  /*-------------------------------------------------------------------------- *
/
  init_tiga(1);
  /*-------------------------------------------------------------------------- *
/
  /* Lock the graphics lib module. Module should be present                    */
  /*-------------------------------------------------------------------------- *
/
  set_module_state(GRAPHICS_LIB_ID, 1);
  printf(”After lock: RLM %spresent\n”,
        function_implemented(DRAW_LINE) ? ” ” : ”not ”);
  /*-------------------------------------------------------------------------- *
/
  /* Free memory. Module should still be present because it                   */
  /* is currently locked                                                       */
  /*-------------------------------------------------------------------------- *
/
  gsp_minit(-1);
  printf(”After gsp_minit(-1): RLM %spresent\n”,
        function_implemented(DRAW_LINE) ? ” ” : ”not ”);
  /*-------------------------------------------------------------------------- *
/
  /* Unlock module. Module is still present but can now be                     */
  /* flushed                                                                   */
  /*-------------------------------------------------------------------------- *
/
  set_module_state(GRAPHICS_LIB_ID, 0);
  printf(”After unlock: RLM %spresent\n”,
        function_implemented(DRAW_LINE) ? ” ” : ”not ”);
  /*-------------------------------------------------------------------------- -*
/
  /* Free memory. Module should not be present now                             */
  /*-------------------------------------------------------------------------- *
/
  gsp_minit(-1);
  printf(”After gsp_minit(-1): RLM %spresent\n”,
                    function_implemented(DRAW_LINE) ? ” ” : ”not ”);

  term_tiga();        /* Terminate the TIGA environment                        */
}
```

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void set_palet(count, index, palet)
   long count;          /* number of palette entries      */
   long index;          /* index to starting entry        */
   PALET *palet;        /* list of palette data           */
```

Type

Core

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. The PALET structure contains the following fields:

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

Refer to the PALET structure description in Appendix A for detailed descriptions of each field.

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 *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 the 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 TIGA graphics 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 <tiga.h>
#include <typedefs.h> /* defines PALET struct                */
#include <extend.h>

main()
{
  short n;
  PALET p[16];

  init_tiga(1);
  clear_screen(-1);
  for (n = 0; n < 16; n++)
    p[n].r = p[n].g = p[n].b = p[n].i = 16*n;
    set_palet(16, 0, p);
  for (n = 0; n < 16; n++)
  {
    set_fcolor(n);
    fill_rect(12, 80, 8+12*n, 8);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

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

Type

Core

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, it returns a nonzero value.

In systems that store the palette data in display memory (such as those using the TMS34070 color palette), this function updates the 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 functions 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 <tiga.h>
#include <extend.h>

main()
{
  short n;

  init_tiga(1);
  clear_screen(-1);
  for (n = 0; n < 16; n++)
    set_palet_entry(n, 16*n, 16*n, 16*n, 16*n);
  for (n = 0; n < 16; n++)
  {
    set_fcolor(n);
    fill_rect(12, 80, 8+12*n, 8);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

void set_pmask(pmask)
    unsigned long pmask;        /* plane mask                  */
```

Type

Core

Description

The *set_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, as is, from the source pixel.

The plane mask is set to its default value of 0 during initialization of the drawing environment. 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 <tiga.h>
#include <typedefs.h>       /* defines CONFIG and FONTINFO    */
#define MINPSIZE      4     /* minimum pixel size             */

main()
{
  CONFIG cfg;
  FONTINFO fntinf;
  unsigned long pmask;
  short x, y;
  char s[80];

  init_tiga(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);
    sprintf(s, "All planes enabled except %d", lmo(pmask));
    text_out(x, y, s);
    y += fntinf.charhigh;

    /* Disable all planes except one                          */
    set_pmask(~pmask);
    sprintf(s, "All planes enabled except %d", lmo(pmask));
    text_out(x, y, s);
    y += fntinf.charhigh;
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

void set_ppop(ppop)
    short ppop;              /* pixel processing operation code      */
```

Type

Core

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 of 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 of 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 4–2, Boolean operations are in the range of 0 to 15, and arithmetic operations are in the range of 16 to 21.

Table 4–2. 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 the drawing environment is initialized, the PPOP code is set to its default value of 0 (*replace* operation). The PPOP code can be read with a call to the *get_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 TIGA graphics library to fill two rectangles that partially overlap. The overlapping region shows the effect of exclusive-ORing identical source and destination pixel values.

```
#include <tiga.h>
#include <extend.h>
#define XOR 10        /* pixel processing operation code    */

main()
{
  init_tiga(1);
  clear_screen(–1);

  set_ppop(XOR);
  fill_rect(100, 20, 10, 50);
  fill_rect(20, 100, 50, 10);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

void set_text_xy(x, y)
   short x, y;                  /* text x-y coordinates        */
```

Type

Core

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.

```
#include <tiga.h>

main()
{
   init_tiga(0);
   clear_screen(-1);
   set_text_xy(10, 20);
   text_outp("hello, world");
   term_tiga();
}
```

Syntax

```
#include <tiga.h>

void set_timeout(value)
    short value;             /* value in milliseconds            */
```

Type

Host-only

Description

The *set_timeout* function sets the timeout value (in milliseconds) that determines how long the host waits for a TMS340 function to complete before calling the error function with a timeout error. The user can ignore timeouts altogether by installing a user error handler function that is called when the timeout occurs. This function can be made to ignore such timeouts.

Example

See *install_usererror*.

Syntax

```
#include <tiga.h>

void set_transp(mode)
    short mode;                    /* transparency mode          */
```

Type

Core

Description

The *set_transp* function, if implemented, changes the transparency mode. When transparency is enabled, this mode determines how a pixel is defined as 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, the system is configured in transparency mode 0, which is the default.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

PTR set_vector(trapnum, gptr)
        short trapnum;       /* trap number                */
        PTR gptr;            /* pointer to TMS340 memory   */
```

Type

Core

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 of –32768 to 32767 for a TMS34020, and in the range of 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.

Syntax

```
#include <tiga.h>

short set_videomode(mode, style)
    unsigned short mode;
    unsigned short style;
```

Type

Host-only

Description

The *set_videomode* function sets up the video mode to be used. Every TIGA application should call this function (after calling *tiga_set* to initialize the TIGA environment), with a mode of TIGA, before invoking any other TIGA function.

The following values, provided in *tiga.h*, are valid mode arguments:

TIGA	CGA
OFF_MODE	VGA
PREVIOUS	AI_8514
MDA	EGA
HERCULES	

TIGA is a high-resolution mode supported by the board. OFF_MODE is used on systems that provide no videomode for the DOS screen (such as EGA); when not running TIGA, such boards are in the OFF_MODE. PREVIOUS is the mode that the board was in before the current mode. All the other modes (MDA, HERCULES, CGA, etc.) provide the graphics modes for DOS. They are either provided by separate hardware or are emulated by the TMS340 processor.

All TIGA applications should call *set_videomode* with *TIGA* mode upon starting. They should then call *set_videomode* again at the end of their program to restore the video mode (since in many cases the board on which TIGA is being run is not the primary video board). The mode selected could be PREVIOUS, which restores the mode set in the last call to *set_videomode*. However, if a particular application wants to switch back and forth between several modes, it is recommended that a call be made to *get_videomode* and that the mode be saved by the application.The saved mode can be used to terminate the TIGA application and to restore the board to the initial state.

If a call is made to *set_videomode* specifying a video mode not supported by the board, the function returns false (0). Otherwise, it returns true (1), indicating successful completion.

The *style* argument is used to determine the manner in which the mode is set up on entry. These are the valid styles:

NO_INIT	Used during an application to switch between TIGA and other modes. It enters TIGA, leaving all global variables intact.
INIT_GLOBALS	Initializes the global variables only, by calling *set_config* with the *init_draw* flag true and by restoring the timeout val-

	ue and the user error handler. The heap pool is retained, which keeps any downloaded extensions installed.
INIT	Initializes global variables and dynamic memory (heap pool).This frees all allocated pointers and thus deletes all unsecured downloaded extensions.
INIT_GM	Forces the TIGA Graphics Manager to be reloaded, initializes GM global variables, and frees all allocated pointers and downloaded TIGA extensions.

The state of the graphics manager is checked by the *set_videomode* function with a mode of *TIGA*. If the graphics manager is not loaded or is corrupted and the specified style argument is INIT or INIT_GLOBALS, *set_videomode* loads and executes it.

The *style* argument contains two additional options, which can be selected by ORing with the above style parameter:

CLR_SCREEN	Clears the screen with zeros when specified. It should be specified at initialization when you are using the INIT_GLOBALS or INIT styles. The screen is blanked while the video registers are initialized. Note that ***all*** display memory is cleared to 0 (including any offscreen areas) when specifying CLR_SCREEN. In other words, specifying CLR_SCREEN is functionally equivalent to calling the *clear_frame_buffer()* function.
NO_ENABLE	This parameter inhibits switching the video output from the current mode to TIGA. It is valid only when TIGA is specified as the mode argument. For example, assuming a single monitor configuration and a current video mode of VGA, calling *set_videomode(TIGA, INIT | NO_ENABLE)* would enable you to call any TIGA core function but would not switch the video output from the VGA board to the TIGA board.

Example See the *init_tiga* and *term_tiga* function listings in Section 3.4, page 3-6.

Syntax

```
#include <tiga.h>

void set_windowing(mode)
    short mode;
```

Type

Core

Description

The *set_windowing* function loads the specified value into the 2-bit windowing field contained in the CONTROL I/O register.

The four windowing modes are

1) 00_2 No windowing.

2) 01_2 Interrupt request on write in window.

3) 10_2 Interrupt request on write outside window.

4) 11_2 Clip to window.

Take care in using this function. TIGA'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 of the extended graphics library functions. 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, the system is configured in windowing mode 3, which is the default.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void set_wksp(addr, pitch)
        PTR addr;           /* starting address       */
        PTR pitch;          /* workspace pitch        */
```

Type

Core

Description

The *set_wksp* function specifies an offscreen workspace. None of the current TIGA core or extended functions 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 offscreen workspace. Argument *pitch* is the difference in memory addresses of two adjacent rows in the offscreen 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 offscreen workspace is a 1-bit-per-pixel bit map of the same width and height as the screen. If the display hardware provides sufficient offscreen memory, the workspace can be allocated statically. By convention, the workspace pitch retrieved by the *get_wksp* function is nonzero when a workspace is allocated; following initialization the pitch can be checked 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 offscreen workspace. For this reason, proprietary extensions to the core functions library that require use of the workspace may be unable to execute on some systems.

Syntax

```
#include <tiga.h>
#include <typedefs.h>

void setup_hostcmd( hinit )
    HOST_INIT *hinit;          /* Far ptr. to HOST_INIT struc.            */
```

Type

Host-only

Description

The *setup_hostcmd* function initializes the TIGA call-back environment, enabling host-resident functions to be called from the TMS340 processor.

The argument *hinit* is a far pointer to a structure of type HOST_INIT. The elements of this structure are defined in the *typedefs.h* include file. No value is returned by this function.

See Section 8.8, page 8-31, for detailed information on how to use this and other functions related to the TIGA call-back feature.

Syntax

```
#include <tiga.h>

short sym_flush(module_id);
    short module_id;
```

Type

Core

Description

The *sym_flush* function flushes the symbols associated with the module specified by the argument *module_id*. Argument *module_id* is returned by the *install_rlm* or *install_alm* function when the module is loaded. Use the *module_id* GRAPHICS_LIB_ID provided in the *tiga.h, tiga.hch,* and *tiga.ndp* include files to specifiy the TIGA extended graphics library module.

If *module_id* is –1, then the symbols associated with all unsecured installed modules are flushed. The TIGA graphics manager and core function symbols are retained.

The function returns true (1) if the symbols were flushed successfully. A return value of false (0) indicates an error, caused by either of these conditions: a *module_id* is invalid, or the module specified by *module_id* is not loaded or is secured.

Flushing symbols frees TMS340 memory, increasing the memory available for allocation by the TIGA application. However, installing at a later time any user-extended module that references previously flushed symbols results in a symbol-referencing load error.

Syntax

```
#include <tiga.h>

void synchronize()
```

Type

Host-only

Description

The *synchronize* function ensures that the TMS340 completes all pending operations before it returns. TIGA supports two-processor execution, and some conditions require the two processors to be synchronized. For instance, if the host downloads data that is being manipulated by the TMS340, it is essential that the TMS340 finishes with it before the host overwrites the data.

Example

See *install_usererror*.

Syntax

```
#include <tiga.h>

short text_out(x, y, s)
   short x, y;        /* starting coordinates    */
   char *s;           /* character string        */
```

Type

Core

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, 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.

```
#include <tiga.h>

main()
{
   init_tiga(0);
   clear_screen(-1);
   text_out(10, 10, "Hello world.");
   term_tiga();
}
```

Syntax

```
#include <tiga.h>

void text_outp(s)
   char *s;
```

Type

Core

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 —TI Roman size 20 and TI Helvetica size 22 —in the same line of text. Use the *set_textattr* function to align the text to the base line.

Note that the function *loadinst_font* is called to load and install a TIGA font from a font file. The *loadinst_font* is not a TIGA function but does make calls to various TIGA functions to load a font. Refer to the *install_font* function description in Chapter 5 for a complete source listing of the *loadinst_font* function.

```
#include <tiga.h>
#include <typedefs.h>
#include <extend.h>

static FONTINFO fontinfo;

main()
{
  short i, j;

  init_tiga(1);
  clear_screen(-1);
  i = loadinst_font("ti_rom20.fnt");
  j = loadinst_font("ti_hel22.fnt");
  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.");
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

short tiga_busy(void )
```

Type

Host-only

Description

The *tiga_busy* function returns a status code indicating whether TIGA is immediately ready to process a host-initiated TIGA command. If so, 0 is returned. However, if TIGA is busy,1 is returned.

When a host application calls a TIGA core or extended function, the function request is queued up in the communication buffer queue. This queue is serviced by the TIGA graphics manager (GM), which pulls the function request information out of the buffer and executes the specified command. There may be instances when the GM cannot service the functions as fast as the host is requesting them. In this case, the TIGA communication driver (CD, on the host side of TIGA) waits until a free communications buffer is available, before writing the requested function information and returning control to the application. You may find this dead time useful for performing other tasks while the GM is processing queued commands.

Note that host-only TIGA functions are always executed immediately by the CD. Thus, the return value of *tiga_busy* is valid only for core or extended functions.

Syntax

```
#include <tiga.h>

long tiga_set(mode)
    short mode;              /* Mode or command                */
```

Type

Host-only

Description

The *tiga_set* function initializes communications with the TIGA communications driver and sets up the TIGA device to a known state. It should be the first function called in a TIGA application.

The *mode* argument specifies the state of the TIGA communication driver or requests that information relating to the current TIGA environment be returned.

These are valid values for the *mode* argument:

- mode = CD_OPEN (1)

  Initialize the TIGA communication driver. If a protected mode TIGA environment is required (that is, the TIGA application is running in protected mode), this is also initialized. Valid return values are

0	Initialization OK
–4	TIGA CD is not installed
–25	TIGA board communications init failure

- mode = CD_CLOSE (0)

  Restore the PC environment to that before *tiga_set*(*CD_OPEN*) was called. Valid return values are

0	Success
–26	TIGA CD was not open

- mode = CD_STATUS (2)

  Return information about the current TIGA CD environment in the 32-bit return value as follows:

Bit	0	CD State: 0=closed / 1=open
Bit	1	CD operating mode: 0=Real / 1=Protected mode
Bit	2	CD communications: 0=Polled / 1=HW interrupt
Bits	3–15	Reserved
Bits	16–23	TIGA CD Minor revision level
Bits	24–31	TIGA CD Major revision level

The *tiga_set* function is new to TIGA 2.0. In previous versions of TIGA, the TIGA environment became active when a call to *cd_is_alive* was executed, and it remained active until the application terminated. Most applications activated the TIGA environment through calls to *set_videomode*, which indirectly executed the function *cd_is_alive*.

Calling any other TIGA function before calling *tiga_set(CD_OPEN*) results in an error return. In addition, the *cd_is_alive* function is no longer available to the application developer, because its functionality is a subset of the *tiga_set* function.

Note that applications linked with the TIGA 1.1 AI library are not affected by the addition of the *tiga_set* function.

Example

See the *init_tiga* and *term_tiga* function listings in Section 3.4, page 3-6.

Syntax

#include <tiga.h>

void transp_off()

Type

Core

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, 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 TIGA graphics 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 <tiga.h>
#include <typedefs.h>      /* defines FONTINFO structure     */
#include <extend.h>

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

  init_tiga(1);
  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.”);
  term_tiga();
}
```

Syntax

#include <tiga.h>

void transp_on()

Type

Core

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, 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 TIGA graphics 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 <tiga.h>
#include <typedefs.h>      /* defines FONTINFO structure     */
#include <extend.h>

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

  init_tiga(1);
  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.");
  term_tiga();
}
```

Syntax

```
#include <tiga.h>

void wait_scan(line)
    short line;             /* scan line number                */
```

Type

Core

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 of 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 an animated sequence of frames is being drawn, 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 <tiga.h>
#include <typedefs.h>      /* defines FONTINFO structure     */
#define RADIUS  60         /* radius of revolution           */

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

  init_tiga(0);
  clear_screen(−1);
  get_fontinfo(−1, &fntinf);
  x = (long)RADIUS << 16;
  y = 0;
  i = j = 0;
  do
  {
    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;
  }while(!kbhit());
  getch();
  term_tiga();
}
```

Chapter 5

Extended Graphics Library Functions

This chapter discusses the extended functions alphabetically. Each discussion

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

Before you use the functions described in this chapter, you must first install them by calling the *install_primitives* function, described on page 4-82.

The examples in this chapter use the functions *init_tiga* and *term_tiga* to initialize and terminate the TIGA environment. Although the *init_tiga* and *term_tiga* functions are not actually TIGA functions, they do make calls to various TIGA functions.The *init_tiga* function initializes the TIGA environment and is called before any other TIGA function. The *term_tiga* function terminates a TIGA application by restoring the previous video mode and closing the TIGA communication driver. Refer to Section 3.4, page 3-6, for a sample TIGA application that illustrates the *init_tiga* and *term_tiga* functions.

5.1 Extended Graphics Library Functions

The following is an alphabetical table of contents for functions reference.

Function	Page
bitblt	5-4
decode_rect	5-7
delete_font	5-9
draw_line	5-11
draw_oval	5-12
draw_ovalarc	5-13
draw_piearc	5-15
draw_point	5-17
draw_polyline	5-18
draw_rect	5-20
encode_rect	5-21
fill_convex	5-25
fill_oval	5-27
fill_piearc	5-28
fill_polygon	5-30
fill_rect	5-32
frame_oval	5-33
frame_rect	5-34
get_env	5-35
get_pixel	5-37
get_textattr	5-38
in_font	5-40
install_font	5-41
move_pixel	5-43
patnfill_convex	5-44
patnfill_oval	5-46
patnfill_piearc	5-47
patnfill_polygon	5-49
patnfill_rect	5-51
patnframe_oval	5-52
patnframe_rect	5-54
patnpen_line	5-56
patnpen_ovalarc	5-57
patnpen_piearc	5-59
patnpen_point	5-61
patnpen_polyline	5-62
pen_line	5-64
pen_ovalarc	5-65
pen_piearc	5-67
pen_point	5-69
pen_polyline	5-70
put_pixel	5-72
seed_fill	5-73
seed_patnfill	5-75
select_font	5-77
set_draw_origin	5-78
set_dstbm	5-79
set_patn	5-81

set_pensize 5-83
set_srcbm 5-84
set_textattr 5-86
styled_line 5-88
styled_oval 5-90
styled_ovalarc 5-92
styled_piearc 5-94
swap_bm 5-96
text_width 5-97
zoom_rect 5-98

Syntax

```
#include <tiga.h>
#include <extend.h>

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 bitmap to the current destination bit map. The source and destination bitmaps are specified by calling the *set_srcbm* and *set_dstbm* functions before calling the *bitblt* function. Calling the *set_videomode* function with the *style* argument set to INIT or INIT_GLOBALS causes both the source and destination bitmaps to be set to the default bitmap, 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 identical in 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 bitmap. 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 bitmap.

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 bitmap'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 bitmap are expanded to the current foreground color before being drawn to the destination; 0s are expanded to the current background color.

If the destination bitmap'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 bitmap is the screen, the specified source or destination coordinates are defined relative to the current drawing origin. In the case of a linear bitmap 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 bitmap is set to the screen, the function clips the destination bitmap to the current rectangular clipping window. When the source bitmap 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 bitmap 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 bitmap; responsibility for clipping is left to the calling routine.

If both source and destination bitmaps are set to the screen, then the function correctly handles the case in which the rectangular areas containing the source and destination bitmaps 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 bitmap to the screen. The original image is 16 pixels wide and 40 pixels high; it has a pitch of 16. Use the *zoom_rect* function to zoom the screen image by a factor of 5.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define PITCH 16
#define W 16
#define H 40
#define IMAGEDEPTH 1
#define ZOOM 5

static unsigned short image[] =
{
   0x00F00, 0x01F80, 0x07FC0, 0x031F0, 0x010C0, 0x01080, 0x00880, 0x00700,
   0x01FC0, 0x07FE0, 0x0FFF0, 0x0FFF8, 0x0FFF8, 0x0FFFC, 0x0FFFC, 0x0FFFC,
   0x0FFFC, 0x0FFFC, 0x0FFFC, 0x0FFFC, 0x0FFF8, 0x0FFF8, 0x0FFB8, 0x0FFD0,
   0x0FFE0, 0x0BFE0, 0x0BDE0, 0x07DE0, 0x03DE0, 0x03DE0, 0x03DE0, 0x03DE0,
   0x03DE0, 0x03DE0, 0x03DE0, 0x03DE0, 0x03DE0, 0x03DE0, 0x07DF0, 0x0F8F8
};
```

```
main()
{
  PTR image_ptr, rowbuf;
  CONFIG c;

  init_tiga(1);
  clear_screen(0);
  /* Allocate memory for image                                   */
  image_ptr = gsp_malloc(sizeof(image));
  /* Copy image to TMS340 memory                                 */
  host2gsp(image, image_ptr, sizeof(image), 0);
  /* Set source bitmap to image                                  */
  set_srcbm(image_ptr, PITCH, W, H, IMAGEDEPTH);
  /* Blit image to screen                                        */
  bitblt(W, H, 0, 0, 10, 10);
  /* Set source bitmap to screen                                 */
  set_srcbm(0, 0, 0, 0, 0);
  get_config(&c);
  rowbuf = gsp_malloc((c.mode.disp_hres *
                    c.mode.disp_psize) / 8);
  zoom_rect(W, H, 10, 10, ZOOM*W, ZOOM*H, 20+W,
            10, rowbuf);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define MAXSIZE 4096  /* max picture size in bytes          */

main()
{
  short w, h, x, y, n;
  char *s;
  PTR image;

  init_tiga(1);
  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);

  image = gsp_malloc(MAXSIZE);
  /* Compress image                                                   */
  encode_rect(w, h, x, y, image, MAXSIZE, 0);

  /* Now decompress the image several times                           */
  for (n = x ; n <= x + w; n += 16)
    decode_rect(n, n, image);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 via the three installed fonts.

Note that the function *loadinst_font* is called to load and install a TIGA font from a font file. The *loadinst_font* is not itself a TIGA function but does make calls to various TIGA functions to load a font. Refer to the *install_font* function description on page 5-41 for a complete source listing of the *loadinst_font* function.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h> /* defines FONTINFO structures        */

#define NFONTS 3      /* number of fonts installed          */
#define NLINES 2      /* number of lines of text per font   */

main()
{
  FONTINFO fontinfo;
  short index[NFONTS];
  short i, j, x, y;
```

```
    init_tiga(1);
    clear_screen(0);
    index[0] = loadinst_font("ti_rom11.fnt");
    /* install block font                                    */
    index[1] = loadinst_font("sys16.fnt");
    index[2] = loadinst_font("ti_rom16.fnt");
    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] = loadinst_font("ti_rom14.fnt");
    }
    term_tiga();
}
```

Syntax

#include <tiga.h>
#include <extend.h>

```
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).

```
#include <tiga.h>
#include <extend.h>

main()
{
  short x1, y1, x2, y2;

  init_tiga(1);
  clear_screen(0);
  x1 = 10;
  y1 = 20;
  x2 = 120;
  y2 = 80;
  draw_line(x1, y1, x2, y2);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y;

  init_tiga(1);
  clear_screen(0);
  w = 130;
  h = 90;
  x = 10;
  y = 10;
  draw_oval(w, h, x, y);
  draw_rect(w, h, x, y);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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.

```
#include <tiga.h>
#include <extend.h>

#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;
```

```
  init_tiga(1);
  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      */
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 screen. Each point on the circle is separated from its two neighbors by angular increments of approximately 1/8 radian.

```
#include <tiga.h>
#include <extend.h>

#define TWOPI 411775L  /* fixed–point 2*PI                    */
#define HALF 32768L    /* fixed–point 1/2                     */
#define RADIUS 60L     /* radius of circle                    */
#define N 3            /* angular increm. = 1/2**N radians    */

main()
{
  short x, y;
  long i;
  long u, v, xc, yc;

  init_tiga(1);
  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;
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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.

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.

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config* function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as:

$$\text{max_verts} = \frac{\text{comm_buff_size(bytes)} - 10}{4}$$

An alternate entry point, *draw_polyline_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although *draw_polyline_a* frees the application from having to check the data size, it takes longer to execute than its non-checking counterpart *draw_polyline*.

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).

```
#include <tiga.h>
#include <extend.h>

#define NVERTS 4 /* numbers of vertices                              */

typedef struct { short x, y; } POINT;

static POINT xy[NVERTS] =
{
  { 0, 0 }, { 60, 70 }, { 120, 10 }, { 120, 80 }
};

main()
{
  init_tiga(1);
  clear_screen(0);
  draw_polyline(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  init_tiga(1);
  clear_screen(0);
  draw_rect(130, 90, 10, 10);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

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 a buffer in TMS340 memory in which to save the compressed image.
- 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 rect. */
  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 that are 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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define MAXSIZE 4096       /* max picture size in bytes      */

main()
{
  short w, h, x, y, n;
  char *s;
  PTR picture;

  init_tiga(1);
  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     */
  picture = gsp_malloc(MAXSIZE);
  n = encode_rect(w, h, x, y, picture, MAXSIZE, 0);
  if (n > MAXSIZE)
  {
    text_out(x, y+h+20, ”Image buffer too small!”);
    term_tiga();
  }

  /* Now decompress the image                                */
  decode_rect(x+w, y+h, picture);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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.

To support 3-D 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 colinear, 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 colinear. If all the vertices are colinear, the polygon is invisible.

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config*

function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as

$$max_verts = \frac{comm_buff_size(bytes) - 10}{4}$$

An alternate entry point, *fill_convex_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although this alternate entry point frees the application from having to check the data size, it takes longer to execute than its nonchecking counterpart.

Example

Use the *fill_convex* function to fill a triangle. The three vertices are at coordinates (10, 10), (130, 10), and (70, 90).

```
#include <tiga.h>
#include <extend.h>

#define NVERTS 3      /* number of vertices                    */

typedef struct {short x, y;} POINT;

static POINT xy[NVERTS] =
{
  {10, 10}, {130, 10}, {70, 90}
};

main()
{
  init_tiga(1);
  clear_screen(0);
  fill_convex(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  init_tiga(1);
  clear_screen(0);
  fill_oval(130, 90, 10, 10);
  draw_rect(130+3, 90+3, 10–2, 10–2);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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.

```
#include <tiga.h>
#include <extend.h>
```

```
main()
{
  short w, h, x, y;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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).

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config* function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as

$$max_verts = \frac{comm_buff_size(bytes) - 10}{4}$$

An alternate entry point, *fill_polygon_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although *fill_polygon_a* frees the application from having to check the data size, it takes longer to execute than its nonchecking counterpart *fill_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.

```
#include <tiga.h>
#include <extend.h>

#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()
{
  init_tiga(1);
  clear_screen(0);
  fill_polygon(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  init_tiga(1);
  clear_screen(0);
  fill_rect(130, 90, 10, 10);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y, dx, dy;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 a 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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y, dx, dy;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

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 ENVIRONMENT structure contains the following fields:

```
typedef struct
{
  unsigned long xyorigin;
  unsigned long pensize;
  PTR srcbm;
  PTR dstbm;
  unsigned long stylemask;
}ENVIRONMENT;
```

Refer to the ENVIRONMENT structure description in Appendix A for detailed descriptions of each field.

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 TIGA 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. Use the *text_out* function to print the parameter values on the screen.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>  /* define ENVIRONMENT and FONTINFO    */

main()
{
  ENVIRONMENT env;
  FONTINFO fontinfo;
  char s[80];
  short h, x, y;

  init_tiga(1);
  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:");

    sprintf(s, "x origin = %d", (short) env.xyorigin);
    text_out(x, y += h, s);

    sprintf(s, "y origin = %d", env.xyorigin >> 16);
    text_out(x, y += h, s);

    sprintf (s, "pen width = %d", (short)env.pensize);
    text_out(x, y += h, s);

    sprintf(s, "pen height = %d", env.pensize >> 16);
    text_out(x, y += h, s);

    sprntf(s, "source bitmap = %lx", env.srcbm);
    text_out(x, y += h, s);

    sprintf(s, "destination bitmap = %lx", env.dstbm);
    text_out(x, y += h, s);

    sprintf(s, "line-style pattern = %lx", env.stylemask);
    text_out(x, y += h, s);
    term_tiga();
}
```

Syntax

#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h> /* defines FONTINFO structure         */

main()
{
  FONTINFO fontinfo;
  short xs, ys, xd, yd, w, h;
  long val;
  char *s;

  init_tiga(1);
  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);
    }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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. Use the *text_out* function to print the attribute values on the screen.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>      /* define FONTINFO structure      */
```

```
main()
{
  ENVIRONMENT env;
  FONTINFO fontinfo;
  char s[80];
  short val[3];
  short h, x, y;

  init_tiga(1);
  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:");

  sprintf(s, "text alignment = %d", val[0]);
  text_out(x, y += h, s);

  sprintf(s, "extra inter char spacing = %d", val[1]);
  text_out(x, y += h, s);

  sprintf(s, "inter char gaps = %d", val[2]);
  text_out(x, y += h, s);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short n;
  unsigned char v, s[80];

  init_tiga(1);
  clear_screen(-1);
  if (v = in_font(' ', '~'))
  {
    sprintf(s, "ASCII character code %d is undefined.", v);
    text_out(10, 10, s);
  }
  else
    text_out(10, 10, "Characters ' ' to '~' are defined.");
  term_tiga();
}
```

Syntax

#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

short install_font(pfont)
    PTR pfont;   /* font structure pointer in TMS340 mem. */

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 in TMS340 memory. (The FONT structure is described in Chapter 7.) 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 number of fonts that can be installed is limited only by the amount of available RAM on the TMS340 board.

Example

The following example illustrates a function to load and install a TIGA font from a TIGA font file. The function, *loadinst_font*, reads the font information from the font file, downloads it into TMS340 memory, and then installs the font and returns the font identifier. This function is extremely useful as a general TIGA font loader for any TIGA application.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>
#include <stdio.h>
#include <malloc.h>

#define FONT_MAGIC 0x8040
typedef struct
{
  ushort magic;
  long   size;
} FILEHDR;

/*LOADINST_FONT() Load, install font and return ref. ID      */
short loadinst_font(name)
char *name;
{
  FILE *fp;
  FILEHDR fh;
  FONT *hpTmp;
  short  id = 0;
  PTR  gpTmp = 0L;
```

```
  /* Examine font hdr. magic num. If incorrect return 0.    */
  if (!(fp = fopen( name, "rb")))
    return (0);
  fread( &fh, sizeof(FILEHDR), 1, fp);
  if (fh.magic != FONT_MAGIC)
  {
    fclose(fp);
    return (0);
  }
  /* Malloc font in host and target. Read font into host,   */
  /* then move to target and free host memory.   */
  if (hpTmp = (FONT*)malloc((ushort)fh.size))
    if (gpTmp = (PTR)gsp_malloc( fh.size))
    {
      rewind(fp);
      fread( hpTmp, fh.size, 1, fp);
      host2gsp (hpTmp, gpTmp, fh.size, 0);
      free( hpTmp);
    }
  /*  If all is OK, then install the font.                  */
  if (gpTmp)
    id = install_font(gpTmp);
  fclose(fp);
  return (id);
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h> /* define FONTINFO structure          */

main()
{
  FONTINFO fontinfo;
  short xs, ys, xd, yd, w, h;
  char *s;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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.

To support 3-D 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 colinear, the back-face test is made again using the next three vertices, *n*–1, 0, and 1. The test repeats until three vertices are found that are not colinear. If all the vertices are colinear, the polygon is invisible.

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config*

function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as

$$max_verts = \frac{comm_buff_size(bytes) - 10}{4}$$

An alternate entry point, *patnfill_convex_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although *patnfill_convex_a* frees the application from having to check the data size, it takes longer to execute than its non-checking counterpart *patnfill_convex*.

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).

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

typedef struct {short x, y;} POINT;

#define  NVERTS 4  /* num. of vertices in quadrilateral      */

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)0};
static POINT xy[NVERTS] =
{
  {96, 16}, {176, 72}, {96, 128}, {16, 72}
};

main()
{
  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(snowflake));
  host2gsp(snowflake, fillpatn.data, sizeof(snowflake), 0);
  set_patn(&fillpatn);
  patnfill_convex(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

/* brick pattern                                                    */
static short patnbits[] =
{
  0xFFFF, 0xD555, 0x8000, 0xC001, 0x8000, 0xC001, 0x8000, 0xD555,
  0xFFFF, 0x55D5, 0x0080, 0x01C0, 0x0080, 0x01C0, 0x0080, 0x55D5
};

static PATTERN current_patn = {16, 16, 1, (PTR)0};

main()
{
  init_tiga(1);
  clear_screen(0);
  current_patn.data = gsp_malloc(sizeof(patnbits));
  host2gsp(patnbits, current_patn.data, sizeof(patnbits), 0);
  set_patn(&current_patn);
  patnfill_oval(144, 96, 16, 16);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define  W 30         /* 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           */
```

```
/* Two contrasting area-fill patterns                          */
static short patnbits[] =
{
   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 };

   init_tiga(1);
   clear_screen(0);
   piepatn.data = gsp_malloc(sizeof(patnbits));
   host2gsp(patnbits, piepatn.data, sizeof(patnbits), 0);
   set_patn(&piepatn);
   patnfill_piearc(W, H, X, Y, 30, 160-30);   /* slice #1     */
   piepatn.data += (16 * 16);
   set_patn(&piepatn);
   patnfill_piearc(W, H, X, Y, 160, 230-160);/* slice #2     */
   piepatn.data -= (16 * 16);
   set_patn((PTR)&piepatn);
   patnfill_piearc(W, H, X, Y, 230, 320-230);/* slice #3     */
   piepatn.data += (16 * 16);
   set_patn(&piepatn);
   patnfill_piearc(W, H, X+20, Y, 320, 390-320);/* slice #4  */
   term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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).

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config* function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as:

$$\text{max_verts} = \frac{\text{comm_buff_size(bytes)} - 10}{4}$$

An alternate entry point, *patnfill_polygon_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical

to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although *patnfill_polygon_a* frees the application from having to check the data size, it takes longer to execute than its non-checking counterpart *patnfill_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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

typedef struct { short x, y; } POINT;

#define  NVERTS    14      /* 14 vertices in polygon         */

/* squares pattern                                           */
static short patnbits[16] =
{
  0x00FF, 0x0081, 0x1881, 0x3C81, 0x3C81, 0x1881, 0x0081, 0x00FF,
  0xFF00, 0x8100, 0x8118, 0x813C, 0x813C, 0x8118, 0x8100, 0xFF00
};

static PATTERN current_patn = {16, 16, 1, (PTR)0};

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()
{
  init_tiga(1);
  clear_screen(0);
  current_patn.data = gsp_malloc(sizeof(patnbits));
  host2gsp(patnbits, current_patn.data, sizeof(patnbits), 0);
  set_patn(&current_patn);
  patnfill_polygon(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

static short 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)0 };

main()
{
   init_tiga(1);
   clear_screen(0);
   current_patn.data = gsp_malloc(sizeof(patnbits));
   host2gsp(patnbits, current_patn.data, sizeof(patnbits), 0);
   set_patn(&current_patn);
   patnfill_rect(144, 96, 16, 16);
   term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 130 in width and 90 in height.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

static short fillpatn[] =
{
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111
};

static PATTERN framepatn = {16, 16, 1, (PTR)0};

main()
{
  short w, h, x, y, dx, dy;
```

```
    init_tiga(1);
    clear_screen(0);
    w = 130;
    h = 90;
    x = 10;
    y = 10;
    dx = w/4;
    dy = h/4;
    framepatn.data = gsp_malloc(sizeof(fillpatn));
    host2gsp(fillpatn, framepatn.data, sizeof(fillpatn), 0)
    set_patn(&framepatn);
    fill_rect(w, h, x, y);
    patnframe_oval(w, h, x, y, dx, dy);
    term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

static short fillpatn[] =
{
  0x0000, 0x0000, 0x7C7C, 0x4444, 0x4444, 0x4444, 0x7FFC, 0x0440,
  0x0440, 0x0440, 0x7FFC, 0x4444, 0x4444, 0x4444, 0x7C7C, 0x0000
};

static PATTERN framepatn = {16, 16, 1, (PTR)0};
main()
{
  short w, h, x, y, dx, dy;
```

```
  init_tiga(1);
  clear_screen(0);
  w = 144;
  h = 96;
  x = 16;
  y = 16;
  dx = 32;
  dy = 16;
  framepatn.data = gsp_malloc(sizeof(fillpatn));
  host2gsp (fillptn, framepatn.data, sizeof(fillpatn), 0);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h> /* define PATTERN structure           */

static short spiral[] =
{
  0x0000, 0x3FFC, 0x7FFE, 0x0006, 0x0006, 0x1FC6, 0x3FE6, 0x3066,
  0x3066, 0x33E6, 0x31C6, 0x3006, 0x3006, 0x3FFE, 0x1FFC, 0x0000
};

static PATTERN fillpatn = {16, 16, 1, (PTR)0};

main()
{
  init_tiga(1);
  clear_screen(0);
  set_pensize(24, 16);
  fillpatn.data = gsp_malloc(sizeof(spiral));
  host2gsp(spiral, fillpatn.data, sizeof(spiral), 0);
  set_patn(&fillpatn);
  patnpen_line(16, 16, 144, 112);
  patnpen_line(144, 112, 144, 16);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

static short stripes[16] =
{
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111
};

static PATTERN fillpatn = { 16, 16, 1, (PTR)0 };

main()
{
  short w, h, x, y;

  init_tiga(1);
  clear_screen(0);
  set_pensize(24, 16);
  fillpatn.data = gsp_malloc(sizeof(stripes));
  host2gsp(stripes, fillpatn.data, sizeof(stripes), 0);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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's 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 of [–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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

static short stripes[] =
{
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111
};

static PATTERN fillpatn = { 16, 16, 1, (PTR)0 };

main()
{
  short w, h, x, y;

  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(stripes));
  host2gsp(stripes, fillpatn.data, sizeof(stripes), 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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 relative to the drawing origin. 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 of a radian.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define FOURPI 823550L     /* fixed-point 4*PI               */
#define HALF 32768L        /* fixed-point 1/2                */
#define AMPL 60L           /* sine wave amplitude            */
#define N 4                /* increment = 1/2**N radians     */


static short stripes[] =
{
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111
};

static PATTERN fillpatn = { 16, 16, 1, (PTR)0 };

main()
{
  long i;
  short x, y;
  long u, v;

  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(stripes));
  host2gsp(stripes, fillpatn.data, sizeof(stripes), 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;
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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.

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config* function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as

$$\text{max_verts} = \frac{\text{comm_buff_size(bytes)} - 10}{4}$$

An alternate entry point, *patnpen_polyline_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although this alternate entry point frees the application from having to check the data size, it takes longer to execute than its nonchecking counterpart.

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).

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

typedef struct {short x, y;} POINT;

#define NVERTS 4      /* number of vertices in polyline     */

static short amoeba[16] =
{
  0x1008, 0x0C30, 0x03C0, 0x8001, 0x4002, 0x4002, 0x2004, 0x2004,
  0x2004, 0x2004, 0x4002, 0x4002, 0x8001, 0x03C0, 0x0C30, 0x1008
};

static PATTERN fillpatn = {16, 16, 1, (PTR)0};

static POINT xy[NVERTS] =
{
  {16, 16}, {64, 128}, {128, 48}, {160, 48}
};

main()
{
  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(amoeba));
  host2gsp(amoeba, fillpatn.data, sizeof(amoeba), 0);
  set_patn(&fillpatn);
  set_pensize(24, 32);
  patnpen_polyline(NVERTS, xy);    /* fat polyline          */
  set_pensize(2, 2);
  pen_polyline(NVERTS, xy);        /* thin polyline         */
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

void pen_line(x1, y1, x2, y2)
        short x1, y1;          /* start coordinates          */
        short x2, y2;          /* end coordinates            */
```

Description

The *pen_line* function draws 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. All coordinates are specified relative to the drawing origin.

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short x1, y1, x2, y2, r;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 from [–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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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's 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 of [–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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 relative to the drawing origin. 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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short w, h, x, y;

  init_tiga(1);
  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);
  }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
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.

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.

Note that large amounts of data passed to this function may overflow the TIGA command buffer. The size of the command buffer is contained in the *comm_buff_size* field of the CONFIG structure returned by the *get_config* function. The application must ensure that data passed to this function will not overflow the command buffer.

The number of vertices that may be sent without overflowing the command buffer is calculated as:

$$\text{max_verts} = \frac{\text{comm_buff_size(bytes)} - 10}{4}$$

An alternate entry point, *pen_polyline_a*, is provided to automatically check the size of the data being passed. The arguments for this function are identical to those described above. If the command buffer is too small to contain the function's data, this entry point will attempt to allocate a temporary buffer in TMS340 memory. If there is not enough memory available for this buffer, the standard TIGA error function is invoked (which can be trapped by using the *install_usererror* function). Although *pen_polyline_a* frees the application from having to check the data size, it takes longer to execute than its nonchecking counterpart *pen_polyline*.

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).

```
#include <tiga.h>
#include <extend.h>

typedef struct {short x, y;} POINT;

#define NVERTS 4       /* number of vertices in polyline     */

static POINT xy[NVERTS] =
{
  {10, 10}, {64, 96}, {100, 48}, {140, 48}
};

main()
{
  init_tiga(1);
  clear_screen(0);
  set_pensize(5, 4);
  pen_polyline(NVERTS, xy);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>      /* define FONTINFO structure     */

main()
{
  FONTINFO fontinfo;
  short xs, ys, xd, yd, w, h;
  unsigned long val;
  char *s;

  init_tiga(1);
  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);
    }
  term_tiga();
 }
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

void seed_fill(x, y, buf, maxbytes)
    short x, y;          /* seed pixel coordinates      */
    PTR 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 in TMS340 memory 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 is 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 before 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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define MAXBYTES 2048

main()
{
  PTR buf;                /* seed-fill temp. buffer        */

  init_tiga(1);
  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                 */
  buf = gsp_malloc(MAXBYTES);
  seed_fill(20, 20, buf, MAXBYTES);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

void seed_patnfill(x, y, buf, maxbytes)
        short x, y;             /* seed pixel coordinates       */
        PTR 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 in TMS340 memory 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 the 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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define MAXBYTES 2048      /* size of temp buffer in bytes   */

static short snowflake[] =
{
  0x0000, 0x01C0, 0x19CC, 0x188C, 0x0490, 0x02A0, 0x31C6, 0x3FFE,
  0x31C6, 0x02A0, 0x0490, 0x188C, 0x19CC, 0x01C0, 0x0000, 0x0000
};

static PATTERN fillpatn = {16, 16, 1, (PTR)0};

main()
{
  short w, h, x, y, n;
  PTR buf;

  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(snowflake));
  host2gsp(snowflake, fillpatn.data, sizeof(snowflake), 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);
  buf = gsp_malloc(MAXBYTES);
  seed_patnfill(20, 20, buf, MAXBYTES);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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

See the *delete_font* function description example on page 5-9.

Syntax

```
#include <tiga.h>
#include <extend.h>

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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  short x, y, w;
  char *s;

  init_tiga(1);
  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);
    }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

void set_dstbm(baseaddr, pitch, xext, yext, psize)
        PTR baseaddr;           /* bitmap base address          */
        short pitch;            /* bitmap pitch                 */
        short xext, yext;       /* x and y extents              */
        short psize;            /* pixel size                   */
```

Description

The *set_dstbm* function sets the destination bitmap for subsequent drawing functions. Currently, only the *bitblt* function can write to a bitmap other than the screen. All other drawing functions abort (return without drawing anything) if the destination bitmap is set to a bitmap other than the screen.

Argument *baseaddr* is a pointer to the destination bitmap. Invoking the function with a *baseaddr* value of 0 sets the destination bitmap 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 bitmap; in other words, the destination bitmap is contained in an offscreen buffer. The specified bitmap should begin on an even pixel boundary in memory. For instance, when the pixel size is 32 bits, the 5 LSBs of the bitmap's base address should be 0s.

Argument *pitch* is the difference in bit addresses from the start of one row of the bitmap to the next. The bitblt function requires that the destination pitch be specified as a positive, nonzero multiple of the destination bitmap's pixel size. The bitblt function executes faster 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 bitmap. 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 bitmap, responsibility for clipping is left to the calling program.

Example

Use the *set_dstbm* function to designate an offscreen buffer as the destination bitmap. Contract an image from the screen to 1 bit per pixel and store the contracted image in the offscreen 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 *typedefs.h*, which defines the FONT and FONTINFO structures.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>      /* define FONTINFO                 */

#define MAXBYTES 4096      /* size of image buffer (bytes)   */

static FONTINFO fontinfo;

main()
{
  short w, h, x, y, pitch;
  char *s;
  PTR image;

  init_tiga(1);
  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!");
    term_tiga();
  }

  /* Capture text image from screen                          */
  image = gsp_malloc(MAXBYTES);
  set_dstbm(image, pitch, w, h, 1);/* offscreen bitmap      */
  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                 */
  term_tiga();
}
```

Syntax

#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

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 in host memory to a PATTERN structure.

The PATTERN structure contains the following fields:

```
typedef struct
{
  unsigned short width;
  unsigned short height;
  unsigned short depth;
  PTR data;
}PATTERN;
```

- 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 bitmap in TMS340 memory containing the actual pattern.

Refer to the PATTERN structure description in Appendix A for detailed descriptions of each field.

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 bitmap. A bit value of 1 in the pattern bitmap 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 bitmap.

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 bitmap containing the pattern is not saved by the function; this bitmap must be preserved by the calling program as long as the pattern remains in use.

During initialization of the drawing environment, 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 the pattern, call the *patnfill_rect* function to tile the screen with alternating heart and star patterns.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>      /* define PATTERN structure      */

static PATTERN fillpatn = {16, 16, 1, (PTR)0};

static short patnbits[] =
/* Heart pattern                                             */
{
0x0000, 0x0000, 0x0E38, 0x1F7C, 0x3FFE, 0x3FFE, 0x3FFE, 0x3FFE,
0x1FFC, 0x0FF8, 0x07F0, 0x03E0, 0x01C0, 0x0080, 0x0000, 0x0000,

/* Star pattern                                              */
0xFFFF, 0xFF7F, 0xFF7F, 0xFF7F, 0xFE3F, 0xFE3F, 0x8000, 0xE003,
0xF007, 0xFC1F, 0xFC1F, 0xF80F, 0xF9CF, 0xF3E7, 0xF7F7, 0xFFFF
};

main()
{
  short x, y, index;
  PTR patn_base_addr;

  init_tiga(1);
  clear_screen(0);
  index = 0;
  patn_base_addr = gsp_malloc(sizeof(patnbits));
  host2gsp(patnbits, patn_base_addr, sizeof(patnbits), 0);
  for (x = 16; x < 160; x += 32)
    for (y = 16; y < 96; y += 32)
    {
      fillpatn.data = patn_base_addr + (16 * 16 * (index ^=
1));
      set_patn(&fillpatn);
      patnfill_rect(32, 32, x, y);
    }
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 a 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.

```
#include <tiga.h>
#include <extend.h>

main()
{
  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

void set_srcbm(baseaddr, pitch, xext, yext, psize)
        PTR baseaddr;          /* bitmap base address        */
        short pitch;           /* bitmap pitch               */
        short xext, yext;      /* x and y extents            */
        short psize;           /* pixel size                 */
```

Description

The *set_srcbm* function sets the source bitmap in TMS340 memory for subsequent drawing functions. Currently, only the *bitblt* and *zoom_rect* functions can access a source bitmap other than the screen.

Argument *baseaddr* is a pointer to the source bitmap in TMS340 memory. Invoking the function with a *baseaddr* value of 0 designates the screen as the source bitmap. In this case, the last four arguments are ignored by the function. A nonzero *baseaddr* is interpreted as a pointer to a linear bitmap; that is, the source bitmap is contained in an offscreen buffer. The specified bitmap should begin on an even pixel boundary in memory. For instance, when the pixel size is 32 bits, the 5 LSBs of the bitmap's base address should all be 0s.

Argument *pitch* is the difference in bit addresses from the start of one row of the bitmap to the next. The bitblt function requires that the source pitch be specified as a positive, nonzero multiple of the source bitmap's pixel size. The bitblt function executes faster 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 bitmap. 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 bitmap, responsibility for clipping is left to the application program.

Example

Use the *set_srcbm* function to designate an offscreen buffer as the source bitmap. Expand the image from 1 bit per pixel to the screen pixel size and copy the image to the screen.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define W 23               /* width of image in pixels       */
#define H 9                /* height of image in pixels      */
#define PITCH 32           /* pitch of image in bits         */


static short image[] =
{
  0xFFFF, 0x007F, 0x0001, 0x0040, 0x45D5, 0x005C,
  0x4455, 0x0054, 0x44DD, 0x0054, 0x4455, 0x0054,
  0xDDD5, 0x005D, 0x0001, 0x0040, 0xFFFF, 0x007F
};

main()
{
  PTR buf, bmaddr;
  short x, y;
  CONFIG c;

  init_tiga(1);
  clear_screen(0);

  /* Expand image to screen                                  */
  x = y = 10;
  bmaddr = gsp_malloc(sizeof(image));
  host2gsp(image, bmaddr, sizeof(image), 0);
  set_srcbm(bmaddr, PITCH, W, H, 1);    /* offscreen bitmap */
  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                     */
  get_config(&c);
  buf = gsp_malloc((c.mode.disp_psize * W)/8);
  zoom_rect(W, H, x, y, 3*W, 3*H, x, y+2*H, buf);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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, the additional intercharacter spacing is 0, which is the default.

The intercharacter gaps attribute controls whether or not 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, the filling of intercharacter gaps is disabled, which is the default.

Example

Set the text alignment to the character base line position. This can be 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 shows two attributes set by a single call to *set_textattr*. It sets the text alignment mode to base line position by 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[2];
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);

Syntax

```
#include <tiga.h>
#include <extend.h>

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, 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.

```
#include <tiga.h>
#include <extend.h>

#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()
{
  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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).

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, 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_oval* function to render the outline of an ellipse with a 32-bit repeating line-style pattern.

```
#include <tiga.h>
#include <extend.h>

#define DOTDASH 0x18FF18FF /* dot-dash line-style pattern   */

main()
{
  init_tiga(1);
  clear_screen(0);
  styled_oval(130, 90, 10, 10, DOTDASH, 0);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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).

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, 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_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.

```
#include <tiga.h>
#include <extend.h>

#define DOTDOTDASH 0x3F333F33 /* ..–..– line-style pattern   */
#define NEW 0                /* mode = load new line style    */
#define OLD 2                /* mode = re-use old line style  */

main()
{
  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

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 of [–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).

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, 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_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.

```
#include tiga.h
#include <extend.h>

#define DOTDOTDASH 0x3F333F33  /* line-style pattern        */

main()
{
  init_tiga(1);
  clear_screen(0);
  styled_piearc(130, 90, 10, 10, -33, -270+33,
                      DOTDOTDASH, 0);
  term_tiga();
}
```

Syntax

#include <tiga.h>
#include <extend.h>

void swap_bm()

Description

The *swap_bm* function swaps the source and destination bitmaps. To move pixels back and forth between two bitmaps, 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 bitmaps. Initially, the destination bitmap is designated as an offscreen buffer, and the source bitmap 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 offscreen buffer by a call to the bitblt function. Following a call to *swap_bm*, the destination bitmap is the screen, and the source bitmap is the offscreen buffer. The captured image is copied to the screen three times by three calls to the bitblt function.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h> /* define FONT and FONTINFO          */
#define MAXBYTES 2048 /* size of image buffer in bytes      */

static FONTINFO fontinfo;

main()
{
  short w, h, x, y, pitch;
  char *s;
  PTR image;

  init_tiga(1);
  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!");
    term_tiga();
  }
  /* Capture text image from screen                         */
  image = gsp_malloc(MAXBYTES);
  set_dstbm(image, pitch, w, h, 1);/* offscreen bitmap      */
  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        */
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>

short text_width(s)
    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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.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;

  init_tiga(1);
  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);
  term_tiga();
}
```

Syntax

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

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      */
    PTR 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 offscreen 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 bitmaps; these bitmaps 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 bitmaps to be set to the default bitmap, which is the screen. The *zoom_rect* function requires that the pixel sizes for the source and destination bitmaps be the same. The destination bitmap 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 bitmap is the screen, the current drawing origin is used. If the source bitmap is an offscreen buffer, the origin lies at the bitmap'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 pointer to a buffer in TMS340 memory, which is 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. You are 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 bitmap 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 bitmap to a color pixel array contained in an offscreen buffer. Second, the *zoom_rect* function is called to zoom the expanded pixel array from the offscreen 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 before 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.

```
#include <tiga.h>
#include <extend.h>
#include <typedefs.h>

#define W 48                /* width of source rectangle      */
#define H 32                /* height of source rectangle     */
#define X 12                /* source rectangle left edge     */
#define Y 12                /* top edge of source rectangle   */
#define Z 3                 /* zoom factor                    */

static short tinyblobs[16] =
{
  0x1008, 0x0C30, 0x03C0, 0x8001, 0x4002, 0x4002, 0x2004, 0x2004,
  0x2004, 0x2004, 0x4002, 0x4002, 0x8001, 0x03C0, 0x0C30, 0x1008
};

static PATTERN fillpatn = {16, 16, 1, (PTR)0};

main()
{
  CONFIG c;
  PTR buf;

  init_tiga(1);
  clear_screen(0);
  fillpatn.data = gsp_malloc(sizeof(tinyblobs));
  host2gsp(tinyblobs, fillpatn.data, sizeof(tinyblobs), 0);
  set_patn(&fillpatn);
  patnfill_rect(W, H, X, Y);
  frame_rect(W, H, X, Y, 1, 1);
  get_config(&c);
  buf = gsp_malloc((c.mode.disp_psize * W)/8);
  zoom_rect(W, H, X, Y, Z*W, Z*H, X+W+10, Y, buf);
  term_tiga();
}
```

Chapter 6

Graphics Library Conventions

The TIGA Extended 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.

The graphics library follows a set of strict conventions 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 functions such as lines and arcs
- Defining the pixels covered by area-fill functions such as polygons and ellipses

The graphics library supports a variety of methods for combining source and destination pixel values during drawing operations. Pixels are combined according to how you configure 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.

This chapter discusses the following topics:

Section **Page**

6.1 Graphics Library Function Naming Conventions 6-2
6.2 Coordinate Systems 6-4
6.3 Area-Filling Conventions 6-6
6.4 Vector Drawing Conventions 6-8
6.5 Rectangular Drawing Pen 6-10
6.6 Area-Fill Patterns 6-12
6.7 Line-Style Patterns 6-13
6.8 Operations on Pixels 6-15
6.9 Clipping Window 6-18
6.10 Pixel-Size Independence 6-19

6.1 Graphics Library Function Naming Conventions

A set of conventions has been adopted for naming extended graphics library functions 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 6–1 is a list of the geometric types supported by the library. The left column specifies the function-name modifier corresponding to each type.

Table 6–1. 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 6–2 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 6–2. Rendering Styles

Function Name	Rendering Style
draw	Draws a pixel-thick line, arc, or outline with the current foreground color.
styled	Similar to draw except that the line, arc, or outline is drawn using 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 background 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 6–3 is a checklist indicating which combinations are supported.

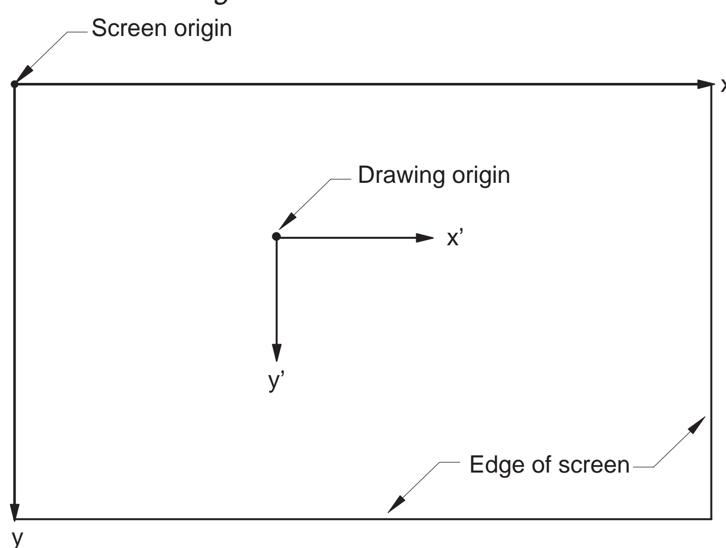
Table 6–3. Checklist of Available Geometric Types and Rendering Styles

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

6.2 Coordinate Systems

Figure 6–1 shows the conventions used by TIGA to map x-y coordinates onto the screen.

Figure 6–1. 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 6–1 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) are visible.

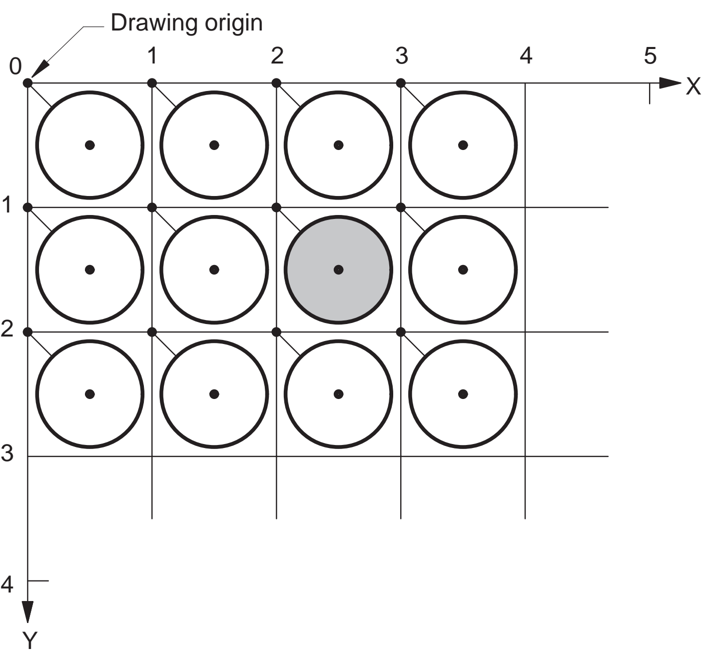
Figure 6–2 is a close-up of several pixels in the vicinity of the drawing origin that illustrates the relationship of the pixels on the screen to the coordinate grid lines. Each vertical or horizontal grid line corresponds to an integer x or y coordinate value. Centered within each square of the grid is a pixel, drawn as a circle.

The *draw* and *styled* functions within the library (refer to Table 6–2) 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 6–2.

Figure 6–2. Mapping of Pixels to Coordinate Grid



TIGA's extended graphics library represents x-y coordinates as16-bit signed integers. Valid coordinate values are limited to the range of –16384 to +16383. Restricting the values to this range provides one guard bit to protect against overflow during 16-bit arithmetic operations.

6.3 Area-Filling Conventions

Geometric objects can be rendered in a variety of styles. Filled functions such as polygons and ellipses can be filled with either a solid color or a two-dimensional area-fill pattern.

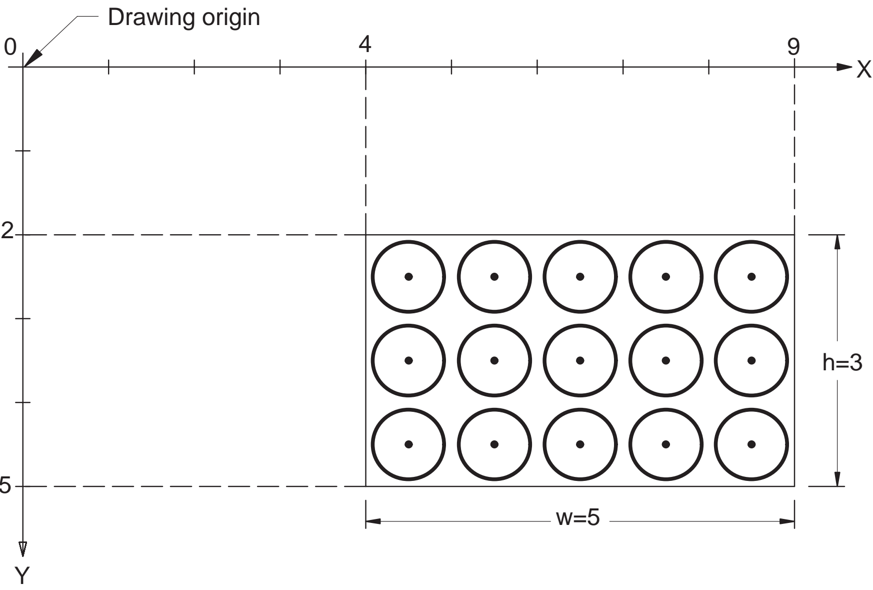
The fill, frame, and pen functions within the library designate a pixel as being part of a filled region if the center of the pixel falls within the boundary of that region.

Figure 6–3 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);

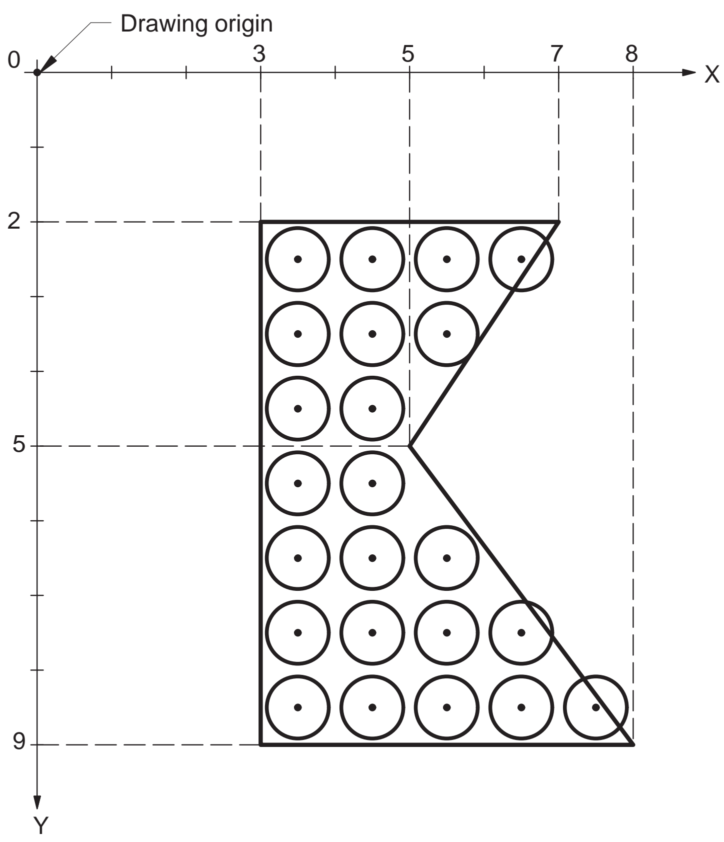
The pixels selected to fill the rectangle are indicated in the figure.

Figure 6–3. A Filled Rectangle



As a second example, a filled polygon is shown in Figure 6–4. 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 6–4. 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.

6.4 Vector-Drawing Conventions

Mathematically ideal points, lines, and arcs are defined to be infinitely thin. Because these figures contain no area, they would be invisible if drawn using the conventions described in Section 6.3 for filled areas. A different set of conventions must be used to make points, lines and arcs visible. These are referred to as vector-drawing conventions to distinguish them from the area-filling conventions discussed previously. Vector-drawing conventions apply to all library functions whose names include the modifiers draw or styled.

Vector functions that produce 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 under the pen is filled with either a solid color or an area-fill pattern.

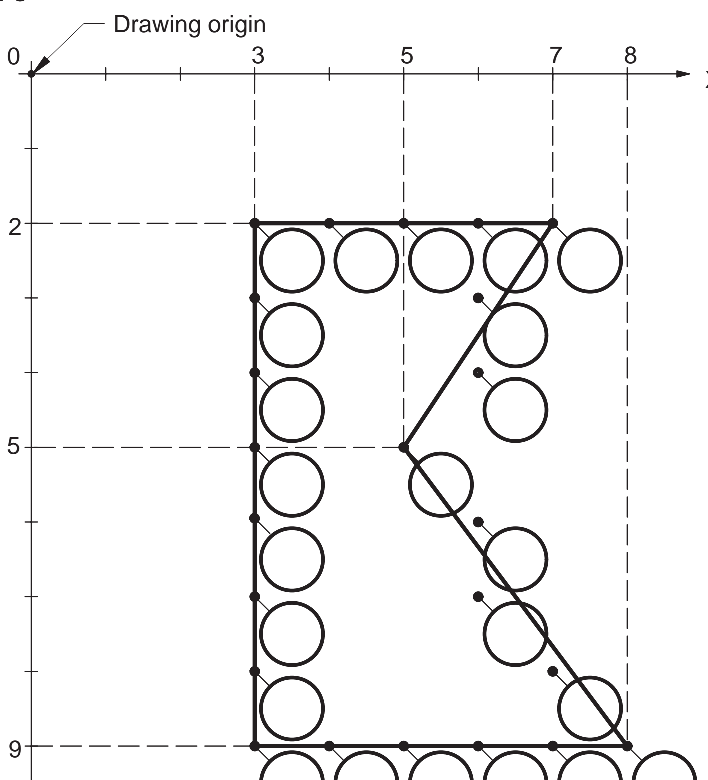
The vector-drawing conventions associate the point specified by the integer coordinate pair (x, y) with the pixel that lies just to the lower right of this point; that is, the pixel whose center lies at coordinates (x+1/2, y+1/2). For example, the function call

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

draws the pixel centered at (2.5, 1.5), as shown in Figure 6–2.

As a second example, the polygon from Figure 6–4 is shown again in Figure 6–5, 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 6–5. 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.

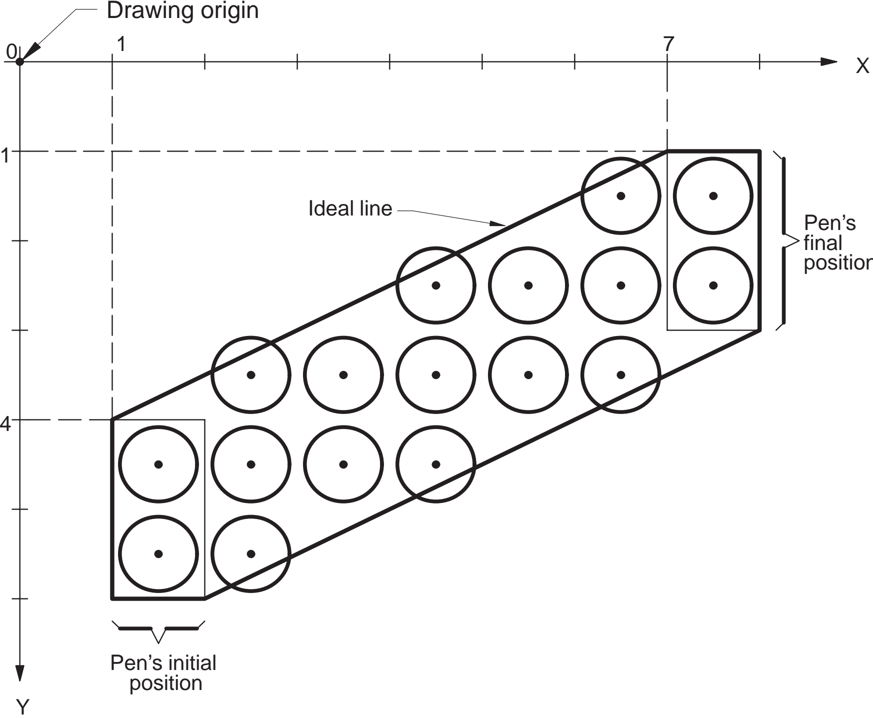
6.5 Rectangular Drawing Pen

The graphics functions that follow the vector-drawing conventions in Section 6.4 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. TIGA 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 6–6 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 in Section 6.3. The pixels interior to the line are indicated in the figure.

Figure 6–6. 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 in Section 6.4. 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.

For instance, consider an ellipse defined by 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 with the same arguments by calling *draw_oval*, which is a vector-drawing function, 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.

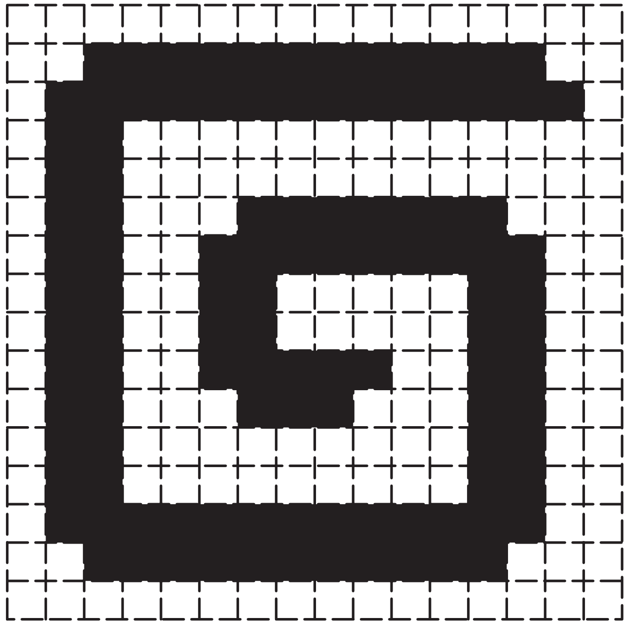
6.6 Area-Fill Patterns

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 of the screen is filled with a 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 6–7, 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 6–7 correspond to1s in the source bit map, and white squares correspond 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 those corresponding to 0s by the background color.

Figure 6–7. A 16-by-16 Area-Fill Pattern



6.7 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 used 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.

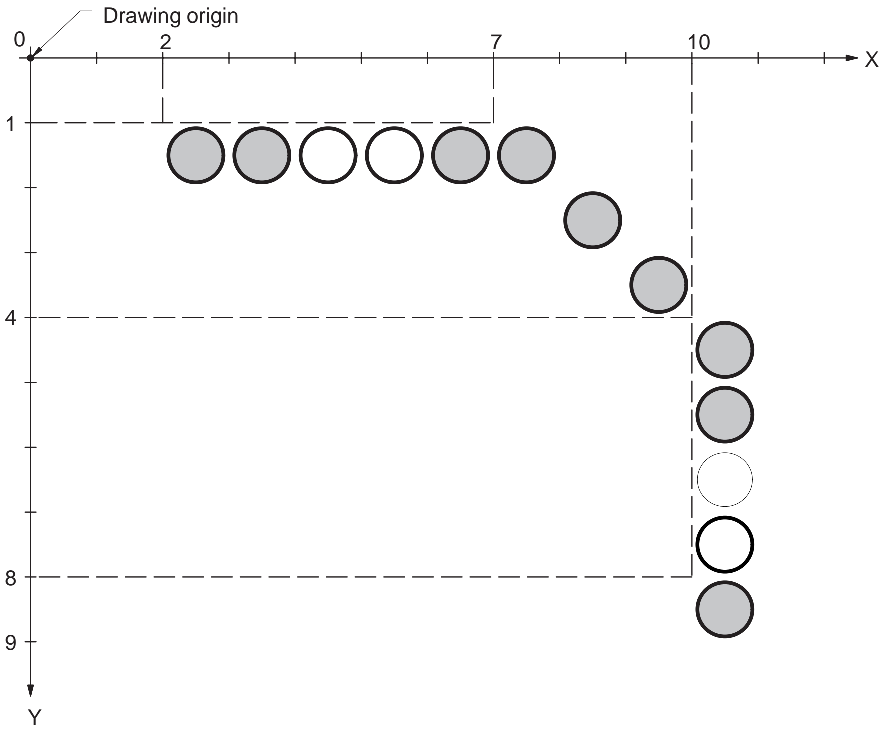
When a line-style pattern function such as *styled_line* is called, either a new pattern mask is specified, or an old one is reused. The latter option supports the drawing of continuous patterns across a series of connecting lines. After a *styled_line* has been used to draw a line *n* pixels in length, 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 6–8. Darkened pixels correspond to 1s in the line-style mask, and white pixels correspond to 0s. The lines in the Figure 6–8 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 6–8. Three Connected Styled Lines



6.8 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 replaced (the source pixel value simply replaces the destination pixel).

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

6.8.1 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 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 is a transparent pixel, regardless of the destination pixel value.

6.8.2 Plane Mask

The plane mask specifies which bits within a pixel are protected from modification, and it 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.

6.8.3 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 6–4 and Table 6–5. 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 6–4. 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 6–5. 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.

6.9 Clipping Window

The graphics output produced by the TIGA drawing functions is always confined to the interior of a rectangular clipping window that occupies all or a portion of the screen. All TIGA 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.

6.10 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-based display hardware system, however, may support only a subset of the pixel sizes that the TMS340 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, 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 chips. Changing the pixel size in software is not much more difficult than loading the TMS340 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 accomplished, 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 half-toning or ordered-dithering algorithms to simulate a larger palette of colors. The application may be unable 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 7

Bit-Mapped Text

The TIGA Interface supports the display of text in a variety of styles and fonts. 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.

Topics in this chapter include

Section		Page
7.1	Bit-Mapped Font Parameters	7-2
7.2	Font Data Structure	7-5
7.3	Proportionally Spaced Versus Block Fonts	7-11
7.4	Font Table	7-12
7.5	Text Attributes	7-13
7.6	Available Fonts	7-14

7.1 Bit-Mapped Font Parameters

Table 7–1 lists the text-related functions available in both the core and extended graphics libraries. Refer to the individual descriptions of these functions in Chapters 4 and 5 for details.

Table 7–1. Text-Related Functions

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

A font is a complete assortment of characters of a particular size and style (or typeface). TIGA 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 current foreground color, and 0s to the background color.

Figure 7–1 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 7–1, 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 7–1, 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 7–1, 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 text alignment attribute. In the default state, the origin lies at the top left corner of the character. Alternately, as shown in Figure 7–1, 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 baseline origin is useful when multiple fonts are mixed in a single row 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 7–1, 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 7–1, 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 bitmap for the character shape. The sides of the rectangle are defined by the image width and the character height, as defined below. For example, in Figure 7–1, the character rectangle for the letter “A” is 16 pixels wide by 27 pixels high.

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

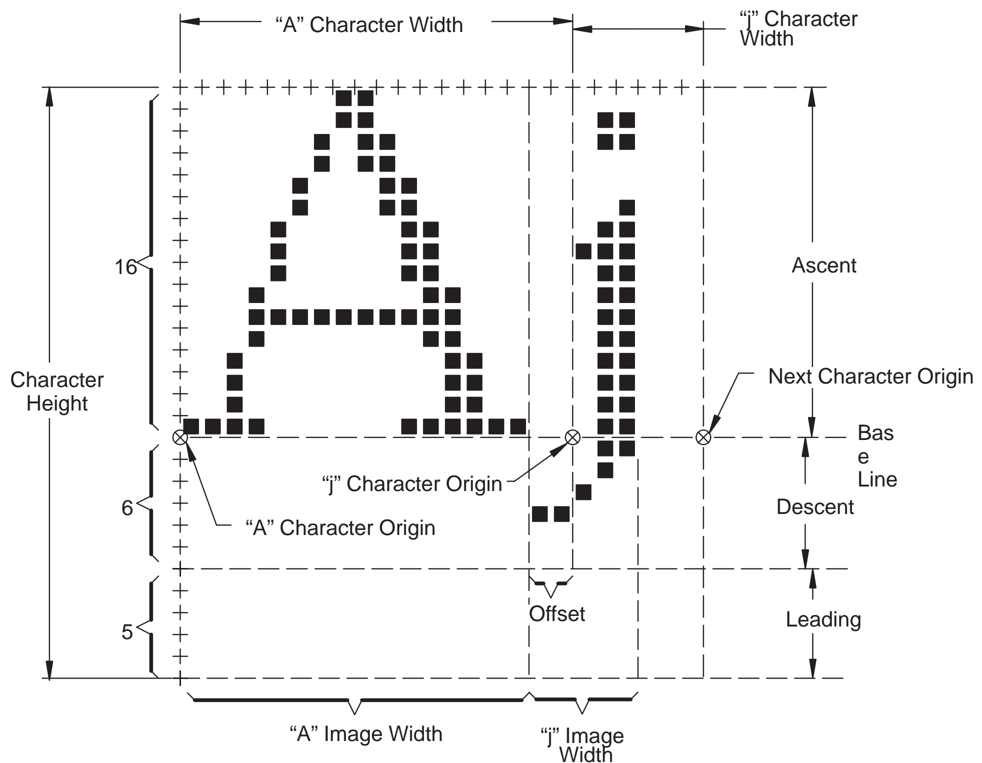
Character Offset

The *character offset* is the horizontal displacement from the character origin to the left edge of character image. If the offset is negative, the character image extends to the left of the character origin. For example, in Figure 7–1, 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 character 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 7–1, the image widths of the characters “A” and “j” are 16 and 5, respectively.

Figure 7–1. Bit-Mapped Font Parameters



7.2 Font Data Structure

The data structure for TIGA bit-mapped fonts is shown in Figure 7–2. 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 7–2. Data Structure for Bit-Mapped Fonts

Header
Pattern Table
Location Table
Offset/width Table

7.2.1 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 default;         /* 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 programs 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) default

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 *default* 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 bottom of one line of text to top of 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).

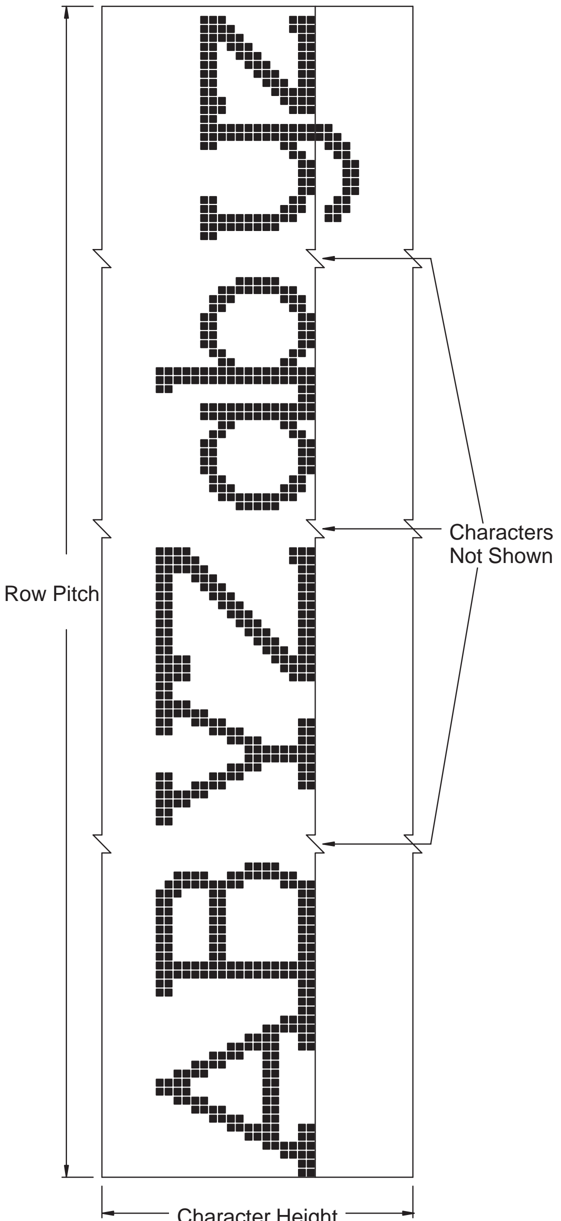
18) oOwTbl

    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).

7.2.2 Font Pattern Table

The font pattern table is a two-dimensional bit map organized as shown in Figure 7–3. 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 7–3. Bit-Mapped Font Representation



7.2.3 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 (top left corner of the bit map in Figure 7–3) 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 7–3.

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.

7.2.4 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.

7.3 Proportionally Spaced Versus Block Fonts

Two varieties of TIGA fonts are distinguished by the value of the *charwide* field in the 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.

7.4 Font Table

The system font, permanently installed in TIGA'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 TIGA as they are installed, and the fonts are thereafter identified by these indices during text operations.

The maximum number of fonts that may be installed is limited only by the amount of memory available on the TMS340-based board.

7.5 Text Attributes

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

1) *Text Alignment*

   Specifies whether the character origin (see previous definition) for each character is located at the base line or at the top edge of the character. The default is the top edge.

2) *Additional Intercharacter Spacing*

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

3) *Intercharacter Gaps*

   Specifies if 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. 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.

7.6 Available Fonts

The TIGA Interface includes a bit-mapped font database consisting of 19 typefaces available in a variety of sizes. The size of a font is specified in terms of its character height in pixels. The available fonts are summarized in Table 7–2. All TIGA-compatible fonts have a filename extension of *.fnt* and are located in the *\tiga\fonts* directory.

Several of the fonts in Table 7–2 are labeled as *monospaced* (type M in the rightmost column) rather than *proportionally spaced*. A monospaced font has uniform character width across the font. The monospaced fonts in Table 7–2 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 7–2. 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 Roman		11		14	16	18	20	22	26	30	33	38		52	78	P
TI Typewriter Elite		11		14	16	18	20	22	26			38				M
Point. size equivalent at 640 × 480 pixels	05	09	10	12	14	16	18	20	24	28	32	36	40	48	72	

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

7.6.1 Installing Fonts

The application program must load the font(s) used by the application. Each font is referred to by a filename that uniquely identifies it. Specify this filename when you load the desired font. The filenames of the available fonts are presented in Table 7–3.

Table 7–3. Installable Font Names

Font Name	Font Filename (all have *.fnt* extension)
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 in the bottom entry of Table 7–2 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 7–2.

See the *install_font* function description in Chapter 5 for an example of how to load a TIGA font.

7.6.2 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.

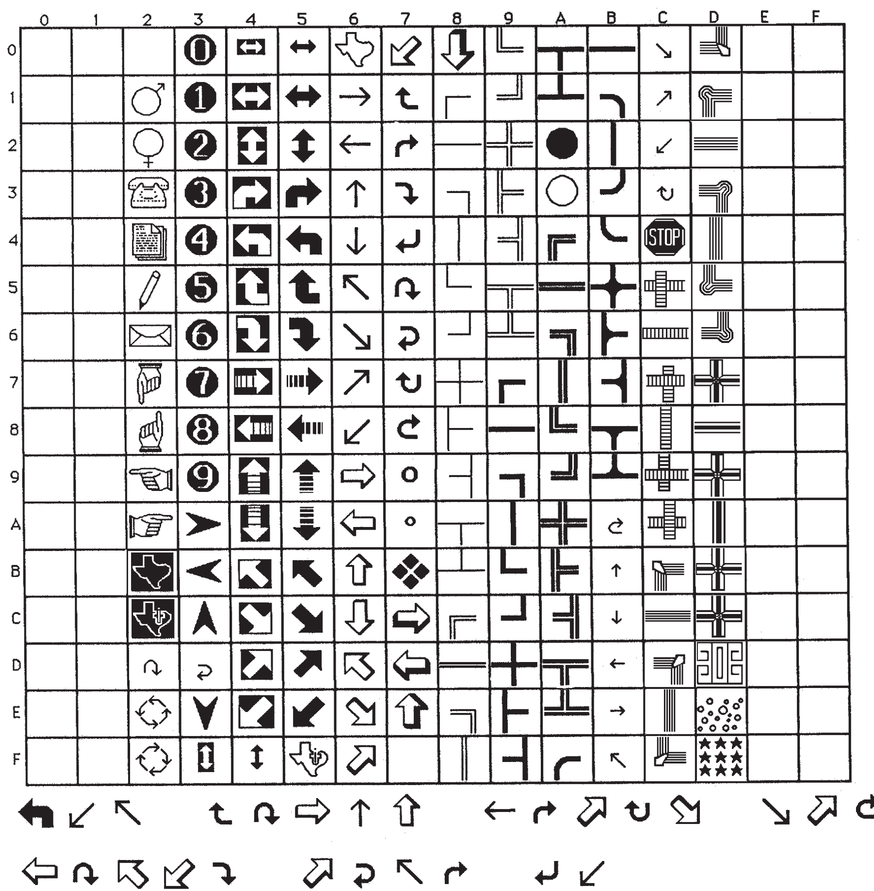
Spacing Monospace

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



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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 fo
typography takes. The selection of

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

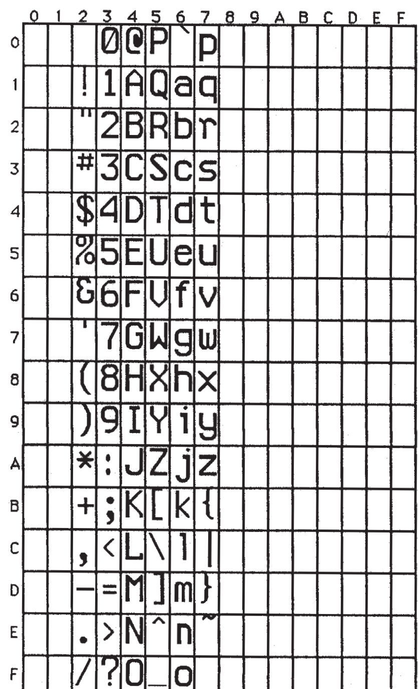
Spacing Monospace

Derivation Original character set, no typesetter's equivalent

Description Designed as a terminal display font. The 16-pixel size renders a standard 80-column display at 640 × 480 resolution. The 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 typograp
takes. The selection of a congruous typeface,
quality and suitability for

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

Taste in printing determi
the form typography takes
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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†			–		
1			!	1	A	Q	a	q			°			—		
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			•			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß			◊		
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©		…			
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			´					
C			,	<	L	\	l	|			¨					
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r			¢					
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©					
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©					
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 for

Spacing Proportional

Derivation Origin 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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 mediumweight, depending on size.

Sizes 16, 19, 24, 32, 44, and 64 pixels

Example

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	|	ℓ	`	ρ				5	/		7	
1			√	1	∇	Γ	α	γ			8		|		9	
2			"	2	∞	Θ	β	θ			4	'		(	0	≈
3			{	3	Ψ	Σ	ψ	σ			3	·		(		
4			}	4	Φ	→	φ	τ			6			)		
5			±	5	←	Ξ	ε	ξ			8			)		
6			/	6	<	∝	ɩ	x			7			/		
7			'	7	Λ	Δ	λ	δ								
8			⌉	8	¶	≡	η	χ								,
9			∫	9	↑	Υ	ι	υ								.
A			⌠	:	>	≃	J	ζ			2			1		
B			∕	;	§	π	κ	Π				9		2		
C			,	<	Ω	\	ω	|				0		3		
D			=	∠	∂	]	μ							4		
E			.	>	∼	\	ν	~				·		5		
F			/	?	↓	_	o							6		

→αστε ιν ρθιντινλ φετεθμινεσ τηε ιοθμ τυρολθα
τακεσ. →ηε σεωεψτιον οι α ψονλθεξοξσ τυρειαψε, ´
γξαιωιτυ ανφ σξιταβιωιτυ

→αστε ιν ρθιντινλ φετεθμινεσ τηε ιοθμ
τυρολθαρηυ τακεσ. →ηε σεωεψτιον οι α ψον
τυρειαψε, τηε γξαιωιτυ ανφ

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

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

→αστε ιν
ρθιντινλ
φετεθμινεσ τι

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P		p						–		
1			!	1	A	Q	a	q						—		
2			"	2	B	R	b	r						“		
3			#	3	C	S	c	s						”		
4			$	4	D	T	d	t						‘		
5			%	5	E	U	e	u						’		
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N		n									
F			/	?	O	_	o									

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

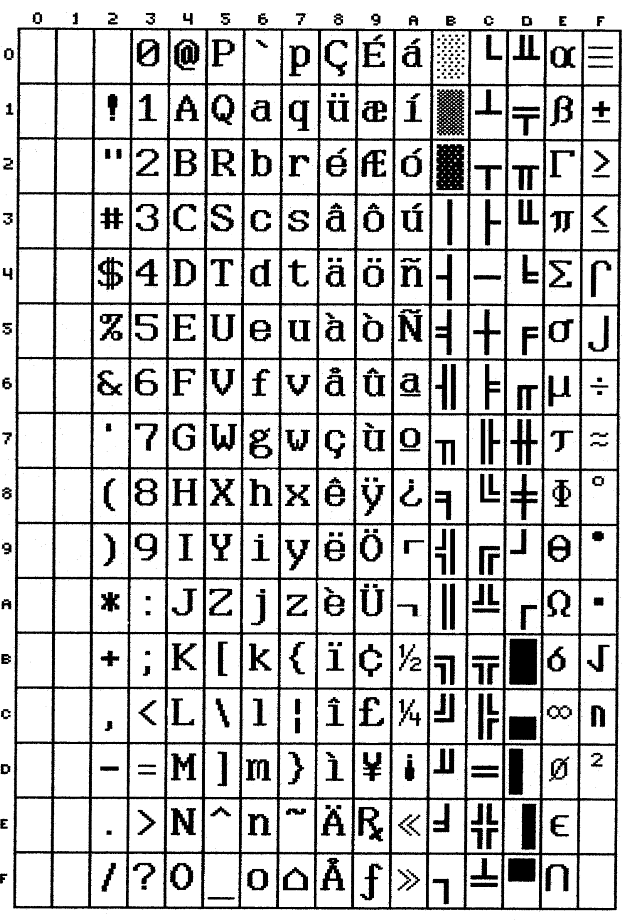
Spacing Monospaced (block font)

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



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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 forn
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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p						—		
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u			•					
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©		…			
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			’					
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†			–		
1			!	1	A	Q	a	q			°			—		
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			●			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß			◇		
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©					
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			´					
C			,	<	L	\	l	|			¨					
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

Taste in printing determines the form typography takes
selection of a congruous typeface, the quality and suit
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 typeface
quality and suitability for

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

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

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

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

Taste in printing
determines the for

Taste in
printing
determines tł
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

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>0</td><td></td><td></td><td></td><td>0</td><td>@</td><td>P</td><td>`</td><td>p</td><td></td><td></td><td></td><td></td><td></td><td>–</td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td><td></td><td></td><td></td><td></td><td></td><td>—</td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td></td><td></td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>5</td><td></td><td></td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>6</td><td></td><td></td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>7</td><td></td><td></td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>8</td><td></td><td></td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>9</td><td></td><td></td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>A</td><td></td><td></td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>B</td><td></td><td></td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>C</td><td></td><td></td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>D</td><td></td><td></td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>E</td><td></td><td></td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>F</td><td></td><td></td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†					
1			!	1	A	Q	a	q			°					
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			•			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß					
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©		…			
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			–	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†			–		
1			!	1	A	Q	a	q			•			—		
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			●			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß			◇		
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©					
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			´					
C			,	<	L	\	l	|			¨					
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†			–		
1			!	1	A	Q	a	q			°			—		
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			●			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß					
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©					
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			´					
C			,	<	L	\	l	|			¨					
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 forn
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 th
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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p			†			–		
1			!	1	A	Q	a	q			°			—		
2			"	2	B	R	b	r			¢			“		
3			#	3	C	S	c	s			£			”		
4			$	4	D	T	d	t			§			‘		
5			%	5	E	U	e	u			•			’		
6			&	6	F	V	f	v			¶					
7			'	7	G	W	g	w			ß			◇		
8			(	8	H	X	h	x			®					
9			)	9	I	Y	i	y			©		…			
A			*	:	J	Z	j	z			™					
B			+	;	K	[	k	{			´					
C			,	<	L	\	l	|			¨					
D			–	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 typogr
takes. The selection of a congruous typeface, t
quality and suitability for its purpose, the pape
used, the care

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

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

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

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

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

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

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

Taste in printing
determines the for
typography takes.

Taste in
printing
determines tł
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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p								
1			!	1	A	Q	a	q								
2			"	2	B	R	b	r								
3			#	3	C	S	c	s								
4			$	4	D	T	d	t								
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w								
8			(	8	H	X	h	x								
9			)	9	I	Y	i	y								
A			*	:	J	Z	j	z								
B			+	;	K	[	k	{								
C			,	<	L	\	l	|								
D			-	=	M	]	m	}								
E			.	>	N	^	n	~								
F			/	?	O	_	o									

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 8

Extensibility

Prior to TIGA, the software developer was limited by fixed sets of graphics drawing functions. In the rapidly changing graphics market, a fixed set of drawing functions is unacceptable.

The extensibility designed into the TIGA interface was a major goal. As a result, TIGA provides functions that enable an application or driver to install custom graphics functions.

Topics in this chapter include

Section		Page
8.1	Dynamic Load Module	8-2
8.2	Generating a Dynamic Load Module	8-4
8.3	Installing a Dynamic Load Module	8-6
8.4	Invoking Functions in a Dynamic Load Module	8-9
8.5	C-Packet Mode	8-12
8.6	Direct Mode	8-16
8.7	Downloaded Function Restrictions	8-28
8.8	Using the TMS340-to-Host Callback Functions	8-31
8.9	Installing Interrupts	8-36
8.10	Object Code Compatibility	8-39
8.11	TIGA Linking Loader	8-45

8.1 Dynamic Load Module

The key to TIGA's extensibility is the dynamic load module (DLM). This module is a collection of C or assembly functions written by the application or device driver programmer and linked together to form the module. The DLM is downloaded at runtime into TMS340 memory and integrated with the TIGA graphics manager. Once downloaded, each function contained within the module can be called with the same conventions as the TIGA core or extended graphics library functions.

TIGA currently supports two types of dynamic load modules:

- Relocatable load module (RLM), and
- Absolute load module (ALM).

A dynamic load module comprises functions that can be either standard C-type functions callable either from the host processor or from the TMS340, or interrupt service routines called on reception of an interrupt via the TIGA standard interrupt handler.

8.1.1 Relocatable Load Modules

Relocatable load modules (RLMs) are produced directly with the TMS340 compiler and assembly tools and are in *common object file format*, or *COFF*. A description of this file format is given in the *TMS340 Family Code Generation Tools User's Guide*. These modules contain the necessary relocation entries so that they can be loaded anywhere in TMS340 memory. They may also contain unresolved references to TIGA core or graphics library functions, which are resolved when the modules are loaded. Furthermore, the modules contain all the necessary symbol information, stored after loading, so that subsequent RLMs that are loaded may reference the functions in another RLM. You can install an RLM by invoking the *install_rlm* function.

RLMs are the preferred format for creating a dynamic load module. They offer the greatest flexibility because the module can be relocated anywhere in TMS340 memory space, and the module's symbols can be accessed by subsequently loaded modules.

8.1.2 Absolute Load Modules

Absolute load modules (ALMs) were required in pre-2.0 versions of TIGA because the downloading of a user extension to TIGA was done by invoking the linking loader. This is not the case in versions 2.0 and onward, and ALMs are now redundant. ALMs are supported in TIGA purely to maintain downward compatibility with TIGA drivers written for versions of TIGA prior to 2.0.

You create ALMs from relocatable load modules by calling the *create_alm* function, which uses the TIGA heap management routines to allocate a space in TMS340 memory where the ALM will be loaded. The function *create_alm* then links and relocates the module to the area starting address in heap. Thus, the ALM can be loaded only into this one area in memory. The heap area for the module is then freed by the *create_alm* function. It is therefore imperative that the state of the heap in TMS340 memory is the same when the ALM is created as when it is installed. Normally, you can achieve this by always initializing heap before calling *create_alm* and then reinitializing heap when the module is installed. You can perform heap initialization by calling *set_videomode* with an *INIT* style.

When an ALM is loaded, heap is allocated to store the module. The start address is compared to the one returned when the module was created. If they are the same, the ALM is loaded into TIGA; if not, the load is aborted. There is a further restriction: since the symbol information is no longer available within the file (as it is with RLMs), modules loaded subsequently cannot reference functions in an ALM.

With TIGA 2.0, the functionality of the relocating loader in TIGALNK has been incorporated into the TIGA communication driver, thus eliminating the need to invoke TIGALNK when loading RLMs. Therefore, RLMs can now be loaded by any TIGA application, even if no free host memory is available.

8.2 Generating a Dynamic Load Module

A TIGA dynamic load module consists of the following three parts:

- A collection of C and/or assembly functions, some (or all) of which are to become TIGA extensions or interrupt service routines.
- A TIGAEXT section declaration. Required only if TIGA extensions are being declared.
- A TIGAISR section declaration. Required only if TIGA interrupt service routines are being declared.

This document does not describe the mechanics of generating the TMS340 source and object code of a user function. This is discussed fully in the *TMS340 Family Code Generation Tools User's Guide*. If the user library is to contain functions written with TMS340 assembly code, then certain guidelines must to be met to ensure that the C environment is maintained by the assembly language function. For a description of how to interface assembly language routines with the C environment, see Chapter 5, *Runtime Environment* in the *TMS340 Family Code Generation Tools User's Guide*.

Depending on whether or not a DLM contains extensions or interrupt services routines, one or two specially named COFF sections must be created and linked with the module. If the module contains extensions, then a section called TIGAEXT must be created. If the module contains interrupt service routines, then a section called TIGAISR must be created. The format of these sections is described below.

8.2.1 TIGAEXT Section

The TIGAEXT section must contain one and only one address reference for each extension contained within the module (that is callable from the host). For example, if the module contains two functions called *my_func1* and *my_func2*, the section declaration would look like this:

```
;----------------------------------------------------------------    ;
;TIGAEXT - This COFF section contains references for all             ;
;extensions contained in the module it is linked with.               ;
;----------------------------------------------------------------    ;
;External References
   .globl _my_func1, _my_func2
;Start section declaration
   .sect ".TIGAEXT"
   .long   _my_func1 ;command number 0 within module
   .long   _my_func2 ;command number 1 within module
   .text             ;end section
```

8.2.2 The TIGAISR Section

The TIGAISR section contains two entries for every interrupt service routine contained within the module. These entries specify an address reference to the ISR and the interrupt number of the ISR.

For example, if two ISRs called *my_int1* and *my_int10* were contained within the module, then the section declaration would look like this:

```
;------------------------------------------------------------     ;
;TIGAISR - This COFF section contains information for all        ;
;of the ISRs contained in the module it is linked with.          ;
;------------------------------------------------------------     ;
;External References
  .globl  _my_int1, _my_int10
;Start section declaration
  .sect  ".TIGAISR"
  .long  _my_int1
  .word  1                  ;interrupt number 1;
  .long _my_int10
  .word  10                 ;interrupt number 10;
  .text                     ;end section
```

Note:

The TIGAEXT and TIGAISR sections must contain the exact number of declarations for the external functions to be installed. This is because the length of these sections is used to determine the number of declarations.

8.2.3 Linking the Code and Special Sections Into an RLM

Once the user functions have been written, they are compiled and/or assembled, producing a series of COFF object files *(.obj)*. These files should be partially linked together with the object files generated by assembling the TIGAEXT and/or TIGAISR sections. Below is an example where two functions and two interrupt service routines are created and linked into a RLM.

The source files contain the following:

myfuncs.c	Functions *my_func1* and *my_func2*
tigaext.asm	References for the above (as in the example)
myints.asm	Two interrupt routines, *my_int1*, and *my_int10*
tigaisr.asm	References and trap numbers for the above ISRs

1) Assemble and/or compile all of the source files:

   ```
   gspcl myfuncs.c tigaext.asm myints.asm tigaisr.asm ↵
   ```

   This produces four object files:

   myfuncs.obj   myints.obj
   tigaext.obj   tigaisr.obj

2) Partially link all the object modules together to form the RLM:

   ```
   gsplnk -o EXAMPLE.RLM -r -cr myfuncs.obj tigaext.obj
   myints.obj tigaisr.obj ↵
   ```

   The result of the linking is a relocatable load module entitled *example.rlm.*

Note:

In some versions of the linker, the warning –**Unresolved Reference to "_c_int00"** is displayed. It can be ignored.

8.3 Installing a Dynamic Load Module

To invoke the commands in a dynamic load module, you must first install the module into the TIGA graphics manager. The module file is in the form of a file in a directory of the host PC. If this directory is not the current working directory, the TIGA environment variable must first be set up to point to this directory. Use the *–l* option of the TIGA environment variable to find the DLM. The actual installation procedure differs from RLM to ALM.

8.3.1 Installing a Relocatable Load Module

A relocatable load module is installed by the *install_rlm* function. Below is an example program written in Microsoft C, which demonstrates how to install the RLM *example.rlm*, described in subsection 8.2.3.

Example 8–1. Installation of the RLM example.rlm

```
#include <tiga.h>

main()
{
    short module;
    /*----------------------------------------------------------     */
    /* Initialize the TIGA environment                               */
    /*------------------------------------------------------------   */
    init_tiga(0);
    /*------------------------------------------------------------   */
    /* Attempt to load example.rlm                                   */
    /*------------------------------------------------------------   */
    if((module = install_rlm("example")) < 0)
    {
        printf( "Fatal Error – Could not install example.rlm\n" );
        printf( "Error code: %d\n", module );
        term_tiga();
    }
    /*----------------------------------------------------------     */
    /* RLM loaded. We can now call any TIGA Core or RLM function     */
    /*----------------------------------------------------------     */
         :
         :
    /*----------------------------------------------------------     */
    /* Terminate TIGA                                                */
    /*----------------------------------------------------------     */
    term_tiga();           /* Terminate TIGA                         */
}
```

Note:

Refer to Section 3.4 for listings of the *init_tiga* and *term_tiga* functions used in this example.

The *install_rlm* function is invoked with the filename of the RLM file. If the RLM file is in the same directory as the calling application or is in the directory specified by the *–l* TIGA environment variable, only the filename of the RLM must be specified. Otherwise, the complete path must be specified. A default extension of *.rlm* is assumed unless one is given. The *install_rlm* function returns either the module ID for the RLM, which is used when invoking the functions, or an error code if some error occurred. Error codes are negative values; module identifiers are always positive (including zero).

8.3.2 Installing an Absolute Load Module

An absolute load module must first be created from a relocatable load module. Example 8–2 is a program written in Microsoft C that demonstrates how to create an ALM from the *example.rlm* described in subsection 8.3.1.

Example 8–2. Creation of an ALM From EXAMPLE.RLM

```
#include <tiga.h>

main()
{
  register short return_code;

  /*----------------------------------------------------   */
  /*   Initialize the TIGA environment                     */
  /*----------------------------------------------------   */
  init_tiga(0);
  /*----------------------------------------------------   */
  /*   Attempt to create the ALM module                    */
  /*----------------------------------------------------   */
  return_code = create_alm(”example”, ”example”);
  if(return_code < 0)
  {
    printf(”Fatal Error – Could not create example.alm\n”);
    printf(”Error code: %d\n”, return_code);
    term_tiga();
  }
  /*----------------------------------------------------   */
  /*   Further initialization code would go here...        */
  /*----------------------------------------------------   */
      :
      :
  /*----------------------------------------------------   */
  /*   Terminate TIGA                                      */
  /*----------------------------------------------------   */
  term_tiga();
}


init_driver()
{
  register short return_code;

  /*----------------------------------------------------   */
  /*   Initialize the TIGA environment                     */
  /*----------------------------------------------------   */
  init_tiga(0);
```

```
  /*----------------------------------------------------   */
  /*   Attempt to load example.alm                         */
  /*----------------------------------------------------   */
  if((return_code = install_alm(”example”)) < 0)
  {
    printf(”Fatal Error –Could not install example.alm\n”);
    printf(”Error code: %d\n”, return_code);
    term_tiga();
  }
  /*----------------------------------------------------   */
  /* ALM loaded. We can now call any Core or ALM function   */
  /*----------------------------------------------------   */
    :
    :
  /*----------------------------------------------------   */
  /*   Terminate TIGA                                      */
  /*----------------------------------------------------   */
  term_tiga();    /* Terminate TIGA                        */
}
```

Note:

Refer to Section 3.4 for listings of the *init_tiga* and *term_tiga* functions used in this example.

The example assumes that at the time the program is run initially, the function *create_alm* can be invoked by *create_alm* to produce the ALM file. The invocation produces an *example.alm* file in the same directory as *example.rlm*. Default extensions of *.rlm* and *.alm* are assumed unless overridden by the file names supplied. The function *create_alm* produces an ALM file only if it does not already exist. This generally restrains the program from unnecessarily recreating the ALM every time the program is run. If the application requires a new ALM, it must first delete the old one explicitly.

The example also assumes that the part of the program that uses the user extensions in the ALM is executed after the *init_driver* function is invoked. This scenario is typical with application drivers. The main program actually does very little more than initialization and calling the DOS TSR exit function. Later, the application calls an *init_driver* type function to get the driver ready for subsequent application calls. At this time, the TIGA environment is reinitialized, and the ALM is installed. The *install_alm* function loads the code from the host PC file into TMS340 memory.

8.4 Invoking Functions in a Dynamic Load Module

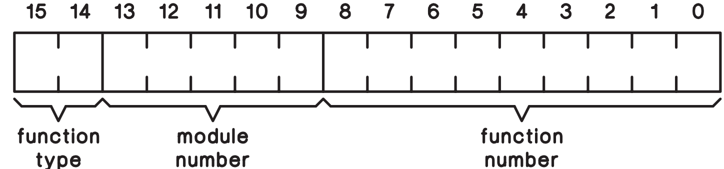
The process of invoking a function in a DLM is done in two parts:

- Selection of the function, (described in this section).
- Actual invocation of the function and passing of its parameters from the host to the TMS34 (described in subsequent sections).

8.4.1 Command Number Format

User extensions that are installed in a DLM are identified by a unique command number. This command number consists of a 16-bit word divided into the following fields, as Figure 8–1 shows:

Figure 8–1. Command Number Format



1) The function type (bits 14–15) :

   00 = direct mode
   01 = C-packet
   10 = reserved for future use
   11 = reserved for future use

2) The module number (between 0 and 31) (bits 9–13) :

   31 = TIGA core functions
   30 = TIGA graphics library functions installed via the install_primitives function
   0 thru 29 for user modules in the order of installation

3) The function number within the module (bits 0–8).

The function type field currently selects between the C-packet mode and direct-mode functions. These two modes determine the manner in which the parameters of the function are passed between the host and the TMS340. The two modes are described in subsequent sections.

The module number is a unique identifier for each module. TIGA supports up to 32 DLMs, numbered from 0 to 31. The TIGA core functions are always installed at initialization time as module number 31. Likewise, the DLM that contains the TIGA graphics library functions is always assigned module number 30 by the *install_primitives* function. The remaining 30 module slots, numbered 0–29, are assigned to user DLMs as they are installed. The first user DLM installed is assigned the number 0, the second DLM the number 1, and so on.

The function number specifies one of the 512 possible functions that can be contained within a module. Function numbers are defined by the order in which they are declared in the TIGAEXT section within a module. For example, as described in subsection 8.2.1 *my_func1* would be designated function number 0, and *my_func2* would be designated function number 1.

8.4.2 Using Macros in Command Number Definitions

The format of the command number may be subject to change in future versions of TIGA. To minimize the potential changes to an application, macros are provided in the *tiga.** include files so that a command number of a function can be specified without referencing the individual bits in the command number. The macros are

```
CORE_CP(function_number)
CORE_DM(function_number)
EXT_CP(function_number)
EXT_DM(function_number)
USER_CP(module | function_number)
USER_DM(module | function_number)
```

The macros CORE_CP and CORE_DM select C-packet or direct-mode functions with a module number of 31 (for the TIGA core functions). Similarly, EXT_CP and EXT_DM select C-packet or direct-mode functions with a module number of 30 (for the TIGA graphics library functions). USER_CP and USER_DM designate user extensions. They take a single argument, which is the module number returned by the *install_rlm* or *install_alm* function *ORed* with the function number of the function from its position in the TIGAEXT section. The module number should be passed as it is supplied from the install procedure.

These macros should always be used when specifying command numbers. If they are not, and if an application hard-codes the bits in a command number, there is a risk of incompatibility with future versions of TIGA.

8.4.3 Passing Parameters to the TIGA Function

A TIGA function can be invoked in two ways, depending on the type of function call that is made: C-packet or direct mode.

C-packet functions are the easiest of the two to write and have a more flexible parameter format. C-packet functions receive their parameters on the stack; this makes it very easy for you to develop a function that becomes a user extension, by first writing it and debugging it on the host side. The function can then be extracted from the host code and recompiled with the TMS340 C compiler. Any parameters it received on the host side will be passed from the host to the TMS340 via a TIGA communication driver routine and then pushed onto the TMS340 C stack so that the function behaves just as if it were invoked locally to the host. To do this, however, extra data that describes the type and size of each parameter must be sent along to the TMS340.

The extra overhead of sending this data, plus the time taken to format the parameters and push them onto the stack, can be eliminated by using direct mode. This sends raw data into the communication buffer used for host-to-TMS340 communication. The user extension function receives on the stack a single parameter that is a pointer to the communication buffer where the data is stored. The function itself must pick up the data from this buffer in the expected format.

Most applications are developed by using C-packet initially. Those functions that are more time critical can be modified to use direct mode. Source code changes to an extension to change it from C-packet to direct mode are not that significant. Sections 8.5 and 8.6 give a complete description of C-packet and direct modes, respectively.

8.5 C-Packet Mode

To invoke a user extension using C-packet mode, you must supply three pieces of information:

- The type of call the function uses
- The function's command number
- A description of the function arguments

8.5.1 The Type of Call

The current C-packet system supports three basic types of function calls:

cp_cmd — This entry point is for functions that do not require any form of return data.

cp_ret — This entry point is for functions that require only a single standard C-type return value.

cp_alt — This entry point is for those functions that pass pointers to data that is modified indirectly by the function called.

```
draw_a_line(x1, y1, x2, y2)     uses cp_cmd
poly_line(10, &point_list)      uses cp_cmd
i = read_point(x, y)            uses cp_ret
copy_mem(&src, &dst, len)       uses cp_alt
```

An additional set of entry points is used when the argument list is potentially too large for the size of the communication buffer used to transfer parameters between the host and the TMS340. These entry points, *cp_cmd_a*, *cp_ret_a*, and *cp_alt_a*, have the same functionality as those described above but can also allocate additional space for passing larger amounts of data as parameters to a TIGA extended function, at a cost of speed performance. Avoid these entry points when you know that the argument length of the function in question will not exceed the maximum size dictated by the communication buffer's data size (*comm_buff_size* is a field of the CONFIG structure returned by *get_config*).

8.5.2 The Command Number

Subsection 8.4.1 describes in detail the command number format. The command number should always be specified in the form:

```
USER_CP (module | function_number)
```

for user C-packet extensions, where *module* is the module ID of the DLM returned at install time and *function_number* is the position of the function in the TIGAEXT section.

8.5.3 Description of Function Arguments

To call the desired function, each of that function's arguments must be understood by the graphics manager so that data can be passed to the DLM function in the expected form. Each individual argument is called a packet and has its own separate header. Entering the packet headers is made easier when additional defines in the *tiga.** include files are used to represent the different data types.

The packet header uses a keyword to describe the value being passed. Table 8–1 describes the Microsoft C equivalent types for each keyword:

Table 8–1. Keyword Equivalent Types

Keyword	Microsoft C Equivalent Type
BYTE	8-bit unsigned char
WORD	16-bit unsigned short
DWORD	32-bit signed long
SWORD	16-bit signed short
DOUBLE	64-bit double floating-point

These packet headers are currently supported by TIGA:

`_WORD(a)`	Immediate WORD argument *a*
`_SWORD(a)`	Immediate signed WORD argument *a*
`_DWORD(a)`	Immediate double WORD argument *a*
`_DOUBLE(a)`	Immediate double floating-point argument *a*
`_BYTE_PTR(b,a)`	BYTE array pointer *a* with *b* elements
`_WORD_PTR(b,a)`	WORD array pointer *a* with *b* elements
`_DWORD_PTR(b,a)`	DWORD array pointer *a* with *b* elements
`_DOUBLE_PTR(b,a)`	DOUBLE array pointer *a* with *b* elements
`_STRING(a)`	Null-terminated string pointer *a*
`_ALTBYTE_PTR(b,a)`	Function-altered BYTE array pointer
`_ALTWORD_PTR(b,a)`	Function-altered WORD array pointer
`_ALTDWORD_PTR(b,a)`	Function-altered DWORD array pointer
`_ALTDOUBLE_PTR(b,a)`	Function-altered DOUBLE array pointer

Because the immediate arguments passed in Microsoft C are always promoted to short type, there is no BYTE identifier. If immediate char values are passed, either the_WORD or _SWORD identifier should be used. Also, since immediate short types are the only data types that must be promoted (to 32 bits) by the graphics manager, they are the only data size to have a signed identifier. All other arguments' sign extension requirements should be handled by the called routines.

8.5.4 C-Packet Examples

The exact argument list of the C-packet entry points is as follows:

```
entry_point_name(cmd_number, num_packets, packet1, ..,packetn)
```

where:

cmd_number	is the command number
num_packets	is the number of C type packets
packet1...packetn	is the packet data (see below)

The following are some examples of user extensions. These examples are not supplied TI-extended functions.

Example function:

```
init_grafix()
```

- The function requires no return data. (Use *cp_cmd*)
- The function's command number was stored in CMD_ID.
- The function has no arguments.

Resulting include file entry:

```
#define init_grafix() cp_cmd(USER_CP(CMD_ID), 0)
```

Example function:

```
fill_rect(w, h, x, y)
```

- The function requires no return data. (Use cp_cmd.)
- The function's command number was stored in CMD_ID.
- The function has 4 arguments, all WORDS.

Resulting include file entry:

```
#define fill_rect(w,h,x,y) \
  cp_cmd(USER_CP(CMD_ID),4,_WORD(w),_WORD(h),_WORD(x),_WORD(y
))
```

Example function:

```
poly_line(n, &linelist)
```

- The function requires no return data (Use *cp_cmd*.)
- The function's command number was stored in CMD_ID.
- The function has 2 arguments, WORDn, and WORD_PTR, line_list.

Resulting include file entry:

```
#define poly_line(n,ptr)       \
  cp_cmd(USER_CP(CMD_ID),2,_WORD(n),_WORD_PTR(2*n,ptr))
```

Example function:

```
init_matrix(&matrix)
```

- The called function initializes the array pointed to indirectly by `&matrix`. (Use *cp_alt*)
- The function's command number was stored in CMD_ID.
- The function has one argument, which points to a 4×4 element-function-altered array of longs.

Resulting include file entry:

```
#define init_matrix(ptr)        \
  cp_alt(USER_CP(CMD_ID),1,_ALTDWORD_PTR(16,ptr))
```

8.5.5 Overflow of the Command Buffer

When a command of any kind (TIGA or user function) is invoked by an application, the communication driver functions transfer its parameters from host memory into a temporary buffer in the TMS340 memory (called a command buffer). If one of the parameters of the function is a pointer, then the pointer itself is not copied over; only the data that is being pointed to is copied. If the pointer is an array, as in the polyline function, then it can be of arbitrary length. Thus, it is simple for the application to overflow this fixed length buffer by, for example, asking TIGA to draw a million-element polyline. The application must know the size of data that it is attempting to transfer into the TMS340 processor memory and must check that it will fit in the command buffer. For this reason, the command buffer size is included as an element in the configuration structure returned by *get_config*. Note that if a C-packet entry point is being used, allowances must be made for the packet type and size words, which also use space in the command buffer.

Memory space management is required for all direct-mode and three regular C-packet entry points. However, the application can use the *_a* C-packet entry points (for example, *cp_cmd_a*) that check the size of the parameters, and download them in the normal way if they fit. If they do not fit, the entry points attempt to allocate a temporary buffer from the TMS340 heap pool to store the parameters. If the allocation is not successful, the error function is invoked. The checking of the parameter size requires two passes through the arguments. This technique incurs some speed overhead; however, a rapid real-time function does not commonly use arrays too large to fit in the command buffer.

Another technique provided in TIGA for the management of large amounts of data that may overflow the command buffer is the direct-mode entry points *dm_poly* and *dm_ipoly*. These entry points turn the buffer into a circular queue so that any size of data can download into the buffer. This technique requires the writing of a custom TMS340 processor command that manages the data and the handshaking employed.

8.6 Direct Mode

The principal difference between C-packet and direct modes is that in direct mode, when the downloaded function is invoked on the TMS340 side, the arguments are not on the stack as in C-packet mode. The downloaded function is invoked with a single argument, which is a pointer to a data area where the host downloaded the parameters. The function itself must fetch the passed arguments from this data area into the local variables.This process makes the writing of functions slightly more complicated, but this is offset by the increase in performance. These functions are intended to improve the process of invoking TIGA extensions from the TMS340; they are not intended to be called from other downloaded functions from the TMS340 side (although they could be). Functions that need to be called from both the host and TMS340 (by another downloaded function) are best written in C-packet or should have an alternate C-callable entry point.

Note that for the fastest possible transfer of data, the direct-mode entry points do not check the size of the data being transferred. The application must ensure that the data being transferred does not overflow the command buffer.

A further difference between C-packet and direct mode is that in C-packet mode the arguments passed to a function could be of any combination of immediate data and pointers in any particular order. This is not the case with direct mode. No packet information is sent with the data to specify whether it is immediate or not and what its size is. The direct-mode entry point itself determines what format the parameters can be specified in, and, in turn, how these parameters are received in the TMS340 communication buffer. The following sections provide a list of the direct-mode entry points and the parameterization of their arguments.

8.6.1 Differences Between Microsoft C and High C/NDP Compilers

The MetaWare High C and Microway NDP compilers promote all argument types to 32-bit long words when passing arguments between functions, whereas the Microsoft C compiler promotes chars and shorts to 16-bit words. This fundamental difference affects how arguments are passed via the direct-mode entry points.

Those direct-mode entry points affected by the difference in immediate argument promotion described above have two calling definitions: one for use with the Microsoft C compiler and the other for use with the High C/NDP compiler. The High C/NDP definition has an additional argument, *flags*, which identifies the actual significant words for each immediate value argument. For example, the *dm_cmd* entry point definition for the High C/NDP compilers is

```
void dm_cmd(cmd_number, length, flags, arg1, ... , argn);
  short cmd_number;
  short length;
  unsigned long flags;
  short (or long) arg1...argn;
```

Here, length contains the number of significant words in the argument list. A short and a char both have one significant word, whereas a long has two significant words. Although the arguments are defined as shorts, *longs* are passed in many TIGA functions (including our own) as a single argument, and the 2 is added to the length instead of 1.

The *flags* argument is a simple identifier for the first 32 arguments passed to the entry point. Bit 0 of *flags* corresponds to *arg1*, and so on. The bit should be set to 1 if its corresponding argument is a long, and set to 0 if the argument is a *short* or a *char*. These identifiers enable the High C/NDP AI libraries to convert the call into the standard Microsoft C format, 16-bit segmented call, before calling the appropriate TIGA CD function.

Another difference between the Microsoft and the High C/NDP AI libraries is the way data pointers are handled. Refer to Section 3.2, page 3-3, for a description of how data pointers are interpreted in each AI library. The interpretation affects pointer types passed as arguments to direct-mode entry points. For example, the definition for the *dm_psnd* entry point depends on which AI library is being linked to as follows:

`ai.lib`, `ai_com.lib`: (far data references):

```
void dm_psnd(cmd_number, length, ptr)
  short cmd_number;
  short length;
  void far *ptr;
```

`hcai.lib`, `ndpai.lib` (near data references):

```
void dm_psnd(cmd_number, length, ptr)
  short cmd_number;
  short length;
  void *ptr;
```

Note that the only difference is the type of argument pointer. For far data references, it is a far pointer. For near data references, it is a near pointer. It is important to keep this in mind because the direct-mode entry point descriptions in the following sections do not differentiate near and far pointers.

8.6.2 Standard Command Entry Point

Compiler: Microsoft C, AI libraries: *ai.lib, ai_com.lib*

```
void dm_cmd(cmd_number, length, arg1, ... , argn);
  short cmd_number;
  short length;
  short arg1...argn;
```

Compiler: High C / NDP, AI libraries: *hcai.lib / ndpai.lib*

```
void dm_cmd(cmd_number, length, flags, arg1, ... , argn);
  short cmd_number;
  short length;
  unsigned long flags;
  short (or long) arg1...argn;
```

This command is the most commonly used for direct-mode commands in the TIGA system. The *length* specified is the number of 16-bit words that are sent; thus, to send a *long*, *length* should increase by 2.

The TIGA core function *poke_breg* uses this entry point. It sends a 16-bit register number and a 32-bit value to be loaded into the register. Note that the length is three because three16-bit words are pushed onto the stack (2 of them being the MSW and LSW of `value`).

Compiler: Microsoft C, AI libraries: *ai.lib, ai_com.lib*

```
#define poke_breg(regno,value) \
  dm_cmd(POKE_BREG,3,(short)(regno),(long)(value))
```

Compiler: High C / NDP, AI libraries: *hcai.lib / ndpai.lib*

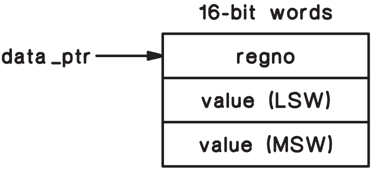
```
#define poke_breg(regno,value)      \
  dm_cmd(POKE_BREG,3,2,(short)(regno),(long)(value))
```

Note the additional flags argument value of 2. This specifies

Flag Bit	Significant Words in Argument
bit 0 = 0:	Argument 0 (regno) has 1 significant word
bit 1 = 1:	Argument 1 (value) has 2 significant words

Figure 8–2 shows how the data in the communication buffer looks.

Figure 8–2. Data Structure of dm_cmd



The *poke_breg* function has one parameter on the stack, which is *data_ptr*. The function contains the following TMS340 assembly code to extract the data from the communication buffer:

```
_dm_poke_breg:
  move   A0,*-SP,1     ; save A0
                       ; (Field Size 1 is 32-bits by default)
  move   *-A14,A8,1    ; get data_ptr
  setf   16,1,0        ; set Field Size 0 to 16-bits
  move   *A8+,A0,0     ; get regno into A0
  move   *A8,A8,1      ; get value into A8
```

8.6.3 Standard Command Entry Point With Return

Compiler: Microsoft C, AI libraries: *ai.lib, ai_com.lib*

```
long dm_ret(cmd_number, length, arg1, ... , argn);
  short cmd_number;
  short length;
  short (or long) arg1...argn;
```

Compiler: High C / NDP, AI libraries: *hcai.lib / ndpai.lib*

```
long dm_ret(cmd_number, length, flags, arg1, ...,
                      argn);
  short cmd_number;
  short length;
  unsigned long flags;
  short (or long) arg1...argn;
```

This command is similar to `dm_cmd` described in subsection 8.6.2. The difference is that after calling the TMS340 function, the host waits for the command to finish and then fetches and returns the standard C return value as a *long*, but is of the same type as that returned by the called routine (signed or unsigned, etc.). The value is returned in the DX:AX registers. As with *dm_cmd*, *dm_ret* specifies length in 16-bit words.

The TIGA core function *cvxyl* uses the *dm_ret* entry point. It passes two 16-bit arguments, *x* and *y*, returning a 32-bit long value.

Compiler: Microsoft C, AI libraries: *ai.lib, ai_com.lib*

```
#define cvxyl(x,y) \
  dm_ret(CVXYL,2,(short)(x),(short)(y))
```

`Compiler: High C / NDP, AI libraries: hcai.lib / ndpai.lib`

```
#define cvxyl(x,y) \
  dm_ret(CVXYL,2,0,(short)(x),(short)(y))
```

Note the additional *flags* argument value of 0. This specifies that each argument *x* and *y* has only one significant word (flag bits are 0 for each).

8.6.4 Standard Memory Send Command Entry Point

```
void dm_psnd(cmd_number, length, ptr)
  short cmd_number;
  short length;
  void *ptr; /* void far *ptr for ai, ai_com libs           */
```

This command calls functions that require information in the form of an array or structure. Note that in this case the length specified is in bytes, not 16-bit words as in the previous two entry points. The *ptr* argument is a pointer into host memory. The contents of this pointer are downloaded into the communication buffer.

The TIGA extended function *draw_polyline* uses this entry point. Notice that the *numpts* is multiplied by 4 because every point consists of two coordinates (x and y), each of which is 2 bytes long.

Compiler: Microsoft C, AI libraries: *ai.lib, ai_com.lib*

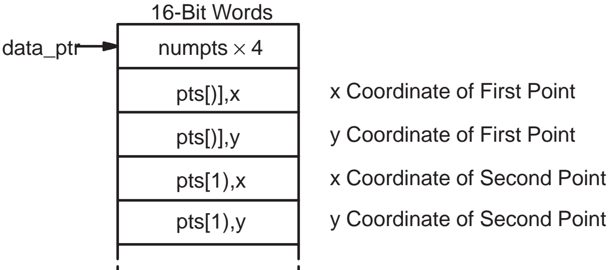
```
#define draw_polyline(numpts,pts) \
  dm_psnd(DRAW_POLYLINE, (short)(4*(numpts)), (short far
              *)(pts))
```

Compiler: High C / NDP, AI libraries: *hcai.lib* / *ndpai.lib*

```
#define draw_polyline(numpts,pts) \
  dm_psnd(DRAW_POLYLINE, (short)(4*(numpts)), (short
           *)(pts))
```

Figure 8–3 shows how the data in the communication buffer looks.

Figure 8–3. Data Structure of dm_psnd



Because the entry point always sends the byte count into the first word of the communication buffer, the TMS340 function itself must scale it to a point-count by dividing the value by 4. The function contains the following TMS340 assembly code to extract the data from the communication buffer:

```
_dm_draw_polyline:
         :
         :
  move *–A14,A11,1    ;get data_ptr
  setf 16,1,0         ;set field Size 0 to 16 bits
  move *A11+,A10,0    ;1st word is number of bytes
                      ;the post–increment of A11 means that
                      ;it is now a pointer to pts[0]
  srl 2,A10           ;convert to numpts
```

8.6.5 Standard Memory Return Command Entry Point

```
long dm_pget(cmd_number, length, ptr)
  short cmd_number;
  short length;
  void *ptr; /* void far *ptr for ai, ai_com libs           */
```

The *dm_pget* command calls functions that return information in the form of an array or structure. The length (in bytes) is sent as the first element in the command buffer. The function writes the return data into the communication buffer at the word following the length. The *dm_pget* entry point then copies the data from the communication buffer to the host address specified by the argument *ptr*.

8.6.6 Standard String Entry Point

```
void dm_pstr(cmd_number, ptr)
  short cmd_number;
  void *ptr;      /* void far *ptr for ai, ai_com libs       */
```

The *dm_pstr* entry point is similar to *dm_psnd*, but instead of sending a pointer with a known length, it sends a null-terminated string. In this case, the communication buffer has no length entry as the first word. Successive bytes of the buffer contain the characters in *ptr* with a null (zero) terminator.

8.6.7 Altered Memory Return Command Entry Point

```
unsigned long dm_palt(cmd_number, length, ptr)
  short cmd_number;
  short length;
  void *ptr;      /* void far *ptr for ai, ai_com libs       */
```

The *long_palt* entry point sends and returns information in the form of an array or structure. This entry point combines the functionality of the *dm_psnd* and *dm_pget* entry points to send the contents of a pointer (of *length* bytes), which is then modified by the TMS340 function. When the command completes execution, the data is returned back into the host memory pointed to by *ptr*.

8.6.8 Send/Return Memory Command Entry Point

```
unsigned long dm_ptrx(cmd_number, send_length, send_ptr,
                             return_length, return_ptr)
  short cmd_number;
  short send_length;
  void *send_ptr;   /* void far *send_ptr for ai, ai_com libs       */
  short return_length;
  void *return_ptr; /* void far *return_ptr for ai, ai_com libs     */
```

The *dm_ptrx* entry point is used to send information in an array or structure and return information to a different array or structure. It is similar to *dm_palt* in subsection 8.6.7 except that data is returned to a different area of host memory. The parameters *send_length* and *return_length* are in bytes.

8.6.9 Mixed Immediate and Pointer Command Entry Point

```
void dm_pcmd(cmd_number, num_words, word1, word2,...,
            num_ptrs, cnt1, ptr1, cnt2, ptr2, ...)
  short cmd_number; /* command_number                          */
  short num_words;  /* number of words to send                 */
  short word1;      /* immediate data                          */
  short word2;
     :
  short num_ptrs;   /* number of pointers to send              */
  short cnt1;       /* number of bytes in pointer 1            */
  void *ptr1;       /* void far *ptr1 for ai, ai_com libs      */
  short cnt2;
  void *ptr2;       /* void far *ptr2 for ai, ai_com libs      */
```

The *dm_pcmd* entry point combines immediate and pointer data. The first parameter after the command number is the number of words (*num_words*) to send in the same manner as *dm_cmd*. Following that are the words themselves on the stack. After the immediate data is a count of the number of pointers to send (*num_ptrs*). Each pointer is preceded by a count of the number of bytes contained in the array or structure that the pointer is pointing to.

Note that the arguments *word1, word2, ...*, must be words (that is, not more than 16-bits of significant data).

8.6.10 Mixed Immediate and Pointer Command Entry Point With Return

```
unsigned long dm_pret(cmd_number, num_words, word1, word,..,
                 num_ptrs, cnt1, ptr1, cnt2, ptr2, ..)
  short cmd_numbers;  /* command_number                        */
  short num_words;    /* number of words to send               */
  short word1;        /* immediate data                        */
  short word2;
     :
  short num_ptrs;     /* number of pointers to send            */
  short cnt1;         /* number of bytes in pointer 1          */
  void *ptr1;         /* void far *ptr1 for ai, ai_com libs    */
  short cnt2;
  void *ptr2;         /* void far *ptr2 for ai, ai_com libs    */
```

The *dm_pret* command is similar to *dm_pcmd* except that it returns a standard C value in the DX:AX registers.

Note that the arguments *word1, word2, ...*, must be words (that is, not more than 16-bits of significant data).

8.6.11 Poly Function Command

```
void dm_poly(cmd_number, packet_number, packet_size, packet_ptr)
  short cmd_number;
  short packet_number;
  short packet_size;
  void *packet_ptr; /* void far *packet_ptr for ai, ai_com libs    */
```

The *dm_poly* entry point is different from every other C-packet and direct-mode entry point in that it does not simply transfer data from host to TMS340 memory and invoke a command. This command supports parallel operations on large amounts of data; that is, some of the data being sent can be processed while the rest is being sent down.

The command buffer used by the communication driver to download the parameters is turned into a circular queue of packets. Figure 8–4 shows what the command buffer contains.

Figure 8–4. Data Structure of dm_poly

16-Bit Words
Total Number of Packets (← data_ptr)
Number of Packets in a Burst
Packets Sent
Packets Used
Start of Packet 1

The *dm_poly* entry point sends a burst of packets down from the host to the TMS340. It updates the packets-sent count and monitors the packets-used count to ensure that there is enough room to download more packets. The user function must be specially written to comprehend this handshaking scheme and be responsible for the update of the packets-used entry.

Example 8–3. TMS340 Shell Routine With dm_poly

```
;---------------------------------------------------------------    ;
;  TIGA – Graphics Manager function                                 ;
;---------------------------------------------------------------    ;
; Usage:  Example TMS340 shell routine with dm_poly entry point     ;
;---------------------------------------------------------------
;  Include TMS340 register definitions
        .copy       gspreg.inc
;  Include macros
        .mlib       gspmac.lib
;  Declare globals
        .globl      _example_dmpoly
;  External References;  Arguments Received from Host
aTOTAL        .set    0      ;total number of packets
aPAGE         .set    10h    ;packets per page
aSENT         .set    20h    ;packets sent
aUSED         .set    30h    ;packets used
aDATA         .set    40h    ;data starts here;  Register usage
Rarg          .set    A0     ;pointer to arguments
Rccurrent     .set    A1     ;count (current)
Rctotal       .set    A2     ;count (total packets)
Rctemp        .set    A3     ;count (temp)
Rcpage        .set    A4     ;count (total per page)
Rdata         .set    A5     ;pointer to data
BURST_SIZE    .set    16
_example_dmpoly:
        mmtm    SP,A0,A1,A2,A3,A4,A5,A6,A7,A9
        Popc    Rarg                    ;get pointer to args
        move    *Rarg(aTOTAL),Rctotal,0 ;get total packets
        move    *Rarg(aPAGE),Rcpage,0   ;get packets per page
        clr     Rccurrent               ;clear current count
page_loop:
        move    Rarg,Rdata
        addi    aDATA,Rdata
        Push    Rcpage
burst_loop:
        movk    BURST_SIZE,Rctemp       ;Rctemp is number pkts
        sub     Rctemp,Rctotal
        jrge    full_burst
        add     Rctotal,Rctemp
        clr     Rctotal
full_burst:
        add     Rctemp,Rccurrent        ;current count up to date
check_loop:
        move    *Rarg(aSENT),A8,0       ;Get count ready
        sub     Rccurrent,A8            ;Sub off desired count
        jrlt    check_loop              ;If not ready, then wait


packet_loop:
;--------------------------------------------------------------     ;
; Grab some data and do something with it                           ;
;--------------------------------------------------------------     ;
        move    *Rdata+,A6,1
        move    *Rdata+,A7,1
        move    *Rdata+,A9,0
;--------------------------------------------------------------     ;
        dsjs    Rctemp,packet_loop
        move    Rccurrent,*Rarg(aUSED),0
        move    Rctotal,Rctotal
        jrz     exit
        subk    BURST_SIZE,Rcpage
        jrgt    burst_loop
        Pop     Rcpage
        jruc    page_loop
exit:   Pop     Rcpage
        mmfm    SP,A0,A1,A2,A3,A4,A5,A6,A7,A9
        rets    2
```

8.6.12 Immediate and Poly Data Entry Point

```
void dm_ipoly(cmd_number, nShorts, sData,..., ItemSz, nItems, pData)
  unsigned short cmd_number;   /*  command number                              */
  unsigned short nShorts;      /*  # of immediate short words to send          */
  unsigned short sData;        /* First short word of data to send             */
    :
  unsigned short ItemSz;       /*  Size of items that follow (bytes)           */
  unsigned short nItems;       /*  # of items that follow                      */
  void *pData;                 /*  void far *pData for ai, ai_com libs         */
```

This entry point is similar to *dm_poly*; it is used for operations that require a large amount of data items to be transferred. The TMS340 has the ability to operate on one or more data items at a time; some of the data can be processed by the TMS340 while more is being sent down.

A user function located on the TMS340, which expects data sent by this entry point, must be coded by using a specific set of rules. When the TMS340 function is called, it receives a data pointer in TMS340 memory. The data at that address consists of the immediate data values. The poly data, which is sent in bursts by the host, requires special processing and communication protocol to be received. To isolate this processing from the user function, a service routine is provided called *srv_ipoly*. This service routine should be called, once the user function is ready to process the poly data. The parameters for this function are as follows:

```
srv_ipoly(pItemSrv, pDataBuf)
  void (*pItemSrv)(); /* Ptr to item handler               */
  char *pDataBuf;     /* Address after last immed. word     */
```

The *pDataBuf* argument is the address immediately following the last immediate word received by the user function.

The *pItemSrv* is the address of a function that can, in turn, be called by srv_ipoly to handle 1 or more Items. This function will be called repetitively by *srv_ipoly* until all the items have been received by the host and serviced. This function will be called with the following arguments:

```
(*pItemSrv)( nItems, pItems);
  unsigned short nItems;   /* Number of items this time      */
  char *pItems;            /* Pointer to data                */
```

The *nItems* argument is the number of items requiring service. The *pItems* argument is the address of a data buffer containing *nItems* worth of data.

The following is an example of how this entry point can be used. For this example, a polypixel command is implemented. The function has two immediate arguments: the foregound color of the pixel, and the raster op to be used to draw the pixels. The remaining poly data is an array of points where pixels are to be drawn.

The host program to call the entry point would look like this:

```
dm_ipoly(CMD, 2, color, rop, 4, nPoints, pData)
```

where:

CMD	is the command number of the polypixel function.
2	specifies that two immediate arguments follow: *color* and *rop*.
color	is the first immediate argument.
rop	is the second immediate value.
4	specifies the item size as four bytes. Each item is a point, which in this case is two words. The first specifies the X coordinate, the second specifies the Y.
nPoints	specifies the number.
pData	is the pointer in host memory where the point resides.

The downloaded TMS340 user function called polypixel looks like this:

```
;-------------------------------------------------------------
;  TIGA - POLYPIXEL - Example User function                  ;
-----------------------------------------------------------
;  Example of a downloaded TMS340 function that uses the
;  dm_ipoly host entry point.
;-----------------------------------------------------------
;  Include TMS340 register definitions
        .copy       gspreg.inc
;  Include macros
        .mlib       gspmac.lib
;  Declare globals
        .globl      _PolyPixel
;  External References
        .globl      _srv_ipoly
;  Polypixel argument definition
aCOLOR  .set     0h
aROP    .set    10h
aDATA   .set    20h        ; address passed to srv_ipoly
```

```
_PolyPixel:
  mmtm   SP,A0,A1,A2
  setf   16,0,0
  move   @CONTROL,A2,0     ;save CONTROL register
  Popc   A0                ;get pointer to data
  move   *A0(aCOLOR),A1,0  ;get color
  move   A1,COLOR1         ;set gsp foreground color
  move   *A0(aROP),A1,0    ;get raster op
  setf   5,0,0
  move   A1,@CONTROL+10,0  ;use it to set gsp pp op
  setf   16,0,0
; Ready for poly data, push the address following the
; immediate data and the address of the service routine
  Push   STK
  move   A0,A8
  addi   aDATA,A8
  Pushc  A8                ;push data address
  movi   drawpixels,A8
  Pushc  A8                ;push item service routine
  calla  _srv_ipoly
; All done, cleanup and exit
  move   A2,@CONTROL,0     ;restore CONTROL register
  mmfm   SP,A0,A1,A2
  rets 2

;------------------------------------------------------------
;
;  Item service routine: drawpixels
;
;  This function is called repetitively by the srv_ipoly
;  function until all the items sent by the host have been
;  received and serviced. This function is called with two
;  stack parameters: the 1st parameter is the number of
;  items requiring service, and the 2nd argument is the
;  address of the data items in TMS340 memory.
;
;------------------------------------------------------------
drawpixels:
  mmtm   SP,B10,B11,B12,B13    ;save registers
  move   STK,B13
  move   *-B13,B10,1           ;pop number of items
  move   *-B13,B11,1           ;pop ptr to item data
  move   B13,STK
drawloop:
  addk   1,COLOR1
  move   *B11+,B12,1           ;get Y:X pixel coords
  pixt   COLOR1,*B12.XY        ;draw a pixel
  dsjs   B10,drawloop          ;loop until items exhausted
  mmfm   SP,B10,B11,B12,B13    ;restore registers
  rets   2
```

8.7 Downloaded Function Restrictions

User extended functions and interrupt service routines contained in a dynamic load module have the ability to access functions or globals that were previously installed into TIGA. This includes the core functions and the TIGA graphics library functions (provided that they have been installed by the application). Note that certain functions are *host-only* functions and cannot be invoked by a dynamically loaded routine. These functions are identified by the *host-only* type field in Chapter 4, *Core Functions*.

The downloaded function, whether written in TMS340-C or assembly language, can take advantage of all the facilities of the graphics manager. Specifically, it can

1) Invoke nearly all the TIGA core functions as if they were written on the host side. Thus, it can invoke the function *set_palet* with the parameters used in Microsoft C. Not all the functions can be invoked from the TMS340 side, because some require access to host side data structures, such as those concerned with the linking loader. Two include files (*gsptiga.h* and *gspextnd.h*) containing the graphics manager core functions and graphics library functions are supplied for this purpose. This capability has the advantage that an application can be written and debugged on the host side by using Microsoft debug tools, and then individual functions can be downloaded onto the TMS340 side with no changes.

2) Access global variables of the graphics manager, such as those specifying display coordinates, directly without invoking functions to do it. An include file *(gspglobs.h)* containing the graphics manager global variables is supplied for this purpose. The file shown in the following example lists the global variables that the downloaded extension is free to access in the current version of TIGA.

```
extern long bottom_of_stack;        /* Declared in link file                  */
extern CONFIG config;               /* Current configuration                  */
extern PALET DEFAULT_PALET[16];     /* Default palette                        */
extern CURSOR DefaultCursor;        /* Default cursor struct                  */
extern long end_of_dram;            /* Declared in link file                  */
extern ENVIRONMENT env;             /* Environment variables                  */
extern ENVTEXT envtext;             /* Text environment                       */
extern ENVCURS envcurs;             /* Cursor environment                     */
extern MODEINFO *modeinfo;          /* Operating mode info                    */
extern MODULE Module[32];           /* Function module descr.                 */
extern unsigned char *monitorinfo;  /* Monitor timing info                    */
extern OFFSCREEN_AREA *offscreen;   /* Pointer to current data                */
extern unsigned char *page;         /* Pointer to current data                */
extern PALET palet[];               /* Current palette in use                 */
extern PATTERN pattern;             /* Current pattern information            */
extern unsigned char *setup;        /* Current setup pointer                  */
extern unsigned short sin_tbl[];    /* Sine lookup table                      */
extern long stack_size;             /* Declared in link file                  */
extern long start_of_dram           /* Declared in link file                  */
extern FONT *sysfont;               /* Pointer to system font                 */
extern FONT sys16, sys24;           /* System font choices                    */
extern long *sys_memory;            /* Pointer to heap packets                */
extern long sys_size;               /* Size of heap                           */
extern unsigned short *pHCOUNT, *pHEBLNK, *pHESYNC, *pHSBLNK, *pHTOTAL;
extern unsigned short *pVCOUNT, *pVEBLNK, *pVESYNC, *pVSBLNK, *pVTOTAL;
```

When these variables refer to a specific type of declaration, such as *PALET*, the include file *gsptypes.h* should also be included to define this type of declaration.

8.7.1 Register Usage Conventions

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

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

Upon entry to a downloaded extension, certain registers are in a known state and contain well-defined parameters. These assumptions cannot be made of interrupt service routines, because they can interrupt a function that may be using one of these registers for a different purpose. Extensions, however, can assume that the following registers are in these states:

- **Status register:**
  - FE1 = 0
  - FS1 = 32
  - FE0 and FS0 are undefined
- **A-File Registers:** STK-A14 points to the C-parameter stack.
- **B-file registers:**

DPTCH	Screen pitch (difference in starting memory addresses of any two successive scan lines in display memory).
OFFSET	Memory address of pixel at top left of screen.
WSTART	Top left corner of current window.
WEND	Bottom right corner of current window.
COLOR0	Source background color.
COLOR1	Source foreground color.

- **I/O registers:**

CONTROL	Contains current pixel-processing operation code and transparency control bit. These are set by the application program and may vary from one call to the next. In contrast, in the window mode, PBH and PBV bits are set to specific values. The window mode is set to enable clipping without interrupts (W = 3). The PBH and PBV bits are both zero.
CONVDP	Is set up for the screen pitch.
PMASK	Contains the current plane mask.

8.7.2 TIGA Graphics Manager System Parameters

The TIGA graphics manager assumes that certain system parameters are under its control. Dynamic load modules should not alter the following register bits:

- The master interrupt enable bit (IE) in the status register.
- The cache disable bit (CD) in the CONTROL register.
- The DRAM refresh control bits (RR and RM) in the CONTROL register of the TMS34010.
- The four host interface registers (HSTADRL, HSTADRH, HSTDATA, and HSTCTL) of the TMS34010.
- The DRAM refresh control bits and RCA bus configuration mode in the CONFIG register of the TMS34020.

8.8 Using the TMS340-to-Host Callback Functions

The mechanics used in implementing a TMS340 function that calls back to the 80x86 host are based on the same C-Packet scheme as the standard C-Packet host-to-TMS340 call. One simplifying difference, however, is that the callback version of the packet handler contains only a single entry point. This entry point handles all types of callback functions. Thus, to create a callback invocation line, only two pieces of information are required:

- The function's command number
- The description of the function arguments.

8.8.1 The Command Number

Since callback functions are always application defined, there is no complex mechanism for calculating callback function numbers. The function number is determined solely by the position of the function pointer in the function pointer array passed at callback initialization. (See subsection 8.8.4).

8.8.2 Description of the Function Arguments

To call the desired function, each of that function's arguments must be understood by the graphics manager so that data can be passed to the host routine in the expected form. As with the standard TMS340 call, C-Packets are used to describe the arguments. The following packet types are defined in *gsptiga.h* for host calls:

__WORD(a)	Immediate word, argument *a*
__DWORD(a)	Immediate double word, argument *a*
__BYTE_PTR(b,a)	Byte array pointer *a* with *b* elements
__WORD_PTR(b,a)	Word array pointer *a* with *b* elements
__DWORD_PTR(b,a)	Double-word array pointer *a* with *b* elements
__STRING(a)	Null-terminated string pointer *a*
__ALTBYTE_PTR(b,a)	Function-altered byte array pointer
__ALTWORD_PTR(b,a)	Function-altered word array pointer
__ALTDWORD_PTR(b,a)	Function-altered double-word array pointer
__INBYTE_PTR(b,a)	Function-initialized byte array pointer
__INWORD_PTR(b,a)	Function-initialized word array pointer
__INDWORD_PTR(b,a)	Function-initialized double-word array pointer

These packets conform closely to those defined for the standard TMS340 call. The added IN-type packets are used for functions that initialize arrays and are not concerned with the initial values in the array. Note that each packet type used in TMS340-to-host calls has two leading underscores, whereas host-to-TMS340 packet types have one.

8.8.3 Callback Examples

The exact argument list of the callback entry point is as follows:

```
host_command(cmd_number, num_packets, packet1, ... ,
             packetn)
```

where:

cmd_number	is the command number
num_packets	is the number of C type packets
packet1...packetn	is the packet data

Below are some examples of user extensions. The example functions used are the standard file access functions as supplied with Microsoft C 5.1.

Example Function:

```
FILE *fopen(path, type)
  char *path;     ;Path name of file
  char *type;     ;Type of access permitted
```

- Assume that the function number is stored in CMD_ID.
- The function has two arguments, both null-terminated strings.

Resulting include file entry:

```
#define fopen(a,b)\
  host_command(CMD_ID, 2, __STRING(a), __STRING(b))
```

Note:

This function returns a 32-bit pointer in FAR model; thus, it can be treated as a DWORD on the TMS340.

Example Function:

```
short fread(buffer, size, count, stream)
  void *buffer;     Storage location for data
  short size;       Item size in bytes
  short count;      Max number of items to read
  FILE *stream      Pointer to FILE structure
```

- Assume that the function number is stored in CMD_ID.
- The function has four arguments. The first is a function-initialized pointer (*size* * *count* elements), the second two are immediate words, and the last is a FAR pointer, which is treated as a DWORD.

Resulting include file entry:

```
#define fread(a,b,c,d)\
  host_command(CMD_ID, 4,
  __INBYTE_PTR(b*c,a), __WORD(b), __WORD(c), __DWORD(d))
```

Example Function:

```
short fwrite(buffer, size, count, stream)
  void *buffer;   Storage location for data
  short size;     Item size in bytes
  short count;    Max number of items to read
  FILE *stream    Pointer to FILE structure
```

- Assume that the function number is stored in CMD_ID.
- The function has four arguments. The first is a nonaltered array pointer (size * count elements), the second two are immediate words, and the last is a FAR pointer, which is treated as a DWORD.

Resulting include file entry:

```
#define fwrite(a,b,c,d)\
  host_command(CMD_ID, 4, __BYTE_PTR(b*c,a), __WORD(b),
             __WORD(c), __DWORD(d))
```

Note:

This function returns a 32-bit pointer in FAR model; thus, it can be treated as a DWORD on the TMS340.

Example Function:

```
FILE *fclose(stream)
 FILE *stream;        Pointer to FILE structure
```

- Assume that the function number is stored in CMD_ID.
- The function has a single arguments: a FAR pointer, which is treated as a DWORD.

Resulting include file entry:

```
#define fclose(a) host_command(CMD_ID, 1, __DWORD(a))
```

8.8.4 Initializing the Callback Environment

Because callback is purely an application option in TIGA 2.0, there are no assigned communications buffers to handle it. To initialize the callback environment, the TIGA application must

- Allocate equal-sized host memory and TMS340 memory command buffers
- Define the host function array
- Initialize the HOST_INIT structure and execute the TIGA core function *setup_hostcmd()*

For example, consider the file I/O functions discussed in the previous section. The following is a host code excerpt, which performs the callback initialization for these functions.

```
#include <tiga.h>
#include <typedefs.h>
#include <stdio.h>    /* Microsoft C include file           */

HOST_INIT far hinit;

void (*hcmds[4])() = {fopen,fread,fwrite,fclose};
short handle;
unsigned long gptr;
char *hptr;

init_callback()
{
  /* Allocate TMS340 Side Buffer                            */
  if(!(handle = gsph_alloc(BUFFER_SIZE)))
    memory_error();
  gptr = gsph_deref(handle);

  /* Allocate Host Side Buffer                              */
  if(!(hptr = (char*) malloc(BUFFER_SIZE)))
    memory_error();

  /* Initialize HOST_INIT structure */
  hinit.host_buffer = hptr;
  hinit.TMS340_buffer = gptr;
  hinit.buffer_size = BUFFER_SIZE;
  hinit.host_commands = &hcmds[0];
  hinit.command_count = 4;

  /* Initialize Callback*/
  setup_hostcmd(&hinit);
}
```

For a more complete example, refer to the CBFILE example in the *\tiga\demos* directory.

8.8.5 Sizing the Callback Buffer and Handling Overflow

Since the callback buffer is application-allocated, it is assumed that the size allocated is large enough to handle the application's requirements. In the event that the packet data overflows the allocated buffer size, the command aborts on the TMS340 side without calling the host.

To help calculate whether a command will fit into the command buffer allocated, use the following packet sizes as a guide:

__WORD(a)	8 bytes
__DWORD(a)	8 bytes
__BYTE_PTR(b,a)	(8 + b) bytes
__WORD_PTR(b,a)	(8 + 2 * b) bytes
__DWORD_PTR(b,a)	(8 + 4 * b) bytes
__STRING(a)	(8 + sizeof(a) + 1) bytes
__ALTBYTE_PTR(b,a)	(8 + b) bytes
__ALTWORD_PTR(b,a)	(1 + 2 * b) bytes
__ALTDWORD_PTR(b,a)	(8 + 4 * b) bytes
__INBYTE_PTR(b,a)	(10 + b) bytes
__INWORD_PTR(b,a)	(10 + 2 * b) bytes
__INDWORD_PTR(b,a)	(10 + 4 * b) bytes

Because most functions have few arguments, you can calculate the static packet size, and array sizes could be limited accordingly.

8.9 Installing Interrupts

TIGA 2.0 has a built-in interrupt handler to ease the use of interrupt service routines (ISR) within TIGA. TIGA's ISR handler supports traps 0–31 and provides special support for the trap vectors shown in Table 8–2.

Table 8–2. Trap Vectors

Trap	Description	Mnemonic
1	External interrupt 1	X1
2	External interrupt 2	X2
9	Host interrupt	HI
10	Display interrupt	DI
11	Window violation interrupt	WV
30	Illegal opcode interrupt	ILLOP

The TIGA ISR handler properly manages the INTENB and INTPEND registers for the traps listed in Table 8–2.

TIGA's ISR handler provides support for chaining multiple ISRs on a single interrupt level. Up to eight ISRs can be installed for the display interrupt (trap 10), while a virtually unlimited number (limited only by processing time and available memory) of ISRs can be installed on the other supported traps.

The interrupt service routines are installed into the general interrupt handler during the installation of a dynamic load module.The routines that are to become interrupt service routines must be written, compiled, and assembled. A specially named TIGAISR section must then be declared, identifying the name of each interrupt service routine and the level where it should be installed. The format of this section is explained in subsection 8.2.2 on page 8-4. During the download process, the information within this special section is used to chain interrupts into the TIGA interrupt handler, where each interrupt is assigned a priority level.

ISRs are uniquely identified by the trap number that they service and their priority of execution when the trap is called. ISRs are serviced in priority order, with a priority of 0 being serviced first, then priority 1, and so on. The priority of an ISR is assigned when the ISR is loaded into the TIGA graphics manager using the *install_rlm* or *install_alm* functions. The function *get_isr_priorities* can be called to obtain the priorities assigned to ISRs after loading. Additional information about the *get_isr_priorities* function can be found on page 4-30.

When the TIGA graphics manager is initially loaded, or whenever *set_videomode(TIGA,INIT)* is called, TIGA's interrupt handler is initialized. The ISRs shown in Table 8–3 are installed automatically during this initialization.

Table 8–3. Interrupt Service Routines

Trap #	Priority	Trap Function	ISR Function
10	0	Display interrupt	Cursor handling
10	1	Display interrupt	Page flip servicing
10	2	Display interrupt	Wait scan servicing
30	0	Illegal opcode trap	Emulation of REV instruction

The *set_interrupt* function must be called to enable or disable a particular interrupt service routine. The interrupt level and the associated priority must be specified as arguments to this function. The TIGA core functions *set_curs_state*, *page_flip*, and *wait_scan* automatically enable and disable the interrupt service routines for the cursor, page flip, and waitscan interrupts, respectively.

Interrupt service routines installed into the TIGA interrupt handler must follow a strict set of conventions and can make certain assumptions concerning the state of the TMS340 device:

- An ISR is called by the TIGA interrupt handler with no arguments.
- The ISR must terminate with a RETS 0 instruction. The TIGA interrupt handler performs the required RETI after processing all ISRs.
- The TIGA interrupt handler sets up a temporary C program stack (register A14) to enable ISRs to make calls to C functions.
- When the ISR is called, fields 0 and 1 are initialized as

  Field 0: FS0: 16 FE0: 0
  Field 1: FS1: 32 FE1: 0

  The ISR does *not* need to restore fields 0 and 1 before exiting. The TIGA interrupt handler resets them upon return from the ISR.
- All A- and B-files registers that are modified by the ISR, with the exception of A8, must be restored before exiting.
- The ISR must not re-enable interrupts via the eint instruction.

Note that it is possible for a downloaded extension to be executed from the host and, in turn, set the traps to its own service routine to avoid the overhead of the global interrupt handler in certain time-critical functions. However, care must be taken, especially in the display interrupt used by TIGA functions such as the cursor functions. If equivalent support is not given to these functions, as provided by the global interrupt handler, certain TIGA functions may not execute correctly.

Certain TMS340 boards provide external connection to the $\overline{\text{LINT}}$1 and $\overline{\text{LINT}}$2 TMS340 processor pins. In such cases, interrupt service routines can be written for them by using the techniques outlined here. However, these techniques are clearly not portable across all TMS340 processor boards.

For an example of how to create, load, and use an interrupt service routine with TIGA, refer to the files in the *\tiga\demos\isr* directory.

8.10 Object Code Compatibility

Because TIGA 2.0 encompasses both the TMS34010 and TMS34020 processors, you must keep in mind the differences between the two processors when you write extensions to TIGA. The two processors are 100% object code compatible, and all code written for the TMS34010 runs on the TMS34020. However, additional functionality has been added to the TMS34020; for this reason, take care in executing the code on that processor. A full list of compatibility guidelines are given in Appendix C of the *TMS34010 User's Guide* and in Chapter 1 of the *TMS34020 User's Guide*. Those guidelines provide a thorough list of restrictions to follow to maintain software compatibility. The following subsections are not exhaustive but provide a more detailed approach on how to deal with the differences between the two processors.

8.10.1 Determining the Processor

Because the processor checks in this section invariably involve a *two-way branch* where different actions must be performed for each of the two processors ('34010 and '34020), you must be able to distinguish them. The *device_rev* field of the CONFIG structure is provided for this purpose. The code to perform the processor check in C and in assembly language is as follows:

Example 8–4. C Code to Determine the TMS340 Processor Type

```
#include <gsptypes.h>
#include <gspglobs.h>
#include <gspreg.h>

{
    :
  if(config.device_rev & (1 << REV_34010))
  {
    /* TMS34010-specific code                              */
    :
  }
  if (config.device_rev & (1 << REV_34020))
  {
    /* TMS34020-specific code                              */
    :
  }
  :
}
```

Example 8–5. Assembly Code to Determine the TMS340 Processor Type

```
include gsptypes.inc
include gspglobs.inc
include gspreg.inc

       :
       move @_config+CONFIG_DEVICE_REV,A8,1
       btst REV_34010,A8
       jrz not_34010
is_34010:
       :                    ;TMS34010-specific code
not_34010:
       btst REV_34020,A8
       jrz not_34020
is_34020:
       :
                            ;TMS34020-specific code
not_34020:
       :
```

8.10.2 Pattern B-File Register

A simple modification must be made to the TMS34020's LINE instruction because the B13 register is used to define a 32-bit pattern with which the line is drawn. On the TMS34010, this register was not used, and, although the TMS34010's user guide states that this register should be set to all 1s to guarantee the drawing of a solid line, this was generally not done. If the routine is to draw a solid line, then the code should be like this:

```
       movi   -1,B13        ;set PATTERN register to all 1s
       move   B11,B11       ;does line point up or down?
       jrlt   down
       line   0             ;draw Bresenham line
       jruc   exit
down:
       line   1             ;draw Bresenham line
exit:
```

Note that although this is required only for the TMS34020, it does not cause a problem for the TMS34010, so, in this case, there is no need to provide a branch around the initialization of B13 for the TMS34010.

8.10.3 Pitch Registers

Another potential problem is with the SPTCH/DPTCH B-file registers and the CONVSP/CONVDP I/O registers. On the TMS34010, the CONVSP and CONVDP I/O registers contain the value of the leftmost one of the SPTCH and DPTCH registers, respectively, and these are used in the conversion of an XY address to a linear address in certain graphics instructions (such as PIXBLT XY,XY). Some of these graphics instructions, however, do not actually use the values in the SPTCH and DPTCH registers; these become, in some sense, spare registers and can be used as temporary variables. The TMS34020 be-

haves in the same way, except that the TMS34020 eliminated one of the restrictions on the TMS34010: that the pitch for XY conversion must be a power of 2. If an arbitrary pitch is to be used, CONVSP and CONVDP no longer contain the leftmost one of the SPTCH and DPTCH registers. In these cases, the I/O registers are not used, and SPTCH and DPTCH must contain the pitch. Thus, these registers are no longer spare. This means that you must be careful in using TMS34010 code written for power-of-two displays on TMS34020 boards with nonpower-of-2 displays. In this case, the SPTCH/CONVSP and DPTCH/CONVDP instruction pairs should contain corresponding values, as the *TMS34010 User's Guide* indicates (although this is sometimes ignored).

The initialization of CONVSP and CONVDP in the TMS34010 is different from that in the TMS34020. Although it is unlikely that you will need to initialize the CONVDP register, it is quite likely that you will need to initialize the CONVSP register. The initialization must be done as follows:

Example 8–6. Initialization of the CONVSP Register

```
include gsptypes.inc
include gspglobs.inc
include gspreg.inc


_set_spitch:
                         ; Assume A0 contains the desired pitch
       move A0,SPTCH     ; Initialize the SPTCH register
       move @_config+CONFIG_DEVICE_REV, A8, 1
       btst REV_34010,A8
       jrz not_34010
is_34010:              ; TMS34010 code (SPTCH must be a power of 2)
       lmo A0,A0
       setf 16,0,0
       move A0,@CONVSP, 0     ; Put lmo (SPTCH) into CONVSP
not_34010:
       btst REV_34020,A8
       jrz not_34020
is_34020:
       setcsp
not_34020:
```

8.10.4 Video Timing Registers

Most applications do not need to access the video timing I/O registers, because they are set up by TIGA at initialization time and thereafter are never accessed directly. The video timing I/O registers are useful for functions that need to synchronize themselves to the display (such as cursor handling, page flipping etc.). These functions are provided by the TIGA graphics manager directly. TIGA also provides mechanisms to allow an application to install an interrupt

service routine that is invoked whenever a particular line of the frame buffer is being displayed. Despite these provisions, there may be a reason to synchronize an application directly into the display hardware and therefore to interrogate the state of the video I/O registers.

Due to enhancements made to the video controller on the TMS34020, it was necessary to modify the addresses where the video I/O registers reside. TIGA provides global pointers that allow an application to access these registers. The global pointers are automatically initialized to either the TMS34010 or the TMS34020 processor I/O addresses, depending on which processor is installed on the board. The application should use these pointers rather than access the I/O addresses directly. The pointers and the TIGA function that initializes them are shown in the following example:

Example 8–7. Initialization of the Video Timing I/O Register Pointers

```
unsigned short *pHCOUNT, *pHEBLNK, *pHESYNC, *pHSBLNK, *pHTOTAL;
unsigned short *pVCOUNT, *pVEBLNK, *pVESYNC, *pVSBLNK, *pVTOTAL;

#include <gsptypes.h>
#include <gspglobs.h>
#include <gspreg.h>

init_ioreg_ptrs()
{
    if(config.device_rev & (1 << REV_34010))
    {
        pHCOUNT = (unsigned short *)HCOUNT10;
        pHEBLNK = (unsigned short *)HEBLNK10;
        pHESYNC = (unsigned short *)HESYNC10;
        pHSBLNK = (unsigned short *)HSBLNK10;
        pHTOTAL = (unsigned short *)HTOTAL10;
        pVCOUNT = (unsigned short *)VCOUNT10;
        pVEBLNK = (unsigned short *)VEBLNK10;
        pVESYNC = (unsigned short *)VESYNC10;
        pVSBLNK = (unsigned short *)VSBLNK10;
        pVTOTAL = (unsigned short *)VTOTAL10;
    }
    if(config.device_rev & (1 << REV_34020))
    {
        pHCOUNT = (unsigned short *)HCOUNT20;
        pHEBLNK = (unsigned short *)HEBLNK20;
        pHESYNC = (unsigned short *)HESYNC20;
        pHSBLNK = (unsigned short *)HSBLNK20;
        pHTOTAL = (unsigned short *)HTOTAL20;
        pVCOUNT = (unsigned short *)VCOUNT20;
        pVEBLNK = (unsigned short *)VEBLNK20;
        pVESYNC = (unsigned short *)VESYNC20;
        pVSBLNK = (unsigned short *)VSBLNK20;
        pVTOTAL = (unsigned short *)VTOTAL20;
    }
}
```

8.10.5 TM34020-Specific Instructions

Although all TMS34010 instructions run on the TMS34020, the TMS34020 contains new instructions that are not available on the TMS34010. In many cases, these instructions provide considerably increased performance over the TMS34010-only instructions, so it is in the application programmer's interest to detect the TMS34020 and use its instructions whenever possible. The following code uses the TMS34020's fast line instruction FLINE, which is 1 cycle per pixel faster than the regular LINE instruction:

Example 8–8. Use of TMS34020-Specific Instructions

```
include gsptypes.inc
include gspglobs.inc
include gspreg.inc

         btst REV_34020,A8
         jrz not_34020
is_34020:                  ;TMS34020-specific code
         cvdxyl  B2        ;convert to linear address
         move B11,B11      ;does line point up or down?
         jrlt fdown
         fline 0           ;draw Bresenham line
         jruc exit
fdown:
         fline 1           ;draw Bresenham line
         jruc  exit
not_34020:                 ;TMS34010-compatible code
         move B11,B11      ;does line point up or down?
         jrlt down
         line 0            ;draw Bresenham line
         jruc exit
down:
         line 1            ;draw Bresenham line
exit:
```

8.10.6 VRAM Block Mode

The TMS34020 supports two instructions (VFILL and VBLT) that use the special VRAM block mode. Certain restrictions limit the use of these instructions. These instructions can be used only if

1) The particular VRAMs used on the board support block mode (the TMS44C251s do support it)

2) DPTCH is an integral multiple of 080h

3) PSIZE is 4, 8, 16, or 32

4) Pixel processing is set to *replace*

5) Transparency is disabled

To assist the checking of these restrictions, TIGA has a *silicon_capability* field in its MODEINFO structure (a substructure of the CONFIG structure), which is a combination of the first three restrictions. If Bit 0 of this field is a 1, then the VRAMs do support block write, and DPTCH and PSIZE of the current mode do allow correct operation of the block write feature. If bit 0 of this field is a 0, then block write support is not available.

Note that this field may change from mode to mode because a board may support different pixel sizes. If block write is supported in an 8-bit-per-pixel mode, it will not be supported in a 1-bit-per-pixel mode. Also note that restrictions 4 and 5 must be checked by an application before the VFILL or VBLT instruction can be executed, but this can be done by a simple check of the CONTROL register. An example of a piece of code showing the use of the VFILL instruction is given below.

Example 8–9. Use of the VFILL Instruction

```
include gsptypes.inc
include gspglobs.inc
include gspreg.inc

       btst   REV_34020,A8                ;Suppose B–file is set up for a FILL XY
       jrne   no_vfill
is_34020:                                 ;34020–specific code
       setf 1,0,0
       move@(_config+CONFIG_SILICON_CAPABILITY), A8, 0
       jrz no_vfill                       ;Check the VRAM_BLOCK_WRITE flag set
       setf 10,0,0
       move @CONTROL+5,A8,0
       andi 3E1h,A8                       ;T and PPOP must be zero for VFILL
       jrnz no_vfill
       clip
       jrz exit
       cvdxyl B2                          ;convert to linear dest address
       vlcol                              ;load VRAM color latches
       vfill L                            ;perform linear fill
       jruc exit
no_vfill:
       fill XY                            ;fill the rectangle
exit:
```

8.11 The TIGA Linking Loader

The TIGA linking loader, TIGALNK, was the mechanism by which extensibility was made possible in TIGA versions prior to TIGA 2.0. The functions performed by TIGALNK are now included in the TIGA communication driver, so TIGALNK is useful only for debugging the installation of relocatable load modules, because it provides useful error messages. Furthermore, the *error_check* option, described in subsection 8.11.3, cannot be performed by an equivalent procedure in the CD. It can be performed only through TIGALNK.

TIGALNK is a full TMS340 linker that provides object code relocation anywhere in TMS340 memory. It is fully portable, using the TIGA communication driver to interface to any TMS340 board that has TIGA ported to it. TIGALNK has extensibility control built into it, so that it can read the TIGAEXT and TIGAISR sections and inform the graphics manager of the user extensions that are to be installed.

TIGALNK's options can be performed via equivalent functions in TIGA's communication driver. A list of the linking loader options with their procedural equivalents is given in Table 8–4.

Table 8–4. Linking Loader Options

Option	Files	Description	Equivalent Function
–ca	RLMNAME, ALMNAME	Link, then create an ALM	create_alm
–cs	COFFNAME	Create external symbol table	create_esym
–ec	RLMNAME	Check the RLM for errors	None
–fs	SYMNAME	Flush external symbol table	flush_esym
–la	ALMNAME	Load ALM into GM	install_alm
–lr	RLMNAME	Link, then load into GM	install_rlm
–lx	COFFNAME	Load and execute COFF file	load_coff / gsp_execute

The rest of this section contains a detailed description of the TIGALNK options. These options can be placed anywhere on the command line; they do not have to be placed before filename arguments.

In addition to the flags are a *–q* (quiet) option and a *–v* (verbose) option. If no options are specified, then the linker assumes normal command line operation, and all working messages and error messages are displayed normally. Selecting quiet mode operation suppresses all textual messages, and only error codes are returned upon termination. In verbose mode operation, the linker provides messages during every internal operation.

8.11.1 /ca — Create Absolute Load Module

This option creates an absolute load module (*.alm*) from the specified relocatable load module (*.rlm*). If the name of the output ALM file is not specified on the command line, then the base name of the RLM file is used, but with a forced file extension of *.alm*. Also, if no path information is supplied for the output file, then it is placed in the path specified by the *–l* option of the TIGA environment variable.

8.11.2 /cs — Create External Symbol Table

This option reads the symbolic information from the TIGA's graphics manager, *tigagm.out* and builds a new symbol table from it.

8.11.3 /ec — Error Check

This command line option can be used to check the integrity of an RLM before installing it.

Once executed, the */ec* option scans the specified RLM and prints out the number of extensions or interrupt service routines contained within the module. If none are present — that is, if no .TIGAEXT or .TIGAISR section is present— then a warning message is displayed. The amount of heap required to load the module is then displayed, and the largest available block of TMS340 heap is also displayed.

If the module contains any unresolved references that would not be resolved at loadtime, these are printed out. This allows you to resolve symbol references before actually attempting to download and install the file.

Note:

Only symbols contained in the TIGA external symbol table are used to resolve symbol references. As supplied, or after creation by the */lx* or */cs* option, this file contains only the symbols for *tigagm.out*, the TIGA core functions. If the module being checked contains references to other modules, such as the TIGA extended functions, then these must be loaded before performing the check.

Example:

```
TIGALNK /LX              ↵  – load and execute TIGAGM.OUT
TIGALNK /LR extprims     ↵  – load TIGA extended functions
                              (EXTPRIMS.RLM)
TIGALNK /EC user         ↵  – check integrity of user.rlm
```

8.11.4 /fs — Flush External Symbol Table

This option flushes all symbols from the external symbol table, except those in the TIGA graphics manager, *tigagm.out*. As the symbols for each installed module are deleted, a call to the TIGA graphics manager is made to delete the module from TMS340 memory.

8.11.5 /la — Load and Install an Absolute Load Module

This option loads and installs an ALM into the active TIGA graphics manager running on the target; this makes it possible for functions contained in the module to be invoked from the host.

Note:

ALMs contain no symbolic information, so modules loaded after an ALM cannot make references to symbols contained within an ALM.

8.11.6 /lr — Load and Install a Relocatable Load Module

This option loads and installs an RLM into the TIGA graphics manager so that functions contained in the module can be invoked from the host.

Symbols contained in the module are added to the external symbol table so that they can be referenced by modules loaded afterwards.

8.11.7 /lx — Load and Execute a COFF File / Execute TIGA GM

This option has the ability to perform two distinct functions, depending on whether or not a COFF file is specified as a command line argument. If a COFF file name is provided on the command line, then it is loaded and executed much like the stand-alone COFF loader provided with the TI software development board.

If a COFF file name is not provided, then it is assumed that the TIGA graphics manager is to be loaded and executed. In this case, two additional functions are performed after *tigagm.out* is loaded and executed. The TIGA external symbol table is created, and the symbols contained in *tigagm.out* are written to it. Once these two functions are complete, a call to the TIGA communication driver function handshake is performed to initialize communications between the host and the TMS340.

Appendix A

Data Structures

This appendix contains the data structure definitions required by TIGA applications. They are defined in the include files *typedefs.h* and *typedefs.pl*.

Section **Page**
A.1 Integral Data Types A-2
A.2 CONFIG Structure A-3
A.3 CURSOR Structure A-5
A.4 ENVIRONMENT Structure A-7
A.5 FONTINFO Structure A-8
A.6 MODEINFO Structure A-9
A.7 OFFSCREEN Structure A-13
A.8 PALET Structure A-14
A.9 PATTERN Structure A-15

The structure definitions supplied refer to the C syntax. In the assembly language equivalent file, *typedefs.inc*, the structure name precedes every field name. Thus, the *hot_x* field in the cursor structure becomes *cursor_hot_x.* This is because in the macro assembler, all fields must be unique. Note that this also applies to the TMS340-side equivalent file *gsptypes.inc*. All type definitions in this file are in upper case. The two TMS340-side type definition files, *gsptypes.h* and *gsptypes.inc,* contain additional type definitions internal to TIGA and are not generally of use to the applications programmer.

A.1 Integral Data Types

The TIGA data structures use the following type definitions throughout:

```
typedef  unsigned char     uchar;
typedef  unsigned short    ushort;
typedef  unsigned long     ulong;
typedef  unsigned long     PTR;
typedef  uchar    far      *HPTR;
```

A.2 CONFIG Structure

The CONFIG structure contains the TMS340 board and display mode configuration information. Part of this structure is the MODEINFO structure defined in Section A.6, which describes the display mode configuration. If alternate configurations are available, they can be set with *set_config*.

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

The CONFIG structure consists of the following fields:

version_number — TIGA Graphics Manager revision number, assigned by Texas Instruments. The major revision number appears in the 8 MSBs, the minor revision number in the 8 LSBs. For example, the TIGA 2.0 Graphics Manager version number assigned to the *version_number* field is 0x0200.

comm_buff_size — Size, in bytes, of each communication buffer. This field can be used by an application to determine if the data being sent to a TIGA function has the potential of overflowing the communications buffer. If so, an alternate entry point can be called.

sys_flags — This 32-bit field describes the silicon devices resident on the target TMS340-based board as follows:

Bit 0	TMS34082 #0 is present (0=no,1=yes)
Bit 1	TMS34082 #1 is present (0=no,1=yes)
Bit 2	TMS34082 #2 is present (0=no,1=yes)
Bit 3	TMS34082 #3 is present (0=no,1=yes)
Bit 4	Reserved (broadcast ID)
Bits 5–6	Reserved for future devices
Bit 7	User-defined coprocessor is present
Bits 8–31	Reserved for future use by TI

Field	Description
device_rev	This 32-bit field describes the revision of the TMS340 processor on the target board as follows: Bits 0–2 Silicon revision number Bit 3 ’34010 present (0=’34020,1=’34010) Bit 4 ’34020 present (0=’34010,1=’34020) Bits 5–15 Reserved for future generation parts Bits 16–31 Reserved for use by TI
num_modes	Total number of display modes available. The mode number argument specified to the *set_config* function must be in the range from 0 to *num_modes–1*.
current_mode	Mode number corresponding to the current display mode. This value will be in the range from 0 to *num_modes–1*.
program_mem_start	Starting linear address of the largest block of TMS340 program memory.
program_mem_end	Ending linear address of the largest block of TMS340 program memory.
display_mem_start	Starting linear address of TMS340 display memory.
display_mem_end	Ending linear address of TMS340 display memory.
stack_size	Size (in bits) of the TIGA graphics manager system stack.
shared_mem_size	Size (in bytes) of shared memory area that is available for use by a TIGA application. This field is 0 for TMS340 boards that do not support shared memory.
shared_host_addr	If shared_mem_size is nonzero, then this field contains the starting host address of the shared memory area. If *shared_mem_size* is zero, then this field is undefined.
shared_gsp_addr	If shared_mem_size is nonzero, then this field contains the starting TMS340 address of the shared memory area. If *shared_mem_size* is zero, then this field is undefined.
mode	This structure contains information pertaining to the current display mode. See the MODEINFO structure definition in Section A.6 for detailed information.

A.3 CURSOR Structure

This structure defines the cursor shape parameter for the *set_curs_shape* function.

```
typedef struct
{
  short  hot_x;
  short  hot_y;
  ushort width;
  ushort height;
  ushort pitch;
  ulong  color;
  ushort mask_rop;
  ushort shape_rop;
  ulong  mask_color;
  PTR    data;
}CURSOR;
```

This structure consists of the following fields:

hot_x — This value is added to the x coordinate of the top-left corner of the cursor shape array to position the cursor hotspot at the pixel specified by the *set_curs_xy* function.

hot_y — This value is added to the y coordinate of the top-left corner of the cursor shape array to position the cursor hotspot at the pixel specified by the *set_curs_xy* function.

width — Width (x-dimension) of the cursor shape in pixels.

height — Height (y-dimension) of the cursor shape in pixels.

pitch — Linear bit difference in the addresses of successive rows of the cursor data.

color — Cursor shape foreground color. This value is automatically replicated by the pixel size before use.

mask_rop — This field specifies the pixel-processing and transparency operations used to draw the cursor mask data to the screen as follows:

Bits	0–4	Pixel processing operating code
Bit	5	Transparency enable (0=disable,1=enable)
Bits	6–15	Reserved for future use

Consult the *set_ppop* function description on page 4-120 for further information on valid pixel processing operating codes. Also, note that the transparency mode is always set to 0 (transparency on result equal 0) for theTMS34020 device by the cursor drawing functions.

shape_rop — This field specifies the pixel-processing and transparency operations used to draw the cursor shape data to the screen. The field bit definitions are the same as those described in the *mask_rop* field above.

mask_color	Cursor mask foreground color. This value is automatically replicated by the pixel size before use.
data	Pointer to TMS340 memory containing two contiguous arrays. The dimensions of these arrays are specified by the width and height CURSOR structure fields. The bit pitch of the arrays is specified by the pitch CURSOR structure field. The first array contains the cursor mask data. The second array contains the cursor shape information.

A.4 ENVIRONMENT Structure

The ENVIRONMENT structure contains the TIGA graphics library environment global variables.

```
typedef struct
{
  ulong  xyorigin;
  ulong  pensize;
  PTR    srcbm;
  PTR    dstbm;
  ulong  stylemask;
}ENVIRONMENT;
```

The ENVIRONMENT structure consists of the following fields:

xyorigin — Current drawing origin in y::x format. The x drawing origin is contained in the 16 LSBs, and the y drawing origin is contained in the 16 MSBs. The drawing origin is modified by using the *set_draw_origin* graphics library function.

pensize — Current size of drawing pen in y::x format. The x dimension of the drawing pen is contained in the 16 LSBs, the y dimension in the 16 MSBs. The pen size is modified by using the *set_pensize* graphics library function.

srcbm — The address in TMS340 memory of the source bitmap structure. The elements of this structure are modified by using the *set_srcbm* graphics library function.

dstbm — The address in TMS340 memory of the destination bitmap structure. The elements of this structure are modified by using the *set_dstbm* graphics library function.

stylemask — Contains the current line style mask used in the *styled_line*, *styled_oval*, *styled_ovalarc*, and *styled_piearc* graphics library functions.

A.5 FONTINFO Structure

The FONTINFO structure contains parameters describing the font currently in use in TIGA. These parameters can be obtained by an application through the core function *get_fontinfo*.

```
typedef struct
{
  char   facename[30];
  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; /* Max character 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)   */
  PTR    fontptr; /* address of font in TMS340 memory       */
  short  id;      /* id of font (set at install time)       */
}FONTINFO;
```

The majority of the fields within the FONTINFO structure are identical to those defined in the FONT structure. An application uses the FONT structure to obtain font information from a font file, while it uses the FONTINFO structure to obtain font information about a loaded font. Consult Chapter 7, *Bit-Mapped Text*, for a complete description of TIGA fonts. Section 7.1, *Bit-Mapped Font Parameters* on page 7-2describes some of the parameters in the FONTINFO structure. Subsection 7.2.1, page 7-5, describes the fields of the FONT structure.

The following FONTINFO structure fields are not described in Section 7.1:

fontptr — The address, in TMS340 memory, where the font shape data is located.

id — The font identifier, returned by the function *install_font*, which is used in subsequent text functions to identify the desired font.

A.6 MODEINFO Structure

This structure contains configuration information for the current display mode. It is part of the configuration structure returned by *get_config* (which returns only the MODEINFO for the current display mode). The *get_modeinfo* function can be used to query the configuration information for any display mode supported by the TMS340 board.

```
typedef struct
{
   ulong            disp_pitch;
   ushort           disp_vres;
   ushort           disp_hres;
   short            screen_wide;
   short            screen_high
   ushort           disp_psize;
   ulong            pixel_mask;
   ushort           palet_gun_depth
   ulong            palet_size;
   short            palet_inset;
   ushort           num_pages;
   short            num_offscrn_areas;
   ulong            wksp_addr;
   ulong            wksp_pitch;
   ushort           silicon_capability;
   unsigned short   color_class;
   unsigned long    red_mask;
   unsigned long    blue_mask;
   unsigned long    green_mask;
   unsigned short   x_aspect;
   unsigned short   y_aspect;
   unsigned short   diagonal_aspect;
}MODEINFO;
```

The MODEINFO structure consists of the following fields:

disp_pitch — Linear difference (in bits) in the starting memory addresses of adjacent rows of the display memory.

disp_vres — Vertical resolution, in scan lines, of the visible portion of the screen.

disp_hres — Horizontal resolution, in pixels, of the visible portion of the screen.

screen_wide — Physical width, in millimeters, of the monitor attached to the TMS340-based board.

screen_high — Physical height, in millimeters, of the monitor attached to the TMS340-based board.

disp_psize — Pixel size, in bits. Valid pixel sizes are 1, 2, 4, 8, and 16. In addition, boards based on the TMS34020 may also support a 32-bit pixel size.

pixel_mask — Bits within a pixel with valid data.The *pixel_mask* field normally contains the value (2^*disp_psize*)–1, indicating that every bit of the pixel is pertinent. However, on some boards, the frame buffer may be arranged by 8

	bits (*disp_psize* = 8), but with only 6 bits actually implemented. In this case, *pixel_mask* would contain the value 0x3F.
palet_gun_depth	Number of bits of resolution in the digital-to-analog converter (DAC) in the display hardware. For a monochrome display, this value is one. For color displays, this value is the maximum of the number of red, blue, or green bits that are used to drive the RGB gun.
palet_size	Number of valid color values for this display mode. For a monochrome system, this value is two. For a single color index system, this value is the number of valid indices. For systems with individual red, green, and blue indices, this value is the maximum valid index for the red, blue or green index.
palet_inset	Linear offset, from the beginning of the scan line to the first valid pixel data, for display boards using the TMS34070 color palette, which stores the palette data in the frame buffer. For most systems, this field is 0.
num_pages	Number of display pages available. Some boards may support display modes with multiple pages. Multiple pages are extremely useful in animation applications. The core function, *page_flip*, can be used to flip rapidly between display pages.
num_offscrn_areas	Number of offscreen memory blocks available in this display mode. If nonzero, then information describing these offscreen areas can be obtained by calling the core function *get_offscreen_memory*.
wksp_addr	Starting linear address, in TMS340 memory, of a one-bit-per-pixel workspace, with the same dimensions as the visible screen. This field is valid only if argument *wksp_pitch* is nonzero. This workspace is not required for any of TIGA's drawing functions.
wksp_pitch	Pitch, in bits, of the workspace area. If *wksp_pitch*=0, then no workspace area is currently allocated.
silicon_capability	Silicon features available in the current display mode. This 16-bit field contains the following bit definitions: Bit 0 — VRAM block write support (0=no, 1=yes) Bits 1–15 — Reserved for future use

color_class This 16-bit field describes the color capabilities of the current display mode. Valid color classes are:

Color Class	Color Description	Palette		Color Type		Pixel Value	
		R/W	Fixed	Color	Gray	Index	RGB
0	Gray scale	√			√	√	
1	Static gray		√		√	√	
2	Pseudo color	√		√		√	
3	Static color		√	√		√	
4	Direct color	√		√			√
5	True color		√	√			√

Color capabilities are dependent on three physical attributes of the TMS340-based display board:

1) Palette type; either programmble (R/W) or nonprogrammable (fixed or no palette at all).
2) Color type; either color or gray scale.
3) Pixel-value usage; the pixel value is either a single index for RG&B or is composed of separate RGB values, which index into different RGB colormap entries.

Note that a monochrome mode is simply a gray-scale or static-gray class with a two-element colormap.

red_mask, green_mask, blue_mask
These 32-bit fields are used for the direct color and true color classes where there is a separate colormap for each primary color. Each mask defines within a pixel value those bits that index into the appropriate red, green, or blue colormap. Each mask is composed of one contiguous set of bits, with no bits in common with the other masks. These fields are zero for color classes 0–3.

x_aspect, y_aspect, diagonal_aspect
These fields specify the relative width, height, and diagonal dimensions of a screen pixel and correspond directly to the screen's aspect ratio. The *x_aspect* and *y_aspect* values are calculated so that the following is true:

(x_aspect * disp_hres) / screen_wide = (y_aspect * disp_vres) / screen_high

For example, a monitor screen with physical dimensions of 280 × 203 millimeters and a display mode of 1024 pixels × 768 lines (horizonal, vertical, respectively) would result in x, y, and diagonal aspects of 1, 0.97, and 1.39, respec-

tively. This corresponds to an aspect ratio of 100 horizontal pixels to every 97 vertical pixels. For screens whose pixels do not have integral diagonal lengths, the field values should be multiplied by a constant factor to derive integral results. Therefore, the correct values for x, y, and diagonal aspects in our previous example would be 100, 97, and 139, respectively. For numerical stability, these field values should be kept under 1000.

A.7 OFFSCREEN Structure

This structure defines the offscreen areas returned by the *get_offscreen_memory* function.

```
typedef struct
{
  PTR    addr;
  ushort xext;
  ushort yext;
}OFFSCREEN_AREA;
```

The OFFSCREEN structure consists of the following fields:

addr   Address in TMS340 memory of the offscreen area.

xext   x extension (width) of the offscreen area in pixels.

yext   y extension (height) of the offscreen area in scan lines.

A.8 PALET Structure

This structure contains the red, green, blue, and intensity components for a palette entry.

```
typedef struct
{
  uchar r;
  uchar g;
  uchar b;
  uchar i;
}PALET;
```

This structure consists of the following fields of the palette entry:

r Value of the red color component

g Value of the green color component

b Value of the blue color component

i Value of the intensity

A.9 PATTERN Structure

The PATTERN structure defines the pattern shape information passed to the *set_patn* function.

```
typedef structure
{
  ushort width;
  ushort height;
  ushort depth;
  PTR    data;
}PATTERN;
```

This structure consists of the following fields:

width	Width of the pattern in bits. Currently, only 16-bit wide patterns are supported.
height	Height of pattern in bits. Currently, only 16-bit high patterns are supported.
depth	Depth (bits/pixel) of pattern. Currently, only monochrome (1 bit-per-pixel) patterns are supported.
data	Pointer to pattern data in TMS340 memory.

Appendix B

TIGA Reserved Symbols

This appendix lists the TIGA and TMS340 reserved symbols in the following sections:

Section **Page**

B.1 Reserved Functions B-2
B.2 TIGA Core Functions Symbols B-3
B.3 TIGA Extended Graphics Library Symbols B-6

B.1 Reserved Functions

TIGA currently reserves the following functions for internal use. Do not choose function names that conflict with these. Avoid calling functions from an application program, because future versions of TIGA may not contain these functions.

add_interrupt
add_module
del_all_modules
del_interrupt
del_module
get_memseg
get_module
get_msg
get_state
get_xstate
gm_is_alive
handshake
init_cursor
init_interrupts
init_video_regs
makename
oem_init
read_hstaddr
read_hstadrh
read_hstadrl
read_hstctl
read_hstdata
rstr_commstate
save_commstate
set_memseg
set_msg
set_xstate
write_hstaddr
write_hstadrh
write_hstadrl
write_hstctl
write_hstdata

B.2 TIGA Core Functions Symbols

TIGA currently uses the following symbols in its core functions and in the TMS340 C environment. To guarantee successful operation, do not use downloadable extensions whose names conflict with any of these symbols.

Downloadable extensions used with the graphics library functions should have names that do not conflict with those in Section B.3.

IsrCStk
IsrEntryTable
IsrSrv
_CoreFunc
_CursorISR
_DEFAULT_PALET
_DefaultCursor
_DiTable
_IsrEnabled
_ModIntIoRegs
_Module
_NextDiEntry
_OutTTY
_PageFlipISR
_TrapVector
_WaitScanISR
_abort
_add_interrupt
_add_module
_ai_rev
_atexit
_c_int00
_cb_buffer
_cb_size
_check_dpyint
_clear_frame_buffer
_clear_page
_clear_screen
_comm_info
_config
_cpacket
_cpw
_csa
_curs_offset
_cvxyl
_default_setup
_del_all_modules
_del_interrupt
_del_module
_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_ai_rev
_dm_set_bcolor
_dm_set_cbbuf
_dm_set_clip_rect
_dm_set_colors
_dm_set_curs_shape
_dm_set_curs_state
_dm_set_cursattr
_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

_envtext
_envcurs
_env
_esym
_exit
_field_insert
_field_extract
_flush_extended
_flush_module
_function_implemented
_get_colors
_get_config
_get_curs_state
_get_curs_xy
_get_fontinfo
_get_isr_priorities
_get_module
_get_nearest_color
_get_offscreen_memory
_get_palet_entry
_get_palet
_get_pmask
_get_ppop
_get_state
_get_text_xy
_get_transp
_get_vector
_get_windowing
_get_wksp
_getrev
_gm_idlefunction
_gsp2gsp
_gsp_calloc
_gsp_free
_gsp_handle
_gsp_malloc
_gsp_maxheap
_gsp_minit
_gsp_realloc
_gsph_alloc
_gsph_compact
_gsph_deref
_gsph_falloc
_gsph_findhandle
_gsph_findmem
_gsph_free
_gsph_init
_gsph_maxheap
_gsph_memtype
_gsph_realloc
_gsph_sinit
_gsph_totalfree
_handleBlock
_handleAlloc
_handle
_handleGrow
_hblock_handle
_high_water_mark
_highlevel_minit
_host_command
_idlefunc_ptr
_illop
_init_cursor
_init_interrupts
_init_ioreg_ptrs
_init_palet
_init_text
_init_trap_vectors
_init_video_regs
_lastMP
_lastSeg
_linmem
_lmem
_lmo
_lowlevel_minit
_main
_makekey
_memcpy
_memmove
_modeinfo
_monitorinfo
_movmem
_mpFree
_null_patn_line
_numMP
_numSegs
_numstr

_oemdata
_oemmsg
_offscreen
_pHCOUNT
_pHEBLNK
_pHESYNC
_pHSBLNK
_pHTOTAL
_pVCOUNT
_pVEBLNK
_pVESYNC
_pVSBLNK
_pVTOTAL
_page_busy
_page
_page_flip
_palet
_palloc
_pattern
_peek_breg
_poke_breg
_printf
_release_buffer
_rmo
_set_bcolor
_set_clip_rect
_set_colors
_set_config
_set_curs_shape
_set_curs_state
_set_cursattr
_set_dpitch
_set_fcolor
_set_interrupt
_set_module_state
_set_palet
_set_palet_entry
_set_pmask
_set_ppop
_set_text_xy
_set_vector
_set_windowing
_set_wksp
_setup
_srv_ipoly
_stack_size
_strcmp
_strcpy
_strlen
_sym_chk
_sym_close
_sym_flush
_sym_getstate
_sym_get
_sym_init
_sym_open
_sym_put
_sys16
_sys_memory
_sysfont
_text_out
_text_outp
_tmphandles
_transp_off
_transp_on
_video_enable
_wait_scan
cp_call
dm_call

B.3 TIGA Extended Graphics Library Symbols

TIGA uses the following symbols in its extended graphics library functions. To guarantee successful operation, downloadable extensions with graphics library functions should not use names that conflict with any of these symbols.

_arc_draw
_arc_fill
_arc_pen
_arc_quadrant
_arc_quad
_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_ovalarc
_dm_styled_oval
_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
_in_font
_install_font
_move_pixel
_onarc
_patn_line
_patnfill_convex
_patnfill_oval
_patnfill_piearc
_patnfill_polygon
_patnfill_rect
_patnframe_oval
_patnframe_rect
_patnpen_line
_patnpen_ovalarc
_patnpen_piearc
_patnpen_point
_patnpen_polyline
_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

Appendix C

Debugger Support for TIGA

TIGA is the definitive interface standard for applications software written to run on the TMS340 architecture. However, it gives no guidelines to debugger developers with special hardware accessing requirements.

A set of routines has been included in TIGA to meet the often unique needs of debugger developers. This appendix contains those routines.

Section **Page**

C.1 Debugger Routines C-2
C.2 TIGA / Debugger Interface C-12
C.3 Compatibility Functions C-14

C.1 Debugger Functions

A separate document describing the use of the debugger functions will be published in the future. Development of debugger functions will be based on user feedback received and on the following criteria:

- Portability to any TIGA environment, which potentially includes all TMS34010- and TMS34020-based PC graphics displays.
- Transparency to share the TMS34010's host interface registers with an application that is being debugged and that uses the host interface for communication between host and TMS340-resident software.
- Ability to support the symbolic debugging of RLMs (Relocatable Load Modules).

The following is a list of the routines in TIGA that provide debugger support:

get_vector	Return contents of TMS340 trap vector
set_vector	Set contents of TMS340 trap vector
get_xstate	Return TMS340 execution state
set_xstate	Set TMS340 execution state
get_memseg	Return high/low bounds of TMS340 memory segment
set_memseg	Set high/low bounds of TMS340 memory segment
get_msg	Receive a message from the TMS340
set_msg	Send a message to the TMS340
save_commstate	Save communication state
rstr_commstate	Restore communication state
oem_init	Initialize board-specific data

Note:

The *get_vector* and *set_vector* functions are described in Chapter 4 because their usefulness is not restricted to debuggers.

Syntax

```
void get_memseg(addrlo, addrhi);
      unsigned long *addrlo, *addrhi;
```

Type

Host-only

Description

The *get_memseg* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function returns the low and high bit addresses of a usable block of TMS340 memory. Note that if the TIGA graphics manager is active (determined by a call to *gm_is_alive*), then this block of memory has been appropriated by the TIGA memory manager and should not be used. Instead, a call to TIGA should allocate usable memory. The two arguments, *addrlo* and *addrhi*, are pointers to unsigned longs where the TMS340 addresses are to be placed.

Syntax unsigned short get_msg();

Type Host-only

Description The *get_msg* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements.This function receives a 3-bit message from the TMS340. The message is located in bits 0–2 of the returned value. The fourth bit, bit 3, is an interrupt bit and indicates that an interrupt was requested by the host.

Syntax unsigned short get_xstate();

Type Host-only

Description The *get_xstate* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function returns the current TMS340 execution state. The returned 16 bits are described below:

Bit 0	1 if TMS340 is halted, 0 if not
Bit 1	1 if NMI set, 0 if not
Bit 2	1 if NMIMODE set, 0 if not
Bit 3	1 if cache flushed, 0 if not
Bit 4	1 if cache disabled, 0 if not
Bit 5	1 if host interrupt (the INTIN bit of the host control register) is set, 0 if it is cleared.
Bits 6–15	Reserved for future use

Example

```
#include <tiga.h>
main()
{
  if (tiga_set(CD_OPEN) >= 0)
  {
    if (get_xstate() & 1)
      printf(”TMS340 is halted\n”);
    else
      printf(”TMS340 is running\n”);
    tiga_set(CD_CLOSE);
  }
}
```

Syntax

```
unsigned long oem_init(gm_size)
    unsigned long gm_size;
```

Type

Host-only

Description

The *oem_init* function halts the TMS340 and performs any board-specific initialization required before loading a COFF file. It also performs TMS340 heap memory initialization using the size of the TIGA graphics manager (GM) in its calculations. This size, in bytes, is specified by the argument *gm_size.*

The *oem_init* function returns a TMS340 address corresponding to the start of the TIGA GM. This address is not the execution entry point of the TIGA GM, but rather the relocation address of the TIGA GM COFF file.

Syntax unsigned short rstr_commstate();

Type Host-only

Description The *rstr_commstate* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function restores the state of TMS340 communications to the state it was in after a previous call to *save_commstate*.The function returns zero if it is unable to save the state, nonzero if it is successful.

Syntax unsigned short save_commstate();

Type Host-only

Description The *save_commstate* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function saves the state of TMS340 communications. The function returns zero if it is unable to save the state, nonzero if it is successful.

Syntax

```
void set_memseg(addrlo, addhi);
      unsigned long addrlo, addrhi;
```

Type

Host-only

Description

The *set_memseg* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function sets the low and high bit addresses of a usable block of TMS340 memory. It should be called to reflect the new memory size after some of the memory returned by *get_memseg* is used.

Syntax

```
void set_msg( msg );
    unsigned short msg;
```

Type

Host-only

Description

The *set_msg* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function sends a 3-bit message to the TMS340. The message is located in bits 0–2 of argument *msg*. The fourth bit, bit 3, is an interrupt bit and requests a host interrupt into the TMS340.

Syntax

void set_xstate(options);
    unsigned short options;

Type

Host-only

Description

The *set_xstate* function is not for general use. It is intended for debuggers and other such tools that have special hardware accessing requirements. This function sets the current TMS340 execution state. The returned 16 bits are described below:

Bit 0	1 to halt the TMS340, 0 to let it run
Bit 1	1 to invoke an NMI, 0 to clear NMI
Bit 2	1 to set NMIMODE, 0 to clear NMIMODE
Bit 3	1 to flush cache, 0 to stop cache flush
Bit 4	1 to disable cache, 0 to enable cache
Bit 5	1 to set the host interrupt (the INTIN bit of the host control register); 0 does nothing because the bit can be cleared only by the TMS340.
Bits 6–15	Reserved for future use; must be set to 0s

Example

```
#include <tiga.h>

main()
{
  if (tiga_set(CD_OPEN) >=0)
  {
    set_xstate(1);  /* halt the TMS340                          */
    set_xstate(0);  /* run the TMS340                           */
    tiga_set(CD_CLOSE);
  }
}
```

C.2 TIGA / Debugger Interface

TIGA 2.2 provides the capability for symbolic debugging of relocatable load modules (RLMs) by providing RLM relocation and filename information to a TMS340-resident debugger monitor via the TRAP 29 software interface. This capability is enabled by specifying the *–d2* option when loading the TIGA communication driver (*tigacd.exe*).

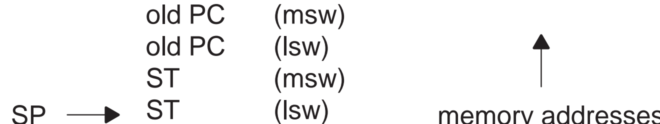
For a debugger to take advantage of this capability, it must provide and install an interrupt service routine (ISR) into the trap 29 vector. The following three services must be provided by the debugger ISR to fully support TIGA's debug capability:

1) Load symbols into the debugger that correspond to an RLM or the TIGA graphics manager.

2) Flush symbols that correspond to a previously loaded RLM.

3) Flush all symbols that correspond to all previously loaded RLMs.

Whenever an application loads or flushes an RLM, TIGA first services the request and then passes the request onto the debugger ISR, via a trap 29 software interrupt, for processing. When control is passed to the debugger's ISR, the stack is set up as follows:



Information regarding the request is stored relative to the old PC on the stack. Once the ISR has fetched the value of the old PC (for example, in Addr), the request information can then be read relative to Addr as follows:

1) Module Load (ML) request

```
Addr:                    RETS instruction    (0x0960)
Addr+0x10:               Magic number        (0x0468)
Addr+0x20                Request type        (0x4D4C, ASCII for ML)
Addr+0x30:               Module ID
Addr+0x40:               Number of sections  (starts with section 0)
Addr+0x50:               LSW of section 0 relocation
Addr+0x60:               MSW of section 0 relocation
        :                         :
Addr+0x50+(n–1)*0x20:    LSW of section n–1 relocation
Addr+0x50+(n–1)*0x20:    MSW of section n–1 relocation
Addr+0x50+n*0x20:        Filename[0]
Addr+0x58+n*0x20:        Filename[1]
        :                         :
```

Note:

Filename is a null-terminated, fully qualified (drive and directory included) string.

2) Module Flush (MF) request

Addr:	RETS instruction	(0x0960)
Addr+0x10:	Magic number	(0x0468)
Addr+0x20:	Request type	(0x4D46, ASCII for MF)
Addr+0x30:	Module ID	

3) Flush all Modules (MA) request

Addr:	RETS instruction	(0x0960)
Addr+0x10:	Magic number	(0x0468)
Addr+0x20:	Request type	(0x4D41, ASCII for MA)

After servicing the request, the debugger ISR should call the RETI instruction to properly terminate the interrupt.

C.3 Compatibility Functions

It is recommended that you *do not* use the compatibility functions. Their operations functions can be performed by the entry points in the previous section and by the communication functions described in Chapter 4. These functions talk directly to TMS34010 hardware, which is not present on the TMS34020. Their functionality can be emulated only on the TMS34020. However, since the TMS34010 has been available for some years now, many utilities have been written that interface to the TMS34010 hardware directly. If these utilities are to be ported to TIGA with the understanding that they may not run correctly on the TMS34020 or other future products, then these functions may provide a quick method of porting.

read_hstaddr	Read the TMS34010 host address register
read_hstadrh	Read the TMS34010 host address high register
read_hstadrl	Read the TMS34010 host address low register
read_hstctl	Read the TMS34010 host control register
read_hstdata	Read the TMS34010 host data register
write_hstaddr	Write to the TMS34010 host address register
write_hstadrh	Write to the TMS34010 host address high register
write_hstadrl	Write to the TMS34010 host address low register
write_hstctl	Write to the TMS34010 host control register
write_hstdata	Write to the TMS34010 host data register

Syntax unsigned long read_hstaddr();

Type Host-only

Description The *read_hstaddr* function returns the contents of the host address register of the TMS34010.

Syntax unsigned short read_hstadrh();

Type Host-only

Description The *read_hstadrh* function returns the contents of the host address high register of the TMS34010.

Syntax unsigned short read_hstadrl();

Type Host-only

Description The *read_hstadrl* function returns the contents of the host address low register of the TMS34010.

Syntax unsigned short read_hstctl();

Type Host-only

Description The *read_hstctl* function returns the contents of the host control register of the TMS34010.

Syntax unsigned short read_hstdata();

Type Host-only

Description The *read_hstdata* function returns the contents of the host data register of the TMS34010.

Syntax

```
void write_hstaddr(value)
       unsigned long value;
```

Type

Host-only

Description

The *write_hstaddr* function writes the supplied 32-bit value into the host address register of the TMS34010.

Syntax

```
void write_hstadrh(value)
      unsigned short value;
```

Type

Host-only

Description

The *write_hstadrh* function writes the supplied 16-bit value into the host address high register of the TMS34010.

Syntax

```
void write_hstadrl(value)
      unsigned short value;
```

Type

Host-only

Description

The *write_hstadrl* function writes the supplied 16-bit value into the host address low register of the TMS34010.

Syntax

```
void write_hstctl(value)
    unsigned short value;
```

Type

Host-only

Description

The *write_hstctl* function writes the supplied 16-bit value into the host control register of the TMS34010. Note that for TIGA to function properly, the values of the INCR, INCW, and LBL bits in host control must be set in a particular manner. If these bits are modified, they *must* be restored before invoking another TIGA function, or else the TIGA environment may be corrupted.

Syntax

```
void write_hstdata(value)
    unsigned short value;
```

Type

Host-only

Description

The *write_hstdata* function writes the supplied 16-bit value into the host data register of the TMS34010.

Appendix D

Error Messages / Error Codes

Error messages are returned by the standard TIGA error handler if problems are encountered while a TIGA driver or application is running. The problems associated with these errors are likely caused by the application or by the driver itself.

Error codes are returned when problems occur during loading of the TIGA Communication Driver (CD), the TIGA Graphics Manager (GM), or TIGA extensions contained in ALM or RLM files. These problems may be caused by the TIGA application or by the driver but are more apt to be the result of improper installation of TIGA or the TMS340 board. The following TIGA functions and utility programs, which provide the CD, GM, RLM and ALM load support in TIGA, are the most likely to generate these error codes:

tigalnk.exe	TIGA linking loader utility
create_alm()	Create alm file
create_esym()	Create external symbol file
flush_esym()	Flush external symbols from symbol file
install_alm()	Load TIGA extensions from ALM file
install_primitives()	Load TIGA graphics library functions
install_rlm()	Load TIGA extensions from RLM file
sym_flush()	Flush relocatable load module symbols

Each description of error message / error code contains the following information:

Error Code The status code returned to signify that an error has occurred

Error Msg. The status message returned to signify that an error has occurred

Error Type The general type of error that has occurred

Cause(s) A list of possible causes that prompted the error

Fix(es) Possible solutions to fix the cause of the error

This appendix contains a list of error messages and error codes returned by TIGA, in the following sections:

Section **Page**
D.1 Error Messages D-2
D.2 Error Codes D-3
D.3 Communication Driver (CD) Errors D-7

D.1 Error Messages

Error Msg.	**Timeout waiting for a free command buffer**
Error Type	TIGA extended function error
Cause(s)	TIGA graphics manager has been corrupted. A called TIGA extended function did not execute to completion
Fix(es)	Check TIGA function source for problems. Also, check host entry point invocation to ensure that arguments are being passed correctly

Error Msg.	**Timeout waiting for current TIGA command**
Error Type	TIGA extended function error
Cause(s)	TIGA graphics manager has been corrupted. A called TIGA extended function did not execute to completion
Fix(es)	Check TIGA function source for problems. Also, check host entry point invocation to ensure that arguments are being passed correctly

Error Msg.	**Not enough memory to store parameters**
Error Type	TIGA function calling error
Cause(s)	An alternate (_*a*) entry point was called, but not enough TMS340 memory was available to allocate temporary command buffer
Fix(es)	Reduce the amount of data being sent to the function

D.2 Error Codes

Error Code	**–1**
Error Type	TIGA system error (TIGA 1.1 and lower only)
Cause(s)	The file *tigalnk.exe* could not be found
Fix(es)	1) Ensure that the *–m* option of the TIGA environment variable is set to your main TIGA directory 2) If so, verify that the file *tigalnk.exe* is contained in the directory specified by the *–m* option

Error Code	**–2**
Error Type	TIGA communication driver (CD) error
Cause(s)	Invalid command line arguments specified for *tigacd.exe*
Fix(es)	Ensure that the command line arguments are specified when running *tigacd.exe* are valid. Check Section 2.6 for valid options

Error Code	**–3**
Error Type	TIGA memory error
Cause(s)	1) Not enough host memory to invoke TIGALNK (TIGA versions 1.1 and lower only) or 2) Not enough TMS340 memory available to load specified module
Fix(es)	1) Free memory on host by eliminating unused TSR's 2) Ensure that your TMS340 board has enough memory to run the application. If not, add additional memory to TMS340 board, if possible

Error Code	**–4**
Error Type	TIGA communication driver (CD) error
Cause(s)	TIGA communication driver (*tigacd.exe*) not running
Fix(es)	Run *tigacd.exe* to install the TIGA communications driver before running a TIGA application or driver

Error Code	**–5**
Error Type	TIGA graphics manager (GM) error
Cause(s)	1)TIGA graphics manager (*tigagm.out*) load error or 2) TMS340 board failure
Fix(es)	1) Ensure that the file *tigagm.out* exists in the directory specified by the *–m* option of the TIGA environment variable 2) Run the manufacturer's supplied diagnostics to ensure your TMS340 board is operating properly

Error Code **–6**

Error Type RLM load error

Cause(s)
1) RLM does not exist in the directory specified in the application or
2) RLM does not exist in the directory specified by the *–l* option of the TIGA environment variable

Fix(es)
1, 2) Ensure that the RLM file exists in either the directory specified by the application or in the directory specified by the *–l* option of the TIGA environment variable
3) Obtain a new copy of the RLM file

Error Code **–7**

Error Type Symbol file error (TIGA versions 1.1 and lower only)

Cause(s)
1) TIGA could not find the file *tiga340.sym*
2) The *tiga340.sym* file is corrupt

Fix(es)
1) Ensure that the *–m* option of the TIGA environment variable is set to your main TIGA directory. Also, make sure that the file *tiga340.sym* exists in this directory
2) Run *tigalnk /ca* to create a new *tiga340.sym* file

Error Code **–8**

Error Type ALM load error

Cause(s)
1) ALM does not exist in directory specified in the application
2) ALM does not exist in directory specified by the *–l* option of the TIGA environment variable
3) ALM file is corrupt

Fix(es)
1, 2) Ensure that the ALM file exists in either the directory specified by the application or in the directory specified by the *–l* option of the TIGA environment variable
3) Obtain a new copy of the ALM file

Error Code **–9**

Error Type COFF (*.out*) file load error

Cause(s) TIGA could not locate specified COFF *.out* file

Fix(es) Ensure that the specified COFF*.out* file exists in the directory expected by the application

Error Code	**–10**
Error Type	Symbol reference error
Cause(s)	An unresolved symbol was encountered when the specified RLM or ALM file was loading
Fix(es)	1) Run *tigalnk /lx* to reinitialize external symbol table 2) Run *tigalnk /ec filename* to determine undefined symbols

Error Code	**–11**
Error Type	Symbol file creation error (TIGA versions1.1 and lower only)
Cause(s)	The COFF file specified in *tigalnk /cs filename* is not linked at an absolute address
Fix(es)	Relink the specified COFF file without the –ar or –r linker option

Error Code	**–12**
Error Type	Symbol file error
Cause(s)	The modules installed in the symbol table do not match those the TIGA graphics manager has installed
Fix(es)	Run *tigalnk –fs* to flush the symbols. Then, rerun your TIGA application

Error Code	**–13**
Error Type	Handshake error
Cause(s)	1) TIGA graphics manager (*tigagm.out*) load error 2) Wrong *tigagm.out* being used 3) TMS340 board failure
Fix(es)	1, 2) Ensure that the correct file *tigagm.out* exists in the directory specified by the *–m* option of the TIGA environment variable 3) Run the manufacturer's supplied diagnostics to make sure your TMS340 board is operating properly.

Error Code	**–14**
Error Type	COFF load error (TIGA versions 2.0 and higher only)
Cause(s)	An error was encountered while attempting to load a COFF file
Fix(es)	Recreate the COFF file and try again

Error Code	**–15**
Error Type	Symbol table error (TIGA versions 2.0 and higher only)
Cause(s)	Not enough TMS340 memory available to store the symbols associated with an RLM
Fix(es)	1) Use the *%f* option of the *install_rlm* function to disable loading of symbols 2) Reduce the number and name length of symbols in RLM 3) Add more memory to the TMS340 board, if possible

Error Code	**–25**
Error Type	TMS340 board error (TIGA versions 2.0 and higher only)
Cause(s)	Communications error with target TMS340 board
Fix(es)	1) Ensure that you installed the TMS340 board properly 2) Run the manufacturer's supplied diagnostics to ensure your TMS340 board is operating properly

Error Code	**–26**
Error Type	TIGA communication driver error (TIGA versions 2.0 and higher only)
Cause(s)	The TIGA communications driver was already closed when the TIGA function *tiga_set(CD_CLOSE)* was issued
Fix(es)	No fix required

D.3 Communication Driver (CD) Errors

Error Msg. **TIGA Driver already installed in interrupt level 0x*??***

Cause(s) An attempt was made to load the TIGA CD at the current TIGA interrupt level when it is already installed.

Fix(es) Do not reload the TIGA CD once it is installed. The TIGA CD remains memory resident until:

1) The PC is rebooted or powered off, or
2) The TIGA CD is uninstalled (using the command *tigacd –u* from the DOS command line.

Error Msg. **NonTIGA Driver installed in interrupt level 0x*??***

Cause(s) An attempt was made to load the TIGA CD at an interrupt level currently in use by another driver.

Fix(es) Load the TIGA CD at a different interrupt level. To change the TIGA interrupt level, modify the *–i* option of the TIGA environment variable. See Section 2.5.

Error Msg. **Abort (No response from display board)**

Cause(s) Initialization of the TIGA display board failed. Possible causes are:

1) The TIGA display board is not installed or is defective.
2) The TIGA display board is not is not set up properly.
3) The wrong TIGA CD is being used.
4) The wrong TIGA configuration file (*tiga.cfg*) is being used.

Fix(es) To fix the causes lilsted above:

1) Ensure that the TIGA display board is installed properly in your PC.
   - Is the board connector fully seated into the adapter slot?
   - Is the board installed in the correct slot? A board with a 16-bit connector should be installed in a 16-bit slot.
   - Are all socketed components fully seated into their sockets?
   - Run any diagnostics provided with the display board to determine if the board is damaged.
2) Rerun the TIGA board setup program that came with the board (if supplied). Ensure that all I/O addresses and memory addresses required during setup correspond to the actual configuration of your diplay board. You may want to try various combinations of options presented during setup to see if any of them work with your display board.
3) Ensure that the TIGA CD you are using is intended for your display board. If in doubt, refer to your board's software installation manual.

- Ensure that the file *tiga.cfg* in use is intended for your display board. If in doubt, refer to your board's software installation manual.

Error Msg. TIGA driver is not installed!

Cause(s) An attempt was made to unload the TIGA CD from memory (using the command *tigacd –u* from the DOS command line), but it failed because no interrupt vector is present at the current TIGA interrupt level.

Fix(es)
1) Do not attempt to unload the TIGA CD when it is not installed.
2) Do not modify the TIGA interrupt level (by changing the *–i* option of the TIGA environment variable) after loading the TIGA CD.

Error Msg. **Attempt to unload TIGA driver failed!**

Cause(s) An attempt was made to unload the TIGA CD from memory (using the command *tigacd –u* from the DOS command line), but it failed because the interrupt vector present at the current TIGA interrupt level does not belong to TIGA.

Fix(es)
1) Do not modify the TIGA interrupt level (by changing the *–i* option of the TIGA environment variable) after loading the TIGA CD.
2) Ensure that no other driver is using the same interrupt level as TIGA.

Error Msg. **Required file TIGA.CFG not found**

Cause(s) The file *tiga.cfg* could not be found in the directory specified by the *–m* option of the TIGA environment variable.

Fix(es)
1) Ensure that the *–m* option of the TIGA environment is set to the main TIGA directory. See Section 2.5 for more information.
2) Ensure that the file *tiga.cfg* exists in the directory specified by the *–m* option of the TIGA envronment variable
3) Ensure that the hidden or read-only file attributes are not set for *tiga.cfg*.
4) Ensure that the file *tiga.cfg* is not damaged. If in doubt, refer to your board's software installation manual.

Appendix E

Glossary

A_DIR: An MS-DOS environment variable; identifies the directories searched when you specify include and macro files for the TMS340 family assembler.

AI: Application interface. A part of TIGA consisting of a linkable application library and include files that reference TIGA type and function definitions. The AI provides the interface between an application and the TIGA communication driver (CD).

ALM: Absolute load module. An extension to the TIGA standard in the form of TMS340 object code. It is linked to an absolute memory location and stored in a memory image format. An application can load the ALM to invoke custom TMS340 functions.

B

bitblt: Bit-aligned block transfer. Transfer of a rectangular array of pixel information from one location in a bitmap to another with the potential of applying 1 of 16 logical operators during the transfer.

bitmap: 1. The digital representation of an image in which bits are mapped to pixels. 2. A block of memory used to hold raster images in a device-specific format.

C

CD: Communication driver. This is a terminate-and-stay-resident program that runs on the PC. It is specific to a particular board and is supplied by the board manufacturer with the board.The CD contains functions that are invoked by an application's calls to the AI to communicate via the PC bus to the target TMS340 board.

C_DIR: An MS-DOS environment variable; it identifies the directories searched when you specify include files for the TMS340 C compiler and object directories for the TMS340 linker.

COFF: Common object file format. An implementation of the object file format of the same name developed by AT&T. The TMS340 family compiler, assembler, and linker use and produce COFF files.

command buffer: An area of TMS340 memory used by the TIGA interface buffer data passed by the application and read by the TMS340 processor.

command number: An identifier of a function to be invoked by an application when the function resides on the TMS340 board. The command number consists of three parts: 1) The function type, which specifies the format in which the parameters are referenced by the TMS340. 2) The module number, which acts as an identifier to the group of functions. Every DLM receives a module number when it is installed . 3) The function number, which specifies the specific function within the DLM that is to be invoked.

communication buffer: See command buffer.

configuration: The hardware setting of the TMS340 board, comprising display resolution, pixel size, palette size, availability of shared memory, etc.

coprocessor: Microdevice that offloads numeric operations from the main processor to speed up overall operation. The TMS34082 is a coprocessor to the TMS34020. The two devices are tightly coupled together. The coprocessor adds to the register and instruction capability of the TMS34020, resulting in improved handling of floating-point arithmetic. In this manual, the TMS340 processors are occasionally defined as coprocessors to the 80x86 PC processor. This is to emphasize that the TMS340 is a programmable processor and can offload much of the burden of the graphics processing and bitmap manipulation from the host PC.

core functions: A group of TIGA functions that can *always* be invoked by an application after TIGA has been installed, as opposed to the extended functions that need to be loaded explicitly by an application.

C-packet mode: A method of passing parameters in TIGA from the host to a function on the TMS340 board. It enables the parameters pushed onto the host stack to appear on the TMS340 program stack, as if the function had been invoked locally to the TMS340.

cursor: In TIGA, an icon on the screen. The cursor is generally under mouse control and is used as a pointing device in a graphics application.

DDK: Driver developer's kit. A product provided by Texas Instruments to allow software developers to write application drivers that interface to the TIGA-340 standard.

direct mode: A TIGA mode that is the fastest mechanism to transfer parameters from the host to a function on the TMS340 board. The parameter data is passed in raw form to a TIGA communication buffer, and the TMS340 function receives a pointer to the data.

DLM: Dynamic load module. The DLM is a key part of TIGA's extensibility. The module consists of a collection of custom C or assembly routines that are not otherwise part of TIGA; thus, they are an extension of TIGA's functionality. The DLM is loaded by an application so that the custom TMS340 functions can be invoked by the application. There are two types of modules: relocatable load modules (RLMs) and absolute load modules (ALMs).

environment or **drawing environment:** A group of attributes consisting of drawing origin, pen size, fill pattern, source and destination bitmaps, and line style.

environment variable: An MS-DOS variable that can have a string assigned by an end-user with the MS-DOS *SET* command. This string can be interrogated by a program running under MS-DOS.

extended functions: A portion of the TIGA interface functions that can be invoked ***only*** by a TIGA application after they have been explicitly installed.They consist mostly of drawing functions.

extensibility: A key feature of TIGA that consists of an expandable function set. An application programmer can write custom TMS340 functions, which can be installed at runtime and invoked from the host application in the same manner as the standard TIGA functions.

font: A set of characters in predefined format that contain alignment information, allowing the text routines to produce a visually correct representation of a given character string.

frame buffer: A portion of memory used to buffer rasterized data to be output to a CRT display monitor. The contents of the frame buffer are often referred to as the bitmap of the display and contain the logical pixels corresponding to the points on the monitor screen.

GM: Graphics manager. A TMS340 object file specific to a particular board, supplied with the board by the manufacturer. It contains a command executive to process commands sent from the application, and a set of functions. Some of these are integral TIGA functions, and some may be user extensions.

GSP: Graphics system processor. A TMS340 family-based system with the processing power and control capabilities necessary to manage high-performance bitmapped graphics.

heap: An area of memory that a program can allocate dynamically.

ISR: Interrupt service routine. A routine to service an interrupt on the TMS340 processor. The most common interrupt is that produced by the display interrupt, but other interrupts are available from the host processor and from two external interrupt pins for window violation. ISRs can be downloaded by an application as part of a DLM .

ISV: Independent software vendor. A company that produces software products. In this guide, ISV refers to a company that writes a software product to interface directly with TIGA. ISVs include Microsoft, Autodesk, etc.

linking loader: A program called TIGALNK that runs under MS-DOS and is an integral part of TIGA. It loads and links a dynamic load module with user extensions to TIGA into the TIGA Graphics Manager on the TMS340.

memory management: Also referred to as dynamic memory allocation. It consists of a group of functions that are used for heap management.

mode: A particular configuration of a board. An individual board may have several modes, for example: 1024 pixels × 768 lines at 8 bits per pixel, or 640 pixels × 480 lines at 4 bits per pixel.

MS-DOS : Microsoft disk operating system. A PC-based operating system. Because MS-DOS and PC-DOS are essentially the same operating system, this manual uses the term MS-DOS to refer to both systems.

N

NMI: Nonmaskable Interrupt.The NMI cannot be disabled; it is usually generated by a host processor.

O

OEM: Original equipment manufacturer. A hardware manufacturing company. In this user's guide, it refers to companies that manufacture PC graphics add-in boards with a TMS340 processor on them.

offscreen memory: The part of the frame buffer not being output to a display. A frame buffer, although typically one contiguous area of linear memory, can be viewed as a rectangular area with a specific pitch. Each row of the rectangular area corresponds to a row of pixels on the screen. If the length is less than the frame buffer pitch, or if there are more lines in the frame buffer than are displayed on the screen, there will be an area of the frame buffer not used for display. This area is named offscreen memory. Offscreen memory does not include the program memory used to store code and data.

origin: The zero intersection of X and Y axes from which all points are calculated.

P

page: Some TMS340 boards may have enough memory in their frame buffer to hold two complete copies of the bitmap output to the screen. This technique, sometimes called double buffering, allows one area of the screen to be displayed (the display page) while another is being updated (the drawing page). When the drawing is completed, the drawing and display pages are interchanged (page flipping). The flip is synchronized to the vertical blank time to ensure a flicker-free display. This technique is useful for producing animation sequences.

palette: A digital-lookup table used in a computer graphics display for translating data from the bitmap into the pixel values shown on the display.

pattern or **fill pattern:** A design that some TIGA graphics output functions use to fill an area with a pattern rather than a solid color. The pattern is specified as a 1-bit-per-pixel map. When the pattern is drawn, 0s in the bitmap are drawn in the current background color, and 1s are drawn in the current foreground color.

pen or **drawing pen:** A software drafting tool that some TIGA graphics functions use to draw a pixel outline on the screen. The application can select the width and the height of the pen. The area covered by the pen can be solid or pattern-filled.

pixel processing: A logical or arithmetic combination of two pixel values (source and destination).

PixBlt: Pixelblock transfer. Operations on arrays of pixels in which each pixel is represented by one or more bits. PixBlt operations are a superset of bitblt operations and include not only commonly used logical operations, but also integer arithmetic and other multibit operations.

plane: (Also bit plane or color plane). A bitmap layer in a display device with multiple bits per pixel. If the pixel size is n bits and the bits in each pixel are numbered 0 to *n*–1, plane 0 is made up of bits 0 from all the pixels, and plane *n*–1 is made up of bits numbered *n*–1 from all the pixels. A layered graphics display allows planes or groups of planes to be manipulated independently of the other planes.

R

raster-op: Raster operation. See pixel processing.

RLM: Relocatable load module. An extension to the TIGA standard in the form of TMS340 object code. The RLM file is in COFF file format. It is loaded by an application so that the application can invoke custom TMS340 functions.

S

SDB: Software development board. A PC-compatible board manufactured by Texas Instruments. Two SDBs are produced by Texas Instruments: a TMS34010-based board and a TMS34020-based board.

SDK: Software developer's kit. A Texas Instruments product that allows software developers to write TMS340 code. The SDK may be used to develop a TIGA extension, but it is equally applicable for programmers who wish to develop stand-alone TMS340 applications.

shift-register transfer: A transfer between RAM storage and the internal shift register in a video RAM.

SPK: Software porting kit. A Texas Instruments product that allows manufacturers of TMS340-based boards to port TIGA to their board. It contains all TIGA software source code.

swizzle: The reversal of every bit in a byte. This is required to convert from big-endian processors (where the smallest numbered bit in a word is most significant) to little-endian processors (where the smallest numbered bit in a word is least significant).

symbol table: A file containing the symbol names of all the variables and functions on the TMS340 side of TIGA. The symbol table is used by the linking loader when it is downloading an RLM to resolve references to those symbols. This enables the functions in the RLM to call TIGA functions that are resident on the TMS340 board.

T

TDB: TIGA development board. A TMS34010-based graphics board available from Texas Instruments.

TIGA: Texas Instruments Graphics Architecture. A software interface standard that allows a host processor to communicate with the TMS340 graphics processors that are typically resident on an add-in board. The current implementation of TIGA is for the PC market and interfaces the 80x86 processor running under MS-DOS with the TMS340.

TIGACD: The filename of the executable program containing the communication driver that you run to install TIGA on your system.

TIGALNK: See linking loader.

time-out: The time allowed for a command to complete. An application invokes a TIGA TMS340 function by placing a command number and appropriate parameters in one of several command buffers. After loading several commands, the command buffers may be full; the host must wait until the TMS340 finishes the current command and frees up a buffer. Also, if the function invoked needs to return data back to the application, the application must wait until the TMS340 completes the command. If the application waits longer than a specified time, a time-out warning message is displayed.

TMS340: A family of graphics system processors and peripherals manufactured by Texas Instruments.

TMS34010: First-generation graphics processor manufactured by Texas Instruments.

TMS34020: Second-generation graphics processor manufactured by Texas Instruments.

TMS34070: First-generation color palette manufactured by Texas Instruments.

TMS34082: Floating-point unit manufactured by Texas Instruments; coprocessor to the TMS34020.

transparency: The attribute of effective invisibility in a pixel. When a transparent pixel is written to the screen, it does not alter that portion of the screen it is written to. For example, in a pixel array containing the pattern for the letter *A,* all pixels surrounding the *A* pattern could be given a special value indicating that they are transparent. When the array is written to the screen, the *A* pattern, but not the pixels in the rectangle containing it, would be invisible.

trap vector: A specific 32-bit address in TMS340 memory that contains the address of an interrupt service routine.

TSR: Terminate and stay resident. A type of program that runs under MS-DOS. When it terminates, this type of program leaves a portion of itself in memory.

window: A specified rectangular area of virtual space on the display.

workspace: An area of memory that is equal in size to a 1-bit-per-pixel representation of the current display resolution. Polygon fill functions use the workspace as a temporary drawing area before drawing on the screen. The workspace can reside either in offscreen memory or in heap.

Index

A

absolute load module, ALM, 4-14, 4-81, 8-3, 8-7, 8-45, 8-47
  installation, 8-7
addr, A-13
AI libraries, development tools, 3-4
application interface, AI, 1-3, 2-5, 2-10
application program, 4-29, 4-32, 4-89, 5-84
area-fill, pattern, 4-101, 5-56, 5-57, 5-59, 5-61, 5-62, 5-75, 5-76, 5-82, 5-101, 6-12
ascent, 7-2
attributes, 8-29
autoexec modification, 2-7
aux_command, 3-10

B

b, A-14
back-face test, 5-25
background color, 8-29
base line, 7-2
bitblt, 3-14
block font, 4-140, 5-86
Bresenham's algorithm, 5-11

C

C-packet, 8-9, 8-10, 8-12, 8-16
callback functions, 8-31
cd_is_alive, 2-15, 4-57
cd_is_alive replacement, 2-13
changes in TIGA 2.0, 2-13
character height, 7-3
character offset, 7-4
character origin, 7-3
character rectangle, 7-3
character width, 7-3
clear functions, 3-10
clear_frame_buffer, 3-11
clear_page, 3-11
clear_screen, 3-11
clipping, 5-73, 5-75, 5-79, 5-84, 5-99
  window, 6-18
cltiga batch file, 2-10
COFF loader, 4-57
color, A-5
comm_buff_size, A-3
command buffer, 1-4, 3-13, 8-10, 8-12, 8-15, 8-16
command number, 1-4, 4-85, 8-4, 8-9, 8-10, 8-12
communication driver, CD, 1-3, 2-8, 2-9, 3-13, 8-10, 8-12, 8-15, 8-45
communication functions, 1-3, 3-18
compatibility functions, C-14
CONFIG structure, 3-10, 3-13, 3-16, 4-61, 8-12, 8-15, A-3
coordinate
  pixel, 4-12, 4-16
  screen, 4-101
  x-y, 4-12, 4-45, 4-97, 4-122, 4-133
cop2gsp, 3-18, 4-11
coprocessor, 3-18, 4-11, 4-52
core functions, reserved symbols, B-3
cp_alt, 8-12, 8-14
cp_cmd, 8-12, 8-14
cp_ret, 8-12
cpw, 3-11
create_alm, 3-18, 4-14, 4-112, 8-2, 8-3, 8-7, 8-8, 8-45
create_esym, 3-18, 8-45
current_mode, A-4

cursor, 4-26, 4-27, 4-61, 4-103, 4-108, 4-109, 8-36, A-5
CURSOR structure, A-5
CURSOR structure change, 2-13
CURSOR structure, version 1.1, 2-14
CURSOR structure, version 2.0, 2-14
cvxyl, 3-16

D

data, A-6, A-15
debugger functions, C-2
decode_rect, 3-14
delete_font, 3-15
depth, A-15
descent, 7-2
destination bit map, 5-4
development tools, 3-2
device_rev, A-4
direct mode, 8-9, 8-10, 8-16, 8-28
disp_hres, A-9
disp_pitch, A-9
disp_psize, A-9
disp_vres, A-9
display_mem_end, A-4
display_mem_start, A-4
dm_cmd, 8-17
dm_ipoly, 8-15, 8-25
dm_palt, 8-21
dm_pcmd, 8-22
dm_pget, 8-20
dm_poly, 8-15, 8-22
dm_pret, 8-22
dm_psnd, 8-19
dm_pstr, 8-21
dm_ptrx, 8-21
dm_ret, 8-18
documentation files, 2-5
double buffering, 4-92
downward compatibility, 2-13
draw_line, 3-12, 6-3, 6-8
draw_oval, 3-12, 6-3, 6-8
draw_ovalarc, 3-12, 6-3, 6-8
draw_piearc, 3-12, 6-3, 6-8
draw_point, 3-12, 6-3
draw_polyline, 3-12, 3-14, 6-3, 6-8
draw_rect, 3-12, 6-3, 6-8
drawing origin, 4-27, 4-109, A-7
driver developer's kit, DDK, 2-2
driver development package, DDP
  subdirectories, 2-5
  system requirements, 2-3
dstbm, A-7
dynamic load module, DLM, 8-2, 8-9

E

elliptical arc, 5-13
encode_rect, 3-14
ENVIRONMENT structure, 3-11, A-7
environment variable, 2-7, 2-8, 4-14, 8-6, 8-45
  –i option, 2-8
  –l option, 2-8
  –m option, 2-8
extended function development, 3-5
extended functions, reserved symbols, B-6
extensibility, 1-2, 1-3, 1-5, 4-14, 4-20, 4-30, 4-61, 4-81, 4-83, 4-112, 8-1
extensibility functions, 3-18

F

field_extract, 3-18
field_insert, 3-18
fill_convex, 3-12, 3-14, 6-3
fill_oval, 3-12, 6-3
fill_piearc, 3-12, 6-3
fill_polygon, 3-12, 3-14, 6-3
fill_rect, 3-12, 6-3
flush_esym, 3-18
flush_extended, 3-18, 4-20
flush_module, 3-18
font, 2-11
  alphabetical listing, 7-16
  available, 7-14
  bit-mapped parameters, 7-2
  block, 7-11
  data structure, 7-5
  database summary, 7-14
  header information, 7-5
  height, 7-3

- installing, 7-15
- location table, 7-10
- names, 7-15
- offset/width, 7-10
- pattern table, 7-8
- proportionally spaced, 4-13, 7-11
- size, 4-80
- table, 5-9, 5-41, 5-77, 7-12
- TIGA-compatible, 2-5

font structure
- ascent, 7-7
- avgwide, 7-7
- charhigh, 7-7
- charwide, 7-7
- default, 7-6
- descent, 7-7
- facename, 7-6
- first, 7-6
- last, 7-6
- leading, 7-7
- length, 7-6
- magic, 7-5
- maxkern, 7-6
- maxwide, 7-6
- oLocTbl, 7-8
- oOwTbl, 7-8
- oPatnTbl, 7-7
- rowpitch, 7-7

FONTINFO structure, A-8
fontptr, A-8
foreground color, 8-29
frame thickness, 5-34
frame_oval, 3-13, 6-3
frame_rect, 3-13, 6-3
function argument, 8-13
function_implemented, 3-10, 4-11, 4-22

G

g, A-14
get_colors, 3-11
get_config, 3-10, 4-61, 8-12, 8-15
get_curs_state, 3-15, 4-26
get_curs_xy, 3-15, 4-27
get_env, 3-11
get_fontinfo, 3-15
get_isr_priorities, 3-18, 4-30, 4-81, 4-83, 4-112
get_memseg, C-3
get_modeinfo, 3-10
get_msg, C-4
get_nearest_color, 3-12
get_offscreen_memory, 3-17, A-13
get_palet, 3-12
get_palet_entry, 3-12
get_pixel, 3-16
get_pmask, 3-11
get_ppop, 3-11
get_text_xy, 3-15
get_textattr, 3-15
get_transp, 3-11
get_vector, 3-19
get_videomode, 3-10, 4-48, 4-126
get_windowing, 3-11
get_wksp, 3-16
get_xstate, C-5
gm_idlefunc, 3-10
graphics attributes control functions, 3-11
graphics cursor functions, 3-15
graphics drawing functions, 3-12
graphics library, 2-11
graphics library functions, 4-82, 8-10
graphics manager, GM, 1-4, 2-9, 3-13, 4-57, 4-81, 4-83, 8-13, 8-28, 8-30, 8-47
graphics utility functions, 3-16
gsp_calloc, 3-17, 4-56
gsp_execute, 3-10, 4-57, 4-88
gsp_free, 3-17, 4-58
gsp_malloc, 3-17, 4-59
gsp_maxheap, 3-17, 4-60
gsp_minit, 3-17, 4-61
gsp_realloc, 3-17, 4-62
gsp2cop, 3-18, 4-22, 4-52
gsp2gsp, 3-18
gsp2host, 3-18, 4-54
gsp2hostxy, 3-18, 4-55
gsph_alloc, 3-17
gsph_calloc, 3-17
gsph_compact, 3-17
gsph_deref, 3-17
gsph_falloc, 3-17
gsph_fcalloc, 3-17
gsph_findhandle, 3-17
gsph_findmtype, 3-17

gsph_free, 3-17
gsph_init, 3-17
gsph_maxheap, 3-17
gsph_memtype, 3-17
gsph_realloc, 3-17
gsph_totalfree, 3-17

H

handle-based functions, 3-16
header, 5-80, 5-101
height, A-5, A-15
host-PC development tools, 3-2
host-PC include files, 3-3
host-PC libraries, 3-3
host2gsp, 3-18
host2gspxy, 3-18, 4-78
hot_x, A-5
hot_y, A-5

I

i, A-14
I/O functions, 3-18
id, A-8
image width, 7-4
in_font, 3-15
include files, 2-5, 2-7, 3-3, 3-5, 8-10, 8-13, 8-28, A-1
include files for PC development, 3-3
INIT_GM, 4-127
init_palet, 3-12, 4-22
init_text, 3-15
initialization, 2-9, 3-6, 4-61, 8-47
initialization functions, 3-10
initialization/termination, 2-13
install_alm, 3-18, 4-30, 4-81, 4-112, 8-8, 8-10, 8-45
install_font, 3-15
install_primitives, 3-10, 3-18, 4-82, 8-9
install_rlm, 3-18, 4-30, 4-83, 4-112, 8-7, 8-10, 8-45
install_usererror, 3-10, 4-85, 4-123, 4-127
installation, 2-4, 8-6, 8-36
integral data types, A-2
intercharacter spacing, 5-38, 5-86, 5-87
interrupt, 2-8, 3-18, 4-30, 4-81, 4-83, 4-112, 8-2, 8-4, 8-28, 8-30
interrupt handler functions, 3-19

L

leading, 7-3
leftmost one, 4-87
libraries, 3-5
line-style, pattern, 4-101, 5-88, 5-89, 5-90, 5-92, 5-93, 5-94, 5-95, 6-13
linking loader, 8-45
  options, 8-45
lmo, 3-16
loadcoff, 3-10, 4-57, 4-88

M

magic, 5-22
mask_color, A-6
mask_rop, A-5
math/graphics, 2-10
memory management, 8-3, A-3
  handle-based functions, 3-16
  pointer-based functions, 3-17
mg2tiga utility, 2-11
mode, A-4
mode arguments, 4-126
  AI_8514, 4-126
  CGA, 4-126
  EGA, 4-126
  HERCULES, 4-126
  MDA, 4-126
  OFF_MODE, 4-126
  PREVIOUS, 4-126
  tiga, 4-126
  VGA, 4-126
MODEINFO structure, A-9

N

new functions, 2-15
NO_ENABLE, 4-127
num_modes, A-4
num_offscrn_areas, A-10
num_pages, A-10

O

oem_init, C-6
OFFSCREEN structure, A-13
offscreen workspace, 2-15
operations on pixels, 6-15
origin
    character, 5-87
    drawing, 4-12, 4-16, 4-45, 4-97, 4-101, 4-122, 4-133, 5-57, 5-59, 5-62, 5-65, 5-67, 5-70, 5-72, 5-73, 5-75, 5-78, 5-81, 5-88, 5-90, 5-92, 5-94, 5-98
outcode, 4-12

P

packet header, 8-13
page flip, 4-89
page_busy, 3-19
page_flip, 3-19
PALET structure, A-14
palet_gun_depth, A-9, A-10
palet_inset, A-10
palet_size, A-10
palette, 4-7, 4-8, 4-34, 4-38, 4-40, 4-79, 4-101, 4-115, 4-116, 4-117, A-14
palette functions, 3-12
patn, 6-12
patnfill_convex, 3-13, 3-14, 6-2, 6-6
patnfill_oval, 3-13, 6-2, 6-6
patnfill_piearc, 3-13, 6-2, 6-6
patnfill_polygon, 3-13, 3-14, 6-2, 6-6
patnfill_rect, 3-13, 6-2
patnframe_oval, 3-13, 6-3
patnframe_rect, 3-13, 6-3
patnpen_line, 3-13, 6-2
patnpen_ovalarc, 3-13, 6-3
patnpen_piearc, 3-13, 6-3
patnpen_point, 6-2
patnpen_polyline, 3-13, 3-14, 6-2
pattern, 3-11, 3-13, 5-81, 5-83, 6-3, A-15
    area-fill, 4-101, 5-56, 5-57, 5-59, 5-61, 5-62, 5-75, 5-76, 5-82, 5-101
    line-style, 4-101, 5-88, 5-89, 5-90, 5-92, 5-93, 5-94, 5-95
PATTERN structure, A-15
peek_breg, 3-16
pen, 3-13, 4-101, 5-57, 5-59, 5-61, 5-62, 5-64, 5-65, 5-67, 5-69, 5-70, 5-83, 6-2, 6-10, A-7
pen_line, 3-13, 6-2
pen_ovalarc, 3-13, 6-2
pen_piearc, 3-13, 6-2
pen_point, 3-13, 6-2
pen_polyline, 3-13, 3-14, 6-2
pie chart, 5-28
pitch, 5-79, 5-80, 5-84, 5-85, A-5
pixel array function, 3-14
pixel processing, 4-103
pixel_mask, A-9
pixel-processing operation, 4-101, 4-138, 4-139, 5-4, 5-100, 6-16
pixel-size independence, 6-19
plane mask, 4-42, 4-46, 4-101, 4-118, 4-138, 4-139, 5-4, 6-16
pointer-based functions, 3-17
poke_breg, 3-16
poly drawing functions, 3-13, 6-3, 8-15
program_mem_end, A-4
program_mem_start, A-4
proportionally spaced, 4-80, 5-86, 5-87
proprietary extension, 4-129
put_pixel, 3-13

R

r, A-14
read_hstaddr, C-15
read_hstadrh, C-16
read_hstadrl, C-17
read_hstctl, C-18
read_hstdata, C-19
rectangular drawing pen, 6-10
register usage, 8-29
relocatable load module, RLM, 4-14, 4-83, 8-2, 8-5, 8-46, 8-47
    installation, 8-6
rightmost one, 4-95
rmo, 3-16
rstr_commstate, C-7
run-length encoding, 5-21

S

sample TIGA application, 3-6
save_commstate, C-8
screen_high, A-9
seed fill, 5-73
seed_fill, 3-13
seed_patnfill, 3-13
select_font, 3-15
set_bcolor, 3-11
set_clip_rect, 3-11
set_colors, 3-11
set_config, 2-14, 3-10, A-3
  return value, 2-14
set_curs_shape, 3-15, 4-61, 4-103, A-5
set_curs_state, 3-15, 4-108
set_curs_xy, 3-15, 4-103, 4-109
set_cursattr, 3-15
set_draw_origin, 3-11, A-7
set_dstbm, 3-14, A-7
set_fcolor, 3-11
set_interrupt, 3-19, 4-112, 8-37
set_memseg, C-9
set_module_state, 3-18
set_msg, C-10
set_palet, 3-12, 4-22
set_palet_entry, 3-12, 4-22
set_patn, 3-11, 6-12, A-15
set_pensize, 3-11
set_pmask, 3-11
set_ppop, 3-11
set_srcbm, 3-14, A-7
set_text_xy, 3-15
set_textattr, 3-15
set_timeout, 3-10, 4-123
set_transp, 3-11, 4-22
set_vector, 3-19
set_videomode, 2-9, 3-10, 4-48, 4-126
set_videomode replacement, 2-13
set_windowing, 3-11
set_wksp, 3-16, 4-61
set_xstate, C-11
setup_hostcmd, 3-10
shape_rop, A-5
share_gsp_addr, A-4
share_host_addr, A-4
share_mem_size, A-4
silicon_capability, A-10
software developer's kit, SDK, 2-2
software development package, SDP, subdirectories, 2-5
software porting kit, SPK, 2-2
software porting package, SPP
  subdirectories, 2-5
  system requirements, 2-3
source bit map, 5-4
srcbm, A-7
stack_size, 4-61, A-4
style argument
  CLR_SCREEN, 4-127
  INIT, 4-127
  INIT_GLOBALS, 4-126
  NO_INIT, 4-126
styled_line, 3-13
styled_oval, 3-13
styled_ovalarc, 3-13
styled_piearc, 3-13
stylemask, A-7
supported development tools, 3-2
swap_bm, 3-14
sym_flush, 3-18
symbol table, 3-18, 8-46, 8-47
synchronize, 3-10, 4-132
sys_flags, 4-11, 4-52, A-3
system requirements, 2-3

T

termination, 3-6
text
  alignment, 5-38, 5-87
  functions, 2-11, 3-15
text attributes
  alignment, 7-13
  intercharacter gaps, 7-13
  intercharacter spacing, 7-13
text_out, 3-15
text_outp, 3-15
text_width, 3-15
text-related functions, 7-2
TIGA, 5-22

TIGA 1.1, 2-13
tiga_busy, 3-10
tiga_set, 3-10
tigacd, 2-9
TIGAEXT section, 4-81, 4-83, 8-4, 8-5, 8-10, 8-12, 8-45
TIGAISR section, 8-4, 8-36, 8-45
TIGALNK, 8-45
  options, 8-45
TMS340 development products, 2-2
TMS340 development tools, 3-2
TMS340 function library, 2-11
TMS340 include files and libraries, 3-5
TMS34082 coprocessor, 4-52
transp_off, 3-11
transp_on, 3-11
transparency, 4-22, 4-42, 4-46, 4-101, 4-118, 4-124, 4-138, 4-139, 5-4, 5-100, 6-15, 8-29
trap vector, 4-47, 8-36

unsupported functions, 2-15
utilities, 2-5, 2-10, 3-16

version_number, A-3

wait_scan, 3-19
width, A-5, A-15
windowing modes, 4-49
  default, 4-49
wksp_addr, A-10
wksp_pitch, A-10
write_hstaddr, C-20
write_hstadrh, C-21
write_hstadrl, C-22
write_hstctl, C-23
write_hstdata, C-24

x-y coordinates, 4-13
xext, A-13
xyorigin, A-7

yext, A-13

zoom, 5-98
zoom_rect, 3-14