TEXAS INSTRUMENTS

TIGA-340™ *Interface*

Texas Instruments Graphics Architecture

User's Guide

User's Guide

TIGA-340™ Interface
Texas Instruments
Graphics Architecture

1989

1989

Graphics Products

TIGA-340™ *Interface*
Texas Instruments Graphics Architecture User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

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

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TIGA and TIGA-340 are trademarks of Texas Instruments Incorporated.

ADI and AutoCAD are trademarks of Autodesk Inc.

DGIS is a trademark of Graphic Software Systems, Inc.

GEM is a trademark of Digital Research Inc.

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

Macintosh is a trademark of Apple Computer Corp.

NEC is a trademark of NEC Corp.

PC-DOS and PGA are trademarks of IBM Corp.

Sony is a trademark of Sony Corp.

Preface

Read This First

How to Use This Manual

This document contains the following chapters:

Chapter 1 Introduction
Introduces the TIGA-340 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 TIGA Application Interface
Describes the application interface and lists all TIGA primitive instructions.

Chapter 4 Extensibility Through the User Library
Describes how to extend TIGA by adding your own functions; also describes the command processing entry points of the communication driver.

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

Appendix B Graphics Output Primitives
Describes the assumptions made and conventions adopted for the drawing primitives.

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

Appendix D Porting Guide
Describes the procedure to port TIGA to any TMS340-based graphics board.

Appendix E Debugger Support for TIGA
Contains the initial TIGA debugger routines.

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

Related Documentation

The following TMS34010 documents are available from Texas Instruments:

- The ***TMS34010 C Compiler User's Guide*** (literature number SPVU005) tells you how to use the TMS34010 C Compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS34010 assembly language source code. We suggest that you use ***The C Programming Language*** book (by Brian W. Kernighan and Dennis M. Ritchie) as a companion to the *TMS34010 C Compiler User's Guide.*
- The ***TMS34010 Assembly Language Tools User's Guide*** (literature number SPVU004) describes common object file format, assembler directives, macro language, and assembler, linker, archiver, simulator, and object format converter operation.
- The ***TMS34010 Data Sheet*** (literature number SPVS002) contains the recommended operating conditions, electrical specifications, and timing characteristics of the TMS34010.
- The ***TMS34010 User's Guide*** (literature number SPVU001) discusses hardware aspects of the TMS34010, such as pin functions, architecture, stack operation, and interfaces, and contains the TMS34010 instruction set.

You may also find the following documentation useful:

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

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

Style and Symbol Conventions

This document uses the following conventions:

- TIGA primitive function names are shown in **bold face** lettering.
- Parameters for the TIGA functions are shown in `program font` (Courier). For example, the TIGA function **draw_line** has parameters `x1, y1, x2, y2`.
- Program examples and filenames (example: `TIGALNK.EXE`) are shown in `program font`. Here is an example program:

```
#include <tiga.h>

main()
{
   short module;

   /* initialize TIGA */
   if (!set_videomode(TIGA, INIT))
   {
      printf("Fatal Error - TIGA not installed\n");
      exit(0);
   }
   /* attempt to install module */
   if ((module = install_rlm("EXAMPLE")) < 0 )
   {
      printf("Fatal Error - couldn't install Example RLM\n");
       printf("Error code = %d\n", module);
      exit(0);
   }
   /* code to invoke the module functions */
      :
      :
   set_videomode(PREVIOUS, INIT);
}
```

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

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

Contents

1 Introduction 1-1
1.1 Developer's Kits 1-2
1.2 Features 1-3
1.3 Architecture 1-4
1.4 Extensibility 1-6

2 Getting Started 2-1
2.1 System Requirements 2-2
2.1.1 TIGA Driver Developer's Kit (DDK) System Requirements 2-2
2.1.2 TIGA Software Porting Kit (SPK) System Requirements 2-2
2.2 Installing TIGA on Your System 2-4
2.2.1 Driver Developer's Kit (DDK) Subdirectories 2-5
2.2.2 Software Porting Kit (SPK) Subdirectories 2-5
2.2.3 TIGA Demonstrations and Example Subdirectories 2-6
2.2.4 Running a TIGA Demonstration 2-6
2.2.5 TIGA Include Files 2-8
2.3 Modifying Autoexec and the Environment 2-9
2.4 Running the TIGA Driver 2-10
2.5 The TIGA Environment Variable 2-11
2.6 TIGA Utility Programs 2-12
2.6.1 cc Utility 2-12
2.6.2 cltiga Batch File 2-12
2.6.3 mg2tiga Utility 2-13
2.6.4 tigamode Utility 2-16
2.7 Non-TIGA Development Utility Programs 2-17
2.8 TIGA Syntax and Programming Examples 2-18

3 TIGA Application Interface 3-1
3.1 Base Set of TIGA Primitives 3-2
3.2 Summary Table of Functions by Functional Group 3-3
3.2.1 Graphics System Initialization Functions 3-3
3.2.2 Clear Functions 3-4
3.2.3 Graphics Attribute Control Functions 3-5

3.2.4 Palette Functions 3-6
3.2.5 Graphics Output Functions 3-7
3.2.6 Poly Drawing Functions 3-8
3.2.7 Workspace Functions 3-8
3.2.8 Pixel Array Functions 3-8
3.2.9 Text Functions 3-9
3.2.10 Cursor Functions 3-10
3.2.11 Graphics Utility Functions 3-10
3.2.12 Pointer-Based Memory Management Functions 3-11
3.2.13 Communication Functions 3-11
3.2.14 Extensibility Functions 3-12
3.3 Alphabetical List of Functions 3-13

4 Extensibility Through the User Library 4-1
4.1 Dynamic Load Module 4-2
4.1.1 Relocatable Load Modules 4-2
4.1.2 Absolute Load Modules 4-2
4.2 Generating a Dynamic Load Module 4-4
4.2.1 TIGAEXT Section 4-4
4.2.2 The TIGAISR Section 4-5
4.2.3 Linking the Code and Special Sections into an RLM 4-5
4.3 Installing a Dynamic Load Module 4-7
4.3.1 Installing a Relocatable Load Module 4-7
4.3.2 Installing an Absolute Load Module 4-8
4.4 Invoking Functions in a Dynamic Load Module 4-10
4.4.1 Command Number Format 4-10
4.4.2 Using Macros in Command Number Definitions 4-11
4.4.3 Passing Parameters to the TIGA Function 4-12
4.5 C-Packet Mode 4-13
4.5.1 The Type of Call 4-13
4.5.2 The Command Number 4-13
4.5.3 Description of Function Arguments 4-14
4.5.4 C-Packet Examples 4-14
4.5.5 Overflow of the Command Buffer 4-16
4.6 Direct Mode 4-18
4.6.1 Standard Command Entry Point 4-18
4.6.2 Standard Command Entry Point with Return 4-20
4.6.3 Standard Memory Send Command Entry Point 4-21
4.6.4 Standard Memory Return Command Entry Point 4-23
4.6.5 Standard String Entry Point 4-24
4.6.6 Altered Memory Return Command Entry Point 4-24

4.6.7 Send/Return Memory Command Entry Point 4-24
4.6.8 Mixed Immediate and Pointer Command Entry Point 4-25
4.6.9 Mixed Immediate and Pointer Command Entry Point w/ Return 4-25
4.6.10 Poly Function Command 4-26
4.6.11 Immediate and Poly Data Entry Point 4-28
4.7 Downloaded Function 4-32
4.7.1 Register Usage Conventions 4-33
4.7.2 TIGA Graphics Manager System Parameters 4-35
4.8 Example Programs 4-36
4.8.1 Stars Example 4-36
4.8.2 Curves Example Program 4-40
4.8.3 ADI Driver Example 4-43
4.9 Installing Interrupts 4-44
4.9.1 Clock Example of Using Interrupts 4-45
4.9.2 Ball Example Using Interrupts 4-46
4.10 The TIGA Linking Loader 4-47
4.10.1 /ca – Create Absolute Load Module 4-48
4.10.2 /cs – Create External Symbol Table 4-48
4.10.3 /ec – Error Check 4-48
4.10.4 /fs – Flush External Symbol Table 4-49
4.10.5 /la – Load and Install an Absolute Load Module 4-49
4.10.6 /lr – Load and Install a Relocatable Load Module 4-49
4.10.7 /lx – Load and Execute a COFF File / Execute TIGA GM 4-49

A TIGA 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-6
A.5 FONTINFO Structure A-7
A.6 MODEINFO Structure A-11
A.7 MONITORINFO Structure A-13
A.8 OFFSCREEN Structure A-14
A.9 PAGE Structure A-15
A.10 PALET Structure A-16
A.11 PATTERN Structure A-17

B Graphics Output Primitives B-1
B.1 Categories of Graphics Output Primitives B-2
B.2 Fill Patterns B-4
B.3 Mapping Pixels to XY Coordinates B-5

B.4 Area Filling Conventions B-6
B.5 Vector Drawing Conventions B-7
B.6 Drawing B-8
B.7 Color Selection B-9

C TIGA Reserved Symbols C-1
C.1 Reserved Functions C-2
C.2 TIGA Core Primitive Symbols C-3
C.3 TIGA Extended Primitive Symbols C-5

D Porting TIGA D-1
D.1 Porting the Communication Driver D-2
D.1.1 Modifying the sdbdefs.inc File D-2
D.1.2 Modifying the oemdata.asm File D-4
D.1.3 Defining the Mode-Specific Information D-4
D.1.4 Defining the Mode Label and Setup_Struc Structure D-5
D.1.5 Defining the Mode_Struc Structure D-5
D.1.6 Defining the Monitor_Info Structure D-6
D.1.7 Defining the Page_Info Structure D-6
D.1.8 Defining the Off_Screen Structure D-7
D.1.9 Defining OEM-Specific Data D-8
D.1.10 Completing Modifications to oemdata.asm D-8
D.1.11 Modifying the oeminit.asm File D-9
D.1.12 Modifying the OEM_Init Function D-9
D.1.13 Modifying the OEM_Sense Function D-10
D.1.14 Modifying the Monitor_Init Function D-10
D.1.15 Modifying the Video_Enable Function D-10
D.1.16 Modifying the setvideo.asm File D-10
D.1.17 Miscellaneous Communication Driver Porting Issues D-11
D.1.18 Default Timeout for gm_is_alive Function D-11
D.1.19 Using Boards with Multi-Addressable Host Port Locations ... D-11
D.1.20 Rebuilding the Communication Driver D-13
D.2 Porting the Graphics Manager D-14
D.2.1 Video Memory Initialization Functions D-14
D.2.2 Palette-Specific Functions D-15
D.2.3 Configuration Functions D-16
D.2.4 Miscellaneous Functions D-17
D.2.5 Rebuilding the Graphics Manager D-19
D.3 Verifying Correct Operation D-21
D.4 Debugging Your Port D-22

E Debugger Support for TIGA E-1
D.1 TIGA Debugger Routines E-2

D.2 Compatibility Functions E-12

F Glossary F-1

Figures

1–1. Block Diagram 1-4
1–2. Primitive Configuration Options 1-6
4–1. Command Number Format 4-10
4–2. Data Structure of dm_cmd 4-19
4–3. Data Structure of dm_psnd 4-21
4–4. Data Structure Before Invoking dm_pget 4-23
4–5. Data Structure After Invoking dm_pget 4-23
4–6. Data Structure of dm_poly 4-26
A–1. Bitmap Font Format A-10
B–1. A 16 x 16 Pattern B-4
B–2. Rectangle Fill B-5
B–3. Polygon Fill B-6
B–4. Polygon Outline B-7

Tables

3–1. Pixel Processing Options 3-74
3–2. Pixel Processing Options 3-165
B–1. List of Function Types and Drawing Styles B-2
B–2. Checklist of Available Figure Types and Drawing Styles B-3

Examples

Example 4–1. 4-7
Example 4–2. 4-8
Example 4–3. 4-27

Chapter 1

Introduction

This user's guide describes the TIGA-340™ (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.

Section **Page**

1.1 Developer's Kits 1-2
1.2 Features 1-3
1.3 Architecture 1-4
1.4 Extensibility 1-6

The TIGA interface standard simplifies the development of portable applications and application drivers to the diverse range of TMS340-based systems. TIGA can be extended so that software developers can customize TIGA-340 to a specific application and so that hardware developers can provide a simple interface to specific target features.

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.

1.1 Developer's Kits

This user's guide supports two basic TIGA developer's kits, the **Driver Developer's Kit (DDK)** and the **Software Porting Kit (SPK)**. A third developer's kit, the **Software Developer's Kit (SDK)**, includes the DDK as a subset.

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

  The TIGA-340 DDK provides software designers with the tools required to produce TIGA-compatible applications. These tools include a copy of the TIGA-340 interface, which runs on the TMS34010 Software Development Board, a user's guide, utilities, an AutoCAD release 9 sample driver, and several TIGA-compatible example programs. With these tools a programmer can modify existing applications to run with the TIGA-340 interface or develop new TIGA-compatible applications. The development of TIGA-compatible applications is easy because the tools in the kit are designed to work with the industry-standard Microsoft C development tools and debugging environments.

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

  The SPK helps hardware manufacturers and software operating environment developers make their TMS340-based systems TIGA-compatible.The SDK, described below, is included as a subset to this kit.

  The SPK contains everything in the DDK plus all source code required to port the TIGA-340 interface to any TMS340 system, allowing all TIGA-compatible applications to run on that system. TMS340 source and 8086 object codes for a TIGA-compatible Windows/286 driver are also included.

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

  The SDK is for those who want to develop direct TMS34010 code or custom, downloadable extensions to TIGA. In addition to the DDK, it includes TI's TMS34010 C Compiler, Assembler, Bitmap Font, and Math/Graphics source code libraries.

1.2 Features

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

Applications run faster TIGA-340 provides the application writer with a dual-processor environment. This enables the tasks in the application to be run in parallel by partitioning them between the host and the TMS340 processors. The TIGA-340 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.

Easy to use TIGA-340 provides applications with a base set of graphics primitives, with all the support required for the graphics subsystem. TIGA-340 is compatible with the Microsoft C environment, and Microsoft development tools can be used for debugging the application.

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

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

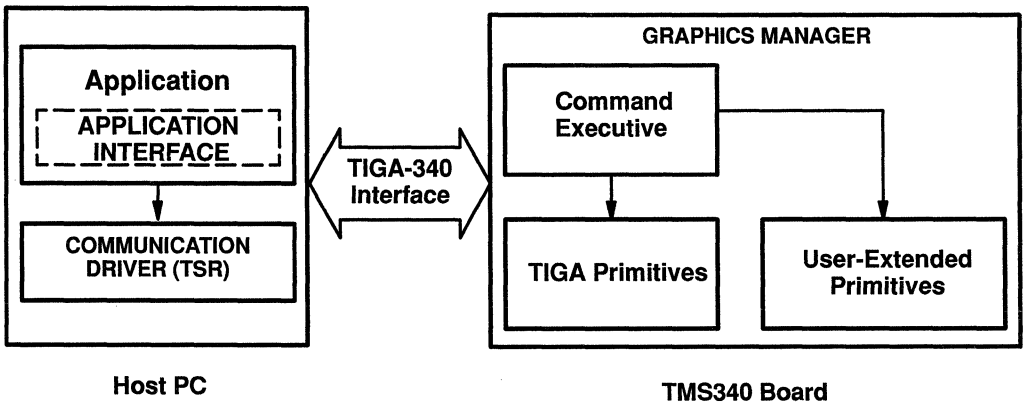
1.3 Architecture

Figure 1–1 shows a block diagram of the TIGA-340 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)
2) Communication Driver (CD)
3) Graphics Manager (GM)
4) TIGA Extensions

Figure 1–1. Block Diagram



The Application Interface (AI) is linked in with a TIGA Application. 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 by the user 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. These functions are in-

voked via calls in the AI made by the application. These communication functions take care of the host side hardware-dependent portion of TIGA, considering such things as whether the TMS340 board is memory-mapped or I/O-mapped.

The Graphics Manager (GM) is a program that runs on the TMS340 board and is specific to the board that it resides on. It consists of a command executive that handles the TMS340 side of the communications with the host, and a set of standard primitives that perform graphics operations. The GM typically resides in RAM on the board (although this is not a requirement) and therefore must be loaded onto the board after power-up. There are two mechanisms for doing this. TIGA comes with a linking loader `TIGALNK.EXE`, which loads the GM explicitly by invoking it with the `-lx` (Load and eXecute) flag. Alternately, the communication driver routines can sense whether the GM is running; when the application makes its first AI call, it detects if AI is not running and loads it.

TIGA contains a set of primitives to perform a wide range of graphics operations.The set of primitives can be extended by downloading the application functions onto the TMS340 system. These downloaded functions may be written using either the TMS340 C or assembly languages. Downloading functions can decrease the host-TMS340 processor communications time and thus improve the performance of the application.

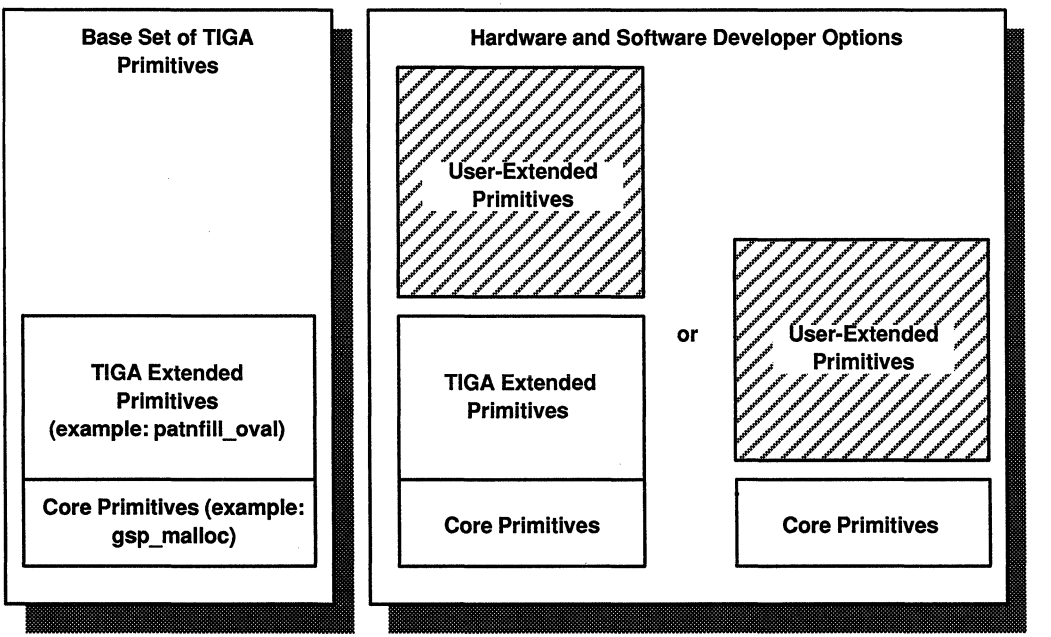
The host application invokes most of the TIGA functions on the TMS340 processor by downloading the parameters of the function, along with a command number, into one of several communication buffers. The command number is an identifier for the function to be executed. The command executive, which forms 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.

1.4 Extensibility

Graphics standards before TIGA limited the software developer by providing a fixed set of graphics drawing primitives. In the rapidly changing graphics market, a fixed set of primitives is unacceptable. During the development of the TIGA graphics interface, incorporating extensibility into the standard was a major design goal.

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

Figure 1–2. Primitive Configuration Options



TIGA-compatible applications can be developed using the base set of primitives provided by the TIGA-340 Interface (as shown in the left hand side of Figure 1-2). These TIGA primitives include the core primitives, which are always available to the application, and the TIGA extended primitives, which are loaded if the application requires them.The set of graphics primitives and the performance of the TMS340 processor give many applications an acceptable level of graphics performance. However, an application has the ability to improve this performance by downloading user-extended primitives. The user-extended primitives can be downloaded to be used in addi-

tion to or instead of the TIGA extended primitives (as shown on the right-hand side of Figure 1-2).

A hardware developer can implement the same concept of adding primitives. For example, if the developer of a TMS340-based graphics system incorporates hardware in addition to the TMS340 processor, access to this hardware can be provided through the TIGA interface. The access is accomplished by developing a set of user-extended primitives which use the additional hardware functionality. Thus, the TIGA-340 interface provides a standard programming platform for the software written by the hardware developer using these user-extended primitives.

Chapter 2

Getting Started

This chapter contains instructions for installing TIGA on your system:

Section		Page
2.1	System Requirements	2-2
2.2	Installing TIGA on your System	2-4
2.3	Modifying the `Autoexec` and the Environment	2-9
2.4	Running the TIGA Driver	2-10
2.5	The TIGA Environment Variable	2-11
2.6	TIGA Utility Programs	2-12
2.7	Required Non-TIGA Development Utility Programs	2-17
2.8	TIGA Syntax and Programming Examples	2-18

2.1 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 kit you are installing.

2.1.1 TIGA Driver Developer's Kit (DDK) System Requirements

- IBM PC, XT, AT or 100% compatible (hard disk required)
- 640K RAM
- LIM expanded memory plus expanded memory manager (for ADI driver only)
- Texas Instruments TMS34010 Software Development Board (SDB)
- MS-DOS or PC-DOS, version 2.13 or above
- Microsoft Macro Assembler, version 5.0 or above (if developing assembler-based applications/drivers)
- Microsoft C Compiler, version 5.0 or above (if developing C applications)
- TMS34010 C Compiler and assembly-language tools, version 3.0 or above (if writing user-extended functions)

Note:

The current version of TIGA supplied with the DDK is designed to run **only** on a Texas Instruments software development board (SDB). Pricing and ordering information for the SDB is available from your local Texas Instruments Sales Office.

2.1.2 TIGA Software Porting Kit (SPK) System Requirements

- IBM PC,XT,AT or 100% compatible (hard disk required)
- 640K RAM
- LIM expanded memory plus expanded memory manager (for ADI driver only)
- Texas Instruments TMS34010 Software Development Board (SDB) (Only if you are not porting TIGA to a different board)
- MS-DOS or PC-DOS, version 2.13 or above

- Microsoft Macro Assembler, version 5.0 or above
- Microsoft C Compiler, version 5.0 or above (if developing C applications)
- TMS34010 C Compiler and assembly-language tools, version 3.0 or above (if writing user extended functions and/or if porting TIGA to a different board)

Note:

The current version of TIGA supplied with the SPK is designed to run on a Texas Instruments software development board. However, all software necessary to port TIGA to a different TMS340 board is included in in the SPK. Consult Appendix D for information on how to port TIGA to a different TMS340 board.

2.2 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` and `tigapgms` directories. For this reason, files of previous versions of TIGA should be backed up, if needed, before proceeding with the new TIGA installation.

Both the TIGA DDK and SPK kits have an automated installation program to aid in installing TIGA on your system. In general, the installation procedure for the DDK and the SPK are the same.

The DDK package is a subset of the SPK; the SPK package contains **all** of the files in the DDK plus additional files required for porting purposes. Therefore, if you have both the DDK and the SPK packages, install only the SPK on your system.

Follow these instructions to install your TIGA kit:

Step 1: Place disk #1 (DDK #1 or SPK #1) of your TIGA kit into drive A:

Step 2: If A: is not your current drive, enter `A:` ↵ at the MS-DOS prompt.

Step 3: Make sure you are at the root directory of A:. If you are not sure, enter **`cd\`** ↵ at the MS-DOS prompt.

Step 4: Enter **`setup`** *`drive:`* ↵ where *drive:* designates your destination drive (hard disk). For example, if you want to install TIGA on your C: hard disk, enter **`setup C:`** ↵.

Step 5: Follow the instructions displayed on the screen to complete installation.

During installation, you have the option of installing a collection of TIGA-compatible examples and demonstrations. It is recommended that these examples be installed because they are referenced as coding examples in this guide.

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

2.2.1 Driver Developer's Kit (DDK) Subdirectories

Installing the DDK on your system creates the following subdirectories:

Subdirectory	Description
\tiga	TIGA root directory, TIGA drivers, system files, and utility programs
\tiga\ai	Application interface library
\tiga\gm\extprims	Extended primitives source code archive
\tiga\include	Include files
\tigapgms	TIGA compatible examples and demonstrations (see Section 2.2.3 for more information)

The \tiga\gm\extprims directory contains the self-extracting archive file extprims.exe. This archive contains source for every extended 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 unarchive the source files contained in this archive, enter **extprims** ↵ from within this directory.

2.2.2 Software Porting Kit (SPK) Subdirectories

Installing the DDK on your system creates the following subdirectories:

Subdirectory	Description
\tiga	TIGA root directory, TIGA drivers, system files, and utility programs
\tiga\ai	Application interface library and source
\tiga\cd	TIGA communication driver source
\tiga\gm	TIGA graphics manager root directory
\tiga\gm\corprims	TIGA graphics manager core primitives
\tiga\gm\extprims	TIGA graphics manager extended primitives
\tiga\gm\sdb	TIGA graphics manager SDB specific files
\tiga\include	Include files
\tigapgms	TIGA compatible examples and demonstrations (see Section 2.2.3 for more information)

2.2.3 TIGA Demonstrations and Example Subdirectories

During installation, you have the option of installing a collection of TIGA compatible demonstrations and examples. When installed, the following directories are created on the destination drive:

`\tigapgms`	TIGA compatible examples and demonstrations
`\tigapgms\adi`	TIGA compatible AutoCAD example driver
`\tigapgms\asmtst`	Assembly language application example
`\tigapgms\ball`	Interrupt example using graphics
`\tigapgms\clock`	Downloaded interrupt function example
`\tigapgms\curves`	Floating-point example
`\tigapgms\examples`	Examples supplied in Chapter 3 of this guide.
`\tigapgms\flysim`	3-D flying simulator example
`\tigapgms\stars`	Downloaded function example
`\tigapgms\tests`	TIGA test suite
`\tigapgms\tigademo`	Demonstrates many of TIGA's functions
`\tigapgms\tigamode`	Board mode query/initialization program
`\tigapgms\tigalogo`	Demonstrates fixed-point math operations
`\tigapgms\windows`	TIGA-compatible MS-Windows driver (SPK only)

Each subdirectory contains a `readme.1st` file, complete source, and a make file to rebuild that particular demonstration. Consult the `readme.1st` file for a description of the demonstration and rebuilding instructions.

If you opted not to load the TIGA examples during the initial TIGA kit installation, you can install the examples and demonstrations at any time, following the installation instructions outlined on page 2-4. Simply insert program disk #1 into drive A: instead of the DDK or SPK disk #1 as directed in step 1. Continue installation as described.

2.2.4 Running a TIGA Demonstration

After installing TIGA on your system, you can run any of the supplied demonstration programs. One such program, `tigademo`, is a free-running demonstration of TIGA's graphics primitives. The following instructions outline how to run `tigademo`:

1) At the MS-DOS prompt, enter

   `set path=c:\tiga`

   This provides access to TIGA's DOS commands from any directory.

2) Enter

   `set TIGA=-mc:\tiga`

   This informs TIGA to look in the `c:\tiga` directory for TIGA-specific runtime files.

3) Enter

   cd \tigapgms\tigademo

   This directory contains the `tigademo` program.

4) Enter

   tigacd

   This loads the TIGA communication driver (CD). The CD provides the communication interface between a TIGA application and the target TMS340-based board. Once loaded, the CD remains resident in memory and requires reloading only after rebooting your system.

5) Finally, enter

   tigademo

   This executes `tigademo`.

You can run any of the other supplied examples in a similar manner.

Note:

Because the DDK/SPK is shipped with a TI software development board (SDB) version of TIGA, you must have an SDB in your system (or have a version of TIGA compatible with your TMS340-board) before running any of the supplied examples.

2.2.5 TIGA Include Files

The directory `\tiga\include` contains include files used in the TIGA source code itself and when writing TIGA applications. These include files are divided in two groups: those designed to run on the host processor and those designed to run on the TMS340 processor. They are also divided, as to whether they are used in C-source files or in assembler source files as follows:

- **Host-side include files**
  - **C-source**

    `extend.h` Use when calling TIGA extended primitives
    `tiga.h` Include always in a TIGA C program
    `typedefs.h` Include when using TIGA structure types
  - **Assembler source**

    `extend.inc` Use when calling TIGA extended primitives.
    `tiga.inc` Include always in a TIGA assembly program.
    `typedefs.inc` Include when using TIGA structure types
- **TMS340-side include files**
  - **C-source**

    `gspextnd.h` Use for external declarations of all TIGA extended primitives
    `gspglobs.h` Use for external declarations of all TIGA global variables
    `gspregs.h` Use for TMS340 register definitions
    `gsptiga.h` Use for external declarations of all TIGA core primitives
    `gsptypes.h` Include when using TIGA structure types
  - **Assembler source**

    `gspextnd.inc` Use for external declarations of all TIGA extended primitives
    `gspglobs.inc` Use for external declarations of all TIGA global variables
    `gspregs.inc` Use for TMS340 register definitions
    `gsptiga.inc` Use for external declarations of all TIGA core primitives
    `gsptypes.inc` Include when using TIGA structure types
    `gspmac.lib` Use for macro library

2.3 Modifying `Autoexec` and the Environment

After installing your TIGA kit, you may want 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 drive designator where you installed your particular TIGA kit:

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

```
PATH=< existing PATH>;c:\tiga
```

2) If you plan to develop TIGA-compatible applications in C or to rebuild any of the TIGA demos or the TIGA test suite, 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 C Compiler and assembly-language tools installed on your system, then append `c:\tiga\include` to the 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
```

See Section 2.5 on page 2-11 for a complete description of the TIGA environment variable.

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

2.4 Running the TIGA Driver

To load TIGA:

Enter **tigacd**↵ at the MS-DOS prompt. The command syntax for `tigacd` is:

tigacd [*flag*]

Available options:

Flag	Description
`-i`	Reinstalls the TSR. This option forces a new copy of the TIGA communication driver (CD) 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.
`-s`	Informs the TIGA CD to select operating modes valid for the SONY Multiscan monitor, rather than for the default monitor, the NEC Multisync monitor (SDB version only).

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 in one of the following ways:

- An application making a call to **set_videomode(TIGA,INIT)** checks 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 prior to returning from the **set_videomode** function.
- Enter **tigalnk -lx** from the MS-DOS command line after loading the TIGA CD, to force the TIGA GM to be loaded and executed.

After loading the host and TMS340 sides of TIGA, your application is free to call TIGA's core primitives.

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 using the following syntax:

set TIGA = [*flag*] [*string*] [*flag*] [*string*]

Currently, TIGA recognizes three flags:

Flag	Description
`-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

When TIGA is initially installed, all TIGA system files are placed in the TIGA directory of the destination drive. Specify this path using the `-m` flag 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` flag in the TIGA environment variable.

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

As an example, assume all TIGA system files are located in `c:\tiga`, user dynamic load modules are in `d:\dlm`, and the desired interrupt level to use is 0x78. Set the corresponding TIGA environment variable:

```
set TIGA=-mc:\tiga -ld:\dlm -i0x78 ↵
```

2.6 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
`cc.exe`	TMS340 tool shell program, used in make files that rebuild TIGA's graphics manager
`cltiga.bat`	Batch file to compile and link a TIGA application
`mg2tiga.exe`	Utility to convert TMS340 math/graphics fonts to TIGA compatible fonts
`tigamode.exe`	Utility to query available modes and select default mode

2.6.1 `cc` Utility

This utility is used primarily by the TIGA graphics manager make files during rebuilding. It is also useful in compiling TMS340 C or assembling assembly-language code.

`cc` is executed from the MS-DOS command line. Enter **`cc`** with no parameters to display usage instructions. Additional information on how to use the `cc` utility can be found in the *TMS34010 Software Developer's Kit (SDK) User's Guide.*

2.6.2 `cltiga` Batch File

The `cltiga.bat` batch file provides an easy way to compile and link a TIGA-compatible application (contained in a single C source file) to the TIGA application interface. It also supports symbolic debugging through Microsoft's CodeView ® debugger. The syntax for `cltiga` is:

cltiga [*–d*] *filename*

where `-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 is independent of the Microsoft C model, thus, the `cltiga` batch file does not specify any particular model and uses the default (small) model unless overridden. This can be done by setting the `cl` environment variable (consult the Microsoft C reference manual for details).

2.6.3 mg2tiga Utility

The mg2tiga utility converts fonts compatible with the TMS340 Math/Graphics function library to a format compatible with the TIGA text functions. The command line syntax for mg2tiga.exe is

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

where:

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 31 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 containsTI Roman fonts. As supplied, this library is called ti_roman.lib and contains 12 fonts. A table of contents of this library:

```
gspar -t ti_roman
```

```
GSP Archiver          Version 3.00
(c) Copyright 1985, 1988, 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 31 characters long and should be the name of the font.

Example:

```
MGFL to TIGA font converter
[ Converting: ti_rom18.obj -> roman18.fnt ]
Enter facename (31 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 entering 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: 00000b
facename: TI ROMAN
first: 0000
last: 00ff
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 ]->
```

2.6.4 `tigamode` Utility

The `tigamode` utility allows you to interrogate the operating modes of the TIGA-compatible graphics board in your system. It also enables you to select a default mode for your board. For additional information and complete source for the `tigamode` utility, consult the `\tigapgms\tigamode` directory.

2.7 Non-TIGA Development Utility Programs

TIGA development and/or porting relies heavily on the use of the Microsoft's Macro Assembler and/or Microsoft's C compiler packages. Specifically, both packages must be version 5.0 or later to ensure compatibility with TIGA.

TIGA contains batch files that aid in rebuilding the TIGA communications driver and graphics manager (if you have the SPK), and batch files to compile and link your applications with the TIGA applications interface. These batch files use the following utility programs, which are bundled with the indicated Microsoft programming package:

Utility program	Description
`cl.exe`	Microsoft C compiler shell program
`lib.exe`	MS-DOS library manager, supplied with DOS
`link.exe`	MS-DOS linker, supplied with both packages
`make.exe`	Make utility, supplied with both packages

These programs must be accessible from TIGA's batch files (via the PATH environment variable) to ensure proper operation.

2.8 TIGA Syntax and Programming Examples

The definition of the TIGA syntax and the coding examples supplied in this guide are written primarily in Microsoft C, since Microsoft C is commonly used to write DOS applications. The high-level language syntax in C is similar to that of many other high-level languages and even programmers unfamiliar with C will understand the syntax. TIGA, however is not restricted to Microsoft C, although this initial release does rely on the Microsoft C calling conventions. Future versions of TIGA may be developed for other C compilers and other languages.

The current version of TIGA also provides a simple interface for Microsoft assembly language programmers. The three TIGA include files (`tiga.h`, `extend.h`, and `typedefs.h`) all have assembly language equivalents that use macros to provide a simple interface to call TIGA functions. An example of how to call TIGA functions from the assembler is provided in the `\tigapgms\asmtst` directory. Consult the `readme.1st` file in that directory for more details on the use of macros. Also provided in the include files are macros used to simplify the definition of user extensions. These macros, which mirror their C counterparts, are described in Chapter 4 of this user's guide.

Chapter 3

TIGA Application Interface

The following sections list the TIGA primitives, first in their functional groups, and then alphabetically, with a description and example of their use. Appendix B provides further information concerning the drawing functions.

This chapter describes the TIGA application interface, including the following topics:

Section		Page
3.1	Base Set of TIGA Primitives	3-2
3.2	Summary Table of Functions by Functional Group	3-3
3.3	Alphabetical List of Functions	3-13

3.1 Base Set of TIGA Primitives

From an application programmer's point of view, TIGA (Texas Instruments Graphics Architecture) consists of a set of functions which the application can invoke to perform graphics-related operations. These functions may run entirely on the host PC, on the TMS340 board, or on both. They fall into two classes, core and extended:

- **Core Primitives**

  The core primitives, such as **set_palet**, are always available during a TIGA session.

- **Extended Primitives**

  The extended primitives, such as **fill_rect**, consist primarily of drawing functions. They are extended because the application has to load them when initializing TIGA. If the application requires installation of its own drawing functions, with parameters different from those used in TIGA, the extended primitives can be excluded at initialization time. Excluding the extended primitives frees up memory on the TMS340 board for user extensions.

3.2 Summary Table of Functions by Functional Group

3.2.1 Graphics System Initialization Functions

Function	Description	Type
cd_is_alive	Return if TIGACD is running	Core
function_implemented	Return if a function is implemented	Core
get_config	Return board configuration	Core
get_modeinfo	Return board configuration	Core
get_videomode	Return current emulation mode	Core
gsp_execute	Execute a COFF program	Core
install_primitives	Install extended primitives	Core
install_usererror	Install user error handler	Core
loadcoff	Load a COFF program	Core
set_config	Set graphics configuration	Core
set_timeout	Set timeout timing value	Core
set_videomode	Set emulation mode	Core
synchronize	Make host wait for GSP to idle	Core

The graphics system initialization functions deal with the initialization of the TIGA environment. Every application must call **set_videomode** with a `TIGA` argument prior to invoking any other TIGA function. There are different styles of initialization that can be performed and they are detailed in the description of **set_videomode**. Similarly, on exit, every TIGA function must call **set_videomode** with a mode of `PREVIOUS` or some other IBM emulation mode. If the TMS340 board also serves as the primary graphics adapter, **set_videomode** switches the TMS340 board back into emulation mode prior to returning to DOS.

To use the TIGA extended primitives, the application must first call the **install_primitives** function. This call needs to be done only once, if the dynamic heap pool on the TMS340 board has not been reinitialized.

The **get_config** function returns the current board configuration (resolution, pixel size etc) to the application. Typically a board can be configured in more than one mode and the **get_modeinfo**, **set_config** functions are provided to inquire and select alternate modes. Typically an application uses whatever mode the board is set up in (via **get_config**). The end user can swap between different modes with the TIGAMODE utility (described in Chapter 2). That utility makes calls to the **get_modeinfo** and **set_config** functions.

TIGA consists of two distinct parts: a host part (the communication driver), which takes data from the application and sends it to one of the communication buffers accessible to the TMS340, and a TMS340 part (the graphics manager), which takes the data from the host and performs the graphics op-

eration. The processors in these two portions work asynchronously. Sometimes the host portion has to wait for the TMS340 portion; for example: when the TMS340 returns data to the host or when the host waits for a free communication buffer to use. The time the host has to wait for the TMS340 depends on what the TMS340 is doing for the application.

TIGA has a built-in default time of 5 seconds, after which it displays an error message on the screen to inform the application that there was no communication with the TMS340. This allows the user to break out if there is an error, or to continue waiting. If this default time is not acceptable it can be changed using the **set_timeout** function. The application can trap error calls by installing its own error handler function using **install_usererror** . There is an example of this after the description of **install_usererror**.

Sometimes it is desirable to force the host to wait for the TMS340 to complete an operation before proceeding. Consider the case where the TMS340 is performing a **bitblt** of data into memory and the host is then to upload the data using **gsp2hostxy**. Because the host and TMS340 work asynchronously, it is possible for the **gsp2hostxy** function to start copying data before the **bitblt** has completed. This can be solved by the application calling the **synchronize** function, which causes the host to wait until all TMS340 functions are completed, prior to the call to upload the data.

3.2.2 Clear Functions

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

The clear functions provide different ways to clear the screen. They all attempt to utilize any special memory functions (such as shift-register transfers) that the board or the memory chips themselves may have to perform the clear function as quickly as possible. The **clear_frame_buffer** clears the entire frame buffer, which may consist of multiple display pages and areas of offscreen (non-displayable) memory.

The **clear_screen** function clears only the visible portion of the screen. For configurations that contain multiple display pages, only the current drawing page is cleared. The **clear_page** function is similar to the **clear_screen** function except that it does not ensure that data in offscreen memory is kept intact. This is because, depending on how the frame buffer is designed, it may be possible to clear a page using shift-register transfers; however, certain offscreen memory areas may also be affected. Thus, to keep offscreen memory integrity, an application should use the **clear_screen** function. If

offscreen memory integrity is not a concern, the **clear_page** function may be the fastest option on certain boards.

3.2.3 Graphics Attribute Control Functions

Function	Description	Type
cpw	Compare point to window	Core
get_colors	Return foreground/background colors	Core
get_env	Return current environment structure	Core
get_pmask	Return color plane mask	Core
get_ppop	Return pixel processing operation	Core
get_transp	Return transparency mode	Core
get_windowing	Return windowing 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 description	Ext
set_pensize	Set current pensize	Ext
set_pmask	Set color plane mask	Core
set_ppop	Set pixel processing operation	Core
set_transp	Set transparency mode	Core
set_windowing	Set windowing mode	Core
transp_off	Disable pixel transparency	Core
transp_on	Enable pixel transparency	Core

The graphics output functions use implied operations called the graphics attributes to perform the drawing operations described below. These attributes are initialized and can be queried using the functions in this group. The graphics attributes consist of

Foreground Color Primary drawing color of all primitives. The color value is an index running from 0 to two-to-the-power-of-the-current-pixel-size. The value will typically be an index into the current palette (see next section).

Background Color Secondary drawing color, used in patterns, text and bitblt functions.

Plane Mask Enables the bits in a pixel to represent different planes.

Pixel Processing Determines the operation performed on source and destination pixels in any pixel operation.

Transparency Determines whether a pixel write should be inhibited if the pixel color is transparent.

Windowing Allows regions of the screen to be clipped to ensure no drawing occurs outside the designated window.

Drawing Origin By default the drawing origin is the top-left hand corner of the screen, but this can be moved anywhere.

Fill Pattern Used by all the **patn** drawing functions that fill a region with a pattern instead of a solid color.

Drawing Pen Used by all the **pen** drawing functions to outline a region with a pen of the specified width and height, rather than with a single- pixel wide line.

For further details concerning the drawing functions, see Appendix B.

3.2.4 Palette Functions

Function	Description	Type
get_nearest_color	Return nearest color in a palette	Core
get_palet	Return an entire palette	Core
get_palet_entry	Return a palette entry	Core
init_palet	Initialize default palette	Core
set_palet	Set an entire palette	Core
set_palet_entry	Set a palette entry	Core

The palette functions are graphics attributes and deserve special attention because they may vary from board to board. Ideally, the application should be able to set the palette to any particular desired value, but if the palette is in ROM, this is not possible. Use **function_implemented** to determine if the palette entries can be set. If they cannot be set, use the **get_nearest_color** function to find the best entry to the desired color stored in ROM. Also, different palettes allow different bits-per-gun. Determine the bits-per-gun using the `palet_gun_depth` field in the CONFIG structure or the **get_palet** function. The **get_palet** and **get_palet_entry** functions return the physical colors stored in the palette. Thus, if a palette entry is set with 8 bits of red to hexadecimal FF on a particular board (such as the TI SDB), where only 4 bits per gun are used, invoking **get_palet_entry** for that entry would return a red value of F0 to indicate that the LS 4 bits were ignored. An example describing this is shown after the description of **get_palet_entry**.

If possible, the palette is initialized to a default set of commonly-used colors (defined in the `TIGA.H` insert file) by a call to **init_palet**.

3.2.5 Graphics Output Functions

Function	Description	Type
draw_line	Draw line	Ext
draw_oval	Draw ellipse outline	Ext
draw_ovalarc	Draw ellipse arc	Ext
draw_piearc	Draw ellipse pie slice	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
frame_oval	Draw oval border	Ext
frame_rect	Draw rectangular border	Ext
patnfill_convex	Draw patterned convex polygon	Ext
patnfill_oval	Draw patterned ellipse	Ext
patnfill_piearc	Draw patterned pie slice	Ext
patnfill_polygon	Draw patterned polygon	Ext
patnfill_rect	Draw patterned rectangle	Ext
patnframe_oval	Draw patterned oval border	Ext
patnframe_rect	Draw patterned rectangular border	Ext
patnpen_line	Draw line with pattern and pen	Ext
patnpen_ovalarc	Draw oval arc with pattern and pen	Ext
patnpen_piearc	Draw pie slice with pattern and pen	Ext
patnpen_point	Draw pixel with pattern and pen	Ext
patnpen_polyline	Draw lines with pattern and pen	Ext
pen_line	Draw line with pen	Ext
pen_ovalarc	Draw an oval arc with pen	Ext
pen_piearc	Draw pie slice with pen	Ext
pen_point	Draw point with pen	Ext
pen_polyline	Draw lines with pen	Ext
seed_fill	Fill region with color	Ext
seed_patnfill	Fill region with pattern	Ext
styled_line	Draw styled line	Ext

The graphics output functions are self-explanatory. Specific examples on how to use fill patterns are shown in **patnfill_piearc** and for drawing pens in **patnpen_line**. These examples also explain other **patn** and **pen** functions. Additional examples of drawing functions are given for **draw_line**, **draw_oval**, **draw_ovalarc**, **draw_point**, **draw_polyline**, **fill_polygon**, and **styled_line.**

For further details concerning the drawing functions, see Appendix B.

3.2.6 Poly Drawing Functions

Function	Description	Type
draw_polyline	Draw polyline	Ext
fill_convex	Fill 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

The TIGA communication driver functions pass the arguments of all the TIGA primitives into a communication buffer for the TIGA graphics manager to use. Nearly all TIGA primitives have fixed size arguments that fit easily into the communication buffer. This is not the case with the poly drawing functions, 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), that use a buffer, allocated from the dynamic heap pool, to store the data. However, these alternate entry points are slower.

3.2.7 Workspace Functions

Function	Description	Type
fill_polygon	Fill polygon	Ext
get_wksp	Return offscreen workspace	Core
patnfill_polygon	Pattern fill polygon	Ext
set_wksp	Set a temporary workspace	Core

The polygon fill functions use an implied operand of a temporary workspace. This workspace is a 1 bit-per-pixel representation of the display screen and may be allocated from offscreen-memory in the TIGA port for the board being used. This can be determined by the **get_wksp** function. If the area cannot be allocated from offscreen-memory, then it must be allocated from heap and assigned using the **set_wksp** function. An example showing how to do this follows the **fill_polygon** description.

3.2.8 Pixel Array Functions

Function	Description	Type
bitblt	Bitblt source array to destination	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

The **bitblt** function transfers a rectangular array of pixels in TMS340 memory. The function uses two implied operands (source bitmap and destination bitmap) which are set, by default, to the screen. When they are set to a linear address, the **bitblt** function can then save data offscreen and restore it again (see example following **set_dstbm**). The **bitblt** function can also be used to expand a linear bitmap which is at 1 bit-per-pixel to a color bitmap (see example following **zoom_rect**).

The destination bitmap is an implied operand for all drawing functions. If it is set to anything other than the screen, all drawing primitives (other than **bitblt**) abort. In the future, linear drawing capability may be added to each of the drawing functions so they can draw into a linear bitmap.

The source bitmap is ignored by all functions except **bitblt** and **zoom_rect**. The latter is used to scale a source pixel array into any size destination array. See the example following the function description.

3.2.9 Text Functions

Function	Description	Type
delete_font	Remove a font from the font table	Ext
get_fontinfo	Return font physical information	Core
get_textattr	Return text rendering attributes	Ext
init_text	Initialize text drawing environment	Core
install_font	Install font into font table	Ext
select_font	Select an installed font for use	Ext
set_textattr	Set text rendering attributes	Ext
text_out	Render an ASCII string	Core
text_width	Return the width of an ASCII string	Ext

The text functions are partly core and partly extended primitives. In the core is a system font and the **text_out** primitive along with **get_fontinfo** (which returns font size information) and **init_text** (to reset the text environment). The extended primitives **install_font**, **select_font**, and **delete_font** allow the addition of TIGA fonts. There are over 100 TIGA fonts available, which the application loads from disk on the host side (or links-in with the host application) and downloads onto the TMS340 side. An example illustrating this is in **install_font**. The **set/get_textattr** function allows the text attributes such as inter-character spacing, to be adjusted by the application.

For more details on the font structure, see Appendix A.

3.2.10 Cursor Functions

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

The cursor functions support the graphics cursor routine. The cursor can be enabled or disabled via the **set_curs_state** function and its position can be modified using **set_curs_xy**. The **set_curs_shape** enables an arbitrary shaped cursor to be used. An example showing how the cursor may be driven by the host mouse follows the description of **set_curs_shape**.

3.2.11 Graphics Utility Functions

Function	Description	Type
get_pixel	Read contents of a pixel	Ext
lmo	Return left-most-one bit number	Core
page_busy	Return status of page flipping	Core
page_flip	Set display and drawing pages	Core
peek_breg	Read from a B-file register	Core
poke_breg	Write to a B-file register	Core
rmo	Return right-most-one bit number	Core
wait_scan	Wait for a designated scan-line	Core

The graphics utility functions are a group of miscellaneous graphics-related functions, most of which require no explanation other than what is given with the individual functions.

The **wait_scan** and **page_flip/busy** functions are mechanisms to aid animated sequences. **wait_scan** waits for a particular scan line, enabling data to be drawn on one part of the screen while a different part is being displayed. This feature provides a flicker-free display but is limited in the amount that it can draw before the display moves to the area that is being drawn into. The use of multiple drawing pages is a much more effective way where one page can be drawn while another is being displayed. This is the only way to ensure a flicker-free display. However, it does require at least double the amount of memory for the frame buffer.

The CONFIG structure supplies the number of display pages. If the number of pages is greater than 1, use the **page_flip** function to select a page for drawing and a page for displaying. The flip of the pages is synchronized with the start of VBLNK for the best visual effect. The **page_busy** function must be polled prior to drawing into the new page.

3.2.12 Pointer-Based Memory Management Functions

Function	Description	Type
get_offscreen_memory	Return offscreen memory blocks	Core
gsp2gsp	Copy from GSP memory to GSP memory	Core
gsp_calloc	Allocate and clear GSP memory	Core
gsp_free	Deallocate GSP memory	Core
gsp_malloc	Allocate GSP memory	Core
gsp_maxheap	Return largest free block	Core
gsp_minit	Reinitialize GSP memory heap pool	Core
gsp_realloc	Resize allocated block of memory	Core

The heap management functions (**gsp_malloc**, **gsp_free**, **gsp_calloc**, **gsp_realloc**) are familiar to C application programmers. Identical heap management for host memory is provided in Microsoft C runtime support. The **gsp_minit** function initializes the heap pool (freeing all allocated pointers). Because the program memory is shared between stack and heap, this function can also be used to adjust the memory used for stack (and thus, leaving the remainder memory for heap). The **gsp_maxheap** function returns the largest contiguous block of heap available (which is also the total heap size at the start of an application) and can determine how much data can be loaded from the host to the TMS340.

The **gsp2gsp** function provides a memory copy function (as opposed to **bitblt,** which is a pixel array copy). It handles overlapping memory areas.

The **get_offscreen_memory** function returns data concerning the size and addresses of available offscreen memory, which are separate from the heap pool. Allocation of these areas is performed by the application. The offscreen memory may be used for the temporary workspace (see Section 3.2.7).

3.2.13 Communication Functions

Function	Description	Type
cop2gsp	Copy coprocessor to GSP memory	Core
field_extract	Extract data from GSP memory	Core
field_insert	Insert data into GSP memory	Core
get_vector	Get address at a TMS340 trap vector	Core
gsp2cop	Copy GSP memory to coprocessor	Core
gsp2host	Copy from GSP into host memory	Core
gsp2hostxy	Copy rectangular area from GSP to host	Core
host2gsp	Copy from host into GSP memory	Core
host2gspxy	Copy rectangular area from host to GSP	Core
set_vector	Set contents of GSP trap vector	Core

The communication functions transfer data between host and TMS340 memory spaces, or between TMS340 and its coprocessor.

3.2.14 Extensibility Functions

Function	Description	Type
create_alm	Create absolute load module	Core
create_esym	Create external symbol table file	Core
flush_esym	Flush external symbol table file	Core
flush_extended	Flush all user extensions	Core
get_isr_priorities	Return interrupt service routine priorities	Core
install_alm	Install absolute load module	Core
install_primitives	Install extended drawing primitives	Core
install_rlm	Install relocatable load module	Core
set_interrupt	Set an interrupt handler	Core

The extensibility functions are all concerned with extensibility. Their use is described in detail in Chapter 4.

3.3 Alphabetical List of Functions

This section contains a reference for TIGA functions in alphabetical order. 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.

Syntax

```
void  bitblt(width, height, srcx, srcy, dstx, dsty)
    short width;
    short height;
    short srcx;
    short srcy;
    short dstx;
    short dsty;
```

Type Extended

Description The **bitblt** function copies data from the TMS340's source bitmap, which is installed by the **set_srcbm** function into the destination bitmap. The destination bitmap is in turn installed by the **set_dstbm** function. The bitmap data is a rectangular area whose top left-hand corner is at coordinates (`srcx, srcy`) in the source bitmap of size `width` by `height`, into the destination bitmap, starting at coordinates (`dstx, dsty`). The pixel size of the two bitmaps should either be equal or, if they are not, one of the pixels sizes must be equal to 1.

If the pixel sizes are equal, then the rectangular area is copied. If both source and destination bitmaps are set to the screen, then a check is made to see if the areas overlap. If they do, the **bitblt** direction is set to avoid destroying the source bitmap before it is copied.

If the source bitmap pixel size is 1, the bitmap is expanded to color in the destination array. 1s in the source bitmap are drawn in the current foreground color and 0s are drawn in the current background color.

If the destination bitmap pixel size is equal to 1, then a contract function is performed. Pixels in the source array that are equal to the current background color are set to 0 in the destination array. All other colors are set to 1.

When the destination bitmap is set to the screen, the function attempts to clip the destination bitmap to the current clipping rectangle set by the **set_clip_rect** function. This only occurs if the pitch of the source and destination pitch are a power of two (greater than or equal to 16). The source pitch is set by the user in the **set_srcbm** function. If the destination bitmap is the screen, its pitch is defined in the `disp_pitch` field of the CONFIG structure (see Appendix A). If the pitches are not a power of two, you **must** pre-clip the destination bitmap.

Example

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

CONFIG config;

main()
{
  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      set_fcolor(BLUE);
      /* fill the top left quarter of the screen          */
      /* with a blue rectangle                            */
      fill_rect(config.mode.disp_hres>>1,
                config.mode.disp_vres>>1, 0, 0);
      /* copy one quarter of the rectangle with top       */
      /* left-hand corner at the center of the screen     */
      bitblt (config.mode.disp_hres>>2,
              config.mode.disp_vres>>2, 0, 0,
              config.mode.disp_hres>>1,
              config.mode.disp_vres>>1);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax `int cd_is_alive()`

Type Core

Description The **cd_is_alive** returns true (nonzero) if the communication driver (CD) has been installed, or false (zero), otherwise. This function is an alternate entry point to **set_videomode** for applications that require only the host-side entry points of TIGA without loading the graphics manager. If this function returns true (nonzero), then the following host-only primitives may be used:

- **field_extract**
- **field_insert**
- **gsp2host**
- **gsp2hostxy**
- **gsp_execute**
- **host2gsp**
- **host2gspxy**
- **loadcoff**

Before a call to **any other** TIGA function, a call must first be made to **set_videomode** with a `TIGA` parameter.

Syntax

```
void clear_frame_buffer(color)
    long color;
```

Type Core

Description The **clear_frame_buffer** function performs a rapid clearing of the entire display memory, by setting it to the color index specified. If the color is set to –1 the current background color is used. If the configuration is such that the screen can be cleared using shift-register transfers, this is done, providing a very rapid clearing. If the configuration contains multiple display pages **all** pages are cleared. The integrity of offscreen data cannot be guaranteed using this function. If this is of concern to the calling program, then the **clear_screen** function should be used.

If the graphics display board uses display memory to store palette information (as in the TMS34070), this area should be left intact by this function.

Syntax

```
void clear_page(color)
   long color;
```

Type

Core

Description

The **clear_page** function performs a rapid clearing of the current drawing page by setting it to the color index specified. If the color is set to –1 the current background color is used. It provides very rapid clearing if the configuration allows the use of shift-register transfers. If the configuration contains multiple display pages, only the current drawing page is cleared. However, the integrity of offscreen data cannot be guaranteed using this function. If this is of concern to the calling program, then the **clear_screen** function should be used.

If the graphics display board uses display memory to store palette information (as in the TMS34070), this area should be left intact by this function.

Syntax

```
void clear_screen(color)
    long color;
```

Type Core

Description The **clear_screen** function performs a rapid clearing of only the **visible** portion of the display memory, by setting it to the color specified index. If the color is set to –1, the current background color is used. It provides very rapid clearing if the configuration allows the use of shift-register transfers. However, this function clears only the current drawing page (in a multiple paged frame buffer) and does not affect any offscreen memory. If the offscreen memory data does not have to be conserved, then a more rapid fill may be possible using the **clear_page** function.

If the graphics display board uses display memory to store palette information (as in the TMS34070), this area should be left intact by this function.

Example See **function_implemented**.

Syntax

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

Type

Core

Description

The **cop2gsp** function copies data from the address space of the coprocessor with ID `copid` (a number from 0—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

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

Type

Core

Description

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

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

0000_2	if the point lies within the window.
$01xx_2$	if the point lies above the window.
$10xx_2$	if the point lies below the window.
$xx01_2$	if the point lies left of the window.
$xx10_2$	if the point lies right of the window.

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

Syntax

```
int create_alm(rlm_name, alm_name)
   char far *rlm_name;
   char far *alm_name;
```

Type Core

Description 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 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 Chapter 4.

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

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in the main `TIGA` directory, either the `TIGA` environment variable `-m` option is not set or that directory does not contain `TIGALNK.EXE`.
–3	**Out of Memory** – Not enough host memory to run `TIGALNK` or not enough TMS340 memory to store the ALM
–4	**Communication Driver not Running** – Run `TIGACD`
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`
–6	**Missing RLM** – 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.
–7	**Symbol File Error** – I/O error obtained in accessing the symbol file. The `-m` option of the TIGA environment variable is not set or the directory does not contain `TIGA340.SYM` or the file is corrupt. In the latter case recopy this file from your backup disk.
–10	**Symbol Reference** – An unresolved symbol was referenced by the RLM. Determine the name 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)
–12	**Symbol Table Mismatch** – The modules installed in the symbol table do not match those the TIGA graphics manager has installed. Reinitialize the modules by a call to **flush_extended** and install them again.

Syntax

```
int create_esym(gm_name)
   char far *gm_name;
```

Type Core

Description The **create_esym** function does not need to be called by the user. It is provided as a procedural level interface to the linking loader. It should be used instead of calling the linking loader directly, to provide compatibility with future versions of TIGA.

This function creates an external symbol table file from the supplied TIGA graphics manager file. If no file extension is supplied, then a file extension of `.OUT` is used. The external symbol table is saved under the name `TIGA340.SYM`. After creation, this file will contain only the global (or external) symbols that were contained in the graphics manager file. During subsequent installation of relocatable load modules, this file is used to resolve external references. Also any external symbols contained in a RLM are added to this file during the installation process so those symbols can be referenced by other RLMs.

If no pathname information is supplied for the `gm_name`, this function uses the path specified by the TIGA environment variable. The external symbol file created also uses this path information. For more details on extensibility, the use of this function refer to Chapter 4.

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

This function returns these

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in the main `TIGA` directory, either the `TIGA` environment variable `-m` option is not set or that directory does not contain `TIGALNK.EXE`.
–4	**Communications Driver not Running** – Run `TIGACD`.
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`.
–7	**Symbol File Error** – I/O error obtained in accessing symbol file. The `-m` option of the TIGA environment variable is not set or the directory does not contain `TIGA340.SYM` or the file is corrupt. In the latter case recopy this file from your backup disk.
–11	**COFF File not Absolute** – The `COFF` file argument for this function is not linked to an absolute address.

Error Code	Description (continued)
–12	**Symbol Table Mismatch** – The modules installed in the symbol table do not match those the TIGA graphics manager has installed. Reinitialize the modules by a call to **flush_extended** and install them again.

Syntax

```
int delete_font(id)
    short id;
```

Type Extended

Description The **delete_font** function removes from the font table the installed font referenced by `id`. A nonzero value is returned if the font was successfully removed, and a value of zero if the font was not installed. Note that if the font removed was also the one selected for current text drawing operations, the system OEM font is selected.

Syntax

```
void draw_line(x1, y1, x2, y2)
    short x1, y1;   /* start coordinates                    */
    short x2, y2;   /* end coordinates                      */
```

Type Extended

Description The **draw_line** function uses Bresenham's algorithm to draw a line from the starting point to the ending point. `x1` and `y1` specify the starting coordinates; `x2` and `y2` specify the ending coordinates. The line is one pixel thick and is drawn in the current foreground color.

Example

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

CONFIG config;

main()
{
  short xs, ys, xe, ye, i, color;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      xs = config.mode.disp_hres>>1;
      ys = config.mode.disp_vres>>1;
      set_fcolor(RED);
      /* set up an add pixel processing option to affect  */
      /* overlapping lines in the center of the screen    */
      set_ppop(16);
      /* draw lines at diff. orientations                 */
      for (xe = 5, ye = 5; xe <= config.mode.disp_hres-6;
           xe += 17)
           draw_line(xs, ys, xe, ye);
      for (xe = 5, ye = config.mode.disp_vres-6;
           xe <= config.mode.disp_hres-6; xe += 17)
           draw_line(xs, ys, xe, ye);
      for (xe = 5, ye = 10; ye <= config.mode.disp_vres-6;
           ye += 17)
           draw_line(xs, ys, xe, ye);
      for (xe = config.mode.disp_hres-6, ye = 10;
           ye <= config.mode.disp_vres-6; ye += 17)
           draw_line(xs, ys, xe, ye);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
void draw_oval(w, h, xleft, ytop)
   short w, h;          /* width, height of rect.          */
   short xleft, ytop;   /* coordinates at top left corner  */
```

Type

Extended

Description

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

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

The outline is one pixel thick and is drawn in the current foreground color.

Example

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

CONFIG config;

main()
{
  int w, h, x, y;
  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      /* restrict drawing to window in center of screen   */
      set_clip_rect(config.mode.disp_hres>>1,
                    config.mode.disp_vres>>1,
                    config.mode.disp_hres>>2,
                    config.mode.disp_vres>>2);
      set_fcolor(GREEN);
      /* draw various sizes ellipses                      */
      for (w = 0, x = 4; w < config.mode.disp_hres/20;
           ++w, x += w + 3)
        for (h = 0, y = 4; h < config.mode.disp_vres/20;
             ++h, y += h + 3)
          draw_oval(w, h, x, y);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type Extended

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 arc is one pixel thick and is drawn in the current foreground color.

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

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

The last two arguments define the limits of the arc:

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

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

Example

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

CONFIG config;

/* x coordinate of screen center */
#define XC  (config.mode.disp_hres>>1)
/* y coordinate of screen center */
#define YC  (config.mode.disp_vres>>1)
/* x coordinate of screen limit  */
#define XMAX  (config.mode.disp_hres-4)
/* y coordinate of screen limit  */
#define YMAX  (config.mode.disp_vres-4)
/* x increment */
#define DX  (config.mode.disp_hres/40)
/* y increment */
#define DY  (config.mode.disp_vres/40)
#define MAXBYTES  2048

main()
{
  short w, h;
  PTR   addr;
  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      /* draw a spiral */
      set_fcolor(YELLOW);
      for (w = XMAX, h = YMAX; w > DX; h -= DY)
      {
        draw_ovalarc (w, h, XC-w/2, YC-h/2, 0, 270);
        w -= DX;
        draw_ovalarc (w, h, XC-w/2, YC-h/2, 270, 90);
      }
      addr = gsp_malloc(MAXBYTES);
      set_fcolor(GREEN);
      seed_fill(XC, YC, addr, MAXBYTES);
      gsp_free(addr);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
void draw_piearc(w, h, xleft, ytop, theta, arc)
    short w, h;          /* width and height                */
    short xleft, ytop;   /* top left corner                 */
    short theta;         /* starting angle (degrees)        */
    short arc;           /* angle extend (degrees)          */
```

Type Extended

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 arc and the two lines are all one pixel thick and are drawn in the current foreground color.

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

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

The last two arguments define the limits of the arc:

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

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

Example See **patnfill_piearc**

Syntax

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

Type Extended

Description The **draw_point** function draws a single pixel. Arguments `x` and `y` give the XY coordinates of the designated pixel. The pixel is drawn in the current foreground color.

Example

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

CONFIG config;

main()
{
  int i, x, y, xy, yx;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      set_fcolor(CYAN);
      x = xy = 0;
      y = yx = config.mode.disp_vres>>1;
      /* draw Lissajous pattern in dots                    */
      for (i = 1200; i > 0; --i)
      {
        draw_point(x+(config.mode.disp_hres>>1),
                   y+(config.mode.disp_vres>>1));
        x += yx >> 4;
        yx -= x >> 4;
        y += xy >> 5;
        xy -= y >> 5;
      }
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
typedef struct
{
    short  x;
    short  y;
}POINTS;

void draw_polyline(n, points);
    short  n;
    POINTS far *points;
```

Type Extended

Description The **draw_polyline** function draws `n` single pixel-wide lines whose endpoints are supplied in an array of structures, described in the syntax. Note that for the polygon drawn to be closed, the calling program must ensure that the first and last points are the same.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

The argument `points` can be of any length. The application can easily overflow the command buffer used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. The application must check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **draw_polyline_a** with the same parameterization is supplied to check the size of the data to be sent. If the command buffer overflows, **draw_polyline_a** attempts to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size of the data being transferred; however, the invocation of the function takes longer, because the length of the data must be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function).

Example

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

CONFIG config;

/* screen independent coordinates of a cube                    */
short far cube[] =
{
   -2,+1, -2,-2, -1,-1, -1,+2, -2,+1,
   -2,-2, +1,-2, +2,-1, -1,-1, -2,-2,
   -1,-1, +2,-1, +2,+2, -1,+2, -1,-1,
};

main()
{
  int   dx, dy, i;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      set_fcolor(MAGENTA);
      dx = config.mode.disp_hres>>4;
      dy = config.mode.disp_vres>>4;
      /* scale cube coordinates to fit the resolution     */
      for (i = 0; i < sizeof(cube)/sizeof(short); )
      {
        cube[i++] *= dx;
        cube[i++] *= dy;
      }
      /* move draw origin to the center of the screen     */
      set_draw_origin(config.mode.disp_hres>>1,
                      config.mode.disp_vres>>1);
      /* draw the outline of the cube */
      draw_polyline((sizeof(cube)/sizeof(short))>>1,cube);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type

Extended

Description

The **draw_rect** function draws the outline of a rectangle. The first four arguments define the rectangle:

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

The outline is one pixel wide and is drawn in the current foreground color.

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);
```

Syntax

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

Type

Core

Description

The **field_extract** function returns the 32-bit, zero-extended data from the TMS340 memory address specified by `gptr`. The field size parameter `fs` must be between 1 and 32 inclusive and specifies the number of bits to read from TMS340 memory. There are no restrictions on the alignment of the TMS340 address.

Syntax

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

Type Core

Description The **field_insert** function writes the value of `data` into the TMS340 memory specified by `gptr`. The field size parameter `fs` must be between 1 and 32 inclusive and specifies the number of bits to be written. Bit 0 (the least significant bit) of `data` is written first, followed by bit 1 and so on until the specified number of bits have been written. There are no restrictions as to the alignment of the TMS340 address.

Syntax

```
typedef struct
{
   short  x;
   short  y;
}POINTS;

void fill_convex(n, points);
   short  n;
   POINTS far *points;
```

Type Extended

Description The **fill_convex** function fills a convex polygon, given a list of points representing the vertices. To be drawn correctly, the polygon must have at least three vertices visible. The first and last points must be the same to ensure that the polygon is closed. The polygon is solid-filled with the current foreground color.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

fill_convex 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 **fill_convex** function automatically culls back faces to support 3-D applications. A polygon is drawn only if its front side is visible, that is, if it is facing toward the screen. If the vertices are specified in counterclockwise order, the polygon is assumed to be facing away from the screen and is therefore not drawn.

The back face test is done by first comparing vertices n –2, n –1, and 0 to determine whether the polygon vertices are specified in clockwise (front face) or counterclockwise (back face) order. This test 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 not colinear. If all the vertices are colinear, the polygon is invisible.

One of the parameters of **fill_convex** is a list of points that can be of any length. The application can easily overflow the command buffer used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. The application must check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **fill_convex_a,** with the same parameterization, is supplied to check the size of the data to be sent. If the command buffer overflows, **fill_convex_a** attempts to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size of the data being transferred; the invocation of the function takes longer, because the length of the data must be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function).

Syntax

```
void fill_oval(w, h, xleft, ytop)
   short w, h;          /* width, height of rect.          */
   short xleft, ytop;   /* coordinates of top left corner  */
```

Type Extended

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

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

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

Example See **draw_oval**

Syntax

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

Type Extended

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

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

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

The last two arguments define the limits of the arc:

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

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

Example See **patnfill_piearc**

Syntax

```
typedef struct
{
   short x;
   short y;
}POINTS;

void fill_polygon(n, points);
   short n;
   POINTS far *points;
```

Type Extended

Description The **fill_polygon** function fills a polygon, given a list of endpoints of the polygon. No restrictions are placed on the shape of the polygons filled by this function: edges can cross each other, filled areas can contain holes, and two or more filled regions can be disconnected from each other. The polygon is solid-filled with the current foreground color. Note that the first and last points of the array should be the same to close the polygon.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

This function also takes as an implied argument a 1-bit representation of the frame buffer, which it uses as a temporary workspace. This workspace must be set up prior to invoking this function (via a call to the **set_wksp** function).

The argument `points` can be of any length. The application can easily overflow the command buffer used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. The application must check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **fill_polygon_a,** with the same parameterization, is supplied to check the size of the data to be sent. If the command buffer overflows **fill_polygon_a** attempts to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size of the data

being transferred; however, the invocation of the function takes longer because the length of the data has to be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function).

Example

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

CONFIG config;

/* screen independent coordinates of a cube                    */
short far cube1[] = {-2,+1, -2,-2, -1,-1, -1,+2, -2, +1};
short far cube2[] = {-2,-2, +1,-2, +2,-1, -1,-1, -2,-2,};
short far cube3[] = {-1,-1, +2,-1, +2,+2, -1,+2, -1,-1,};

main()
{
  PTR  wksp, pitch;
  int  dx, dy, i;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      /* see if wksp allocated                                 */
      get_config(&config);
      if (!get_wksp(&wksp, &pitch))
      {
        /* workspace not set up in offscreen memory,          */
        /*use malloc to create it                             */
        pitch = 1<<lmo(config.mode.disp_hres);
        if (pitch < config.mode.disp_hres) pitch <<= 1;
        wksp = gsp_malloc(((pitch * config.mode.disp_vres)
                           + 7) / 8);
        set_wksp(wksp, pitch);
      }
      dx = config.mode.disp_hres>>4;
      dy = config.mode.disp_vres>>4;
      /* scale cube coordinates to fit the resolution         */
      for (i = 0; i < sizeof(cube1)/sizeof(short); )
      {
        cube1[i++] *= dx;
        cube1[i--] *= dy;
        cube2[i++] *= dx;
        cube2[i--] *= dy;
        cube3[i++] *= dx;
        cube3[i++] *= dy;
      }
```

```
        /* move draw origin to the centre of the screen     */
        set_draw_origin(config.mode.disp_hres>>1,
                        config.mode.disp_vres>>1);
        /* fill in the sides of the cube                    */
        set_fcolor(BLUE);
        fill_polygon((sizeof(cube1)/sizeof(short))>>1,
                     cube1);
        set_fcolor(LIGHT_BLUE);
        fill_polygon((sizeof(cube2)/sizeof(short))>>1,
                     cube2);
        set_fcolor(CYAN);
        fill_polygon((sizeof(cube3)/sizeof(short))>>1,
                     cube3);
      }
      set_videomode(PREVIOUS, INIT);
    }
  }
```

Syntax

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

Type Extended

Description The **fill_rect** function draws a rectangle solid-filled with the current foreground color. The first four arguments define the rectangle:

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

Example See **bitblt**

Syntax `int flush_esym()`

Type Core

Description The **flush_esym** function does not need to be called by the application. It is a procedural level interface to the linking loader. It should be used instead of calling the linking loader directly to provide compatibility with future versions of TIGA.

This function flushes the external symbols from the symbol table `TIGA340.SYM`, leaving just the global symbols in this file.

For more details on extensibility and the use of this function, refer to Chapter 4.

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

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in the main TIGA directory, either the TIGA environment variable `-m` option is not set or that directory does not contain `TIGALNK.EXE`.
–4	**Communications Driver not Running** – Run `TIGACD`
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`
–7	**Symbol File Error** – I/O error obtained in accessing symbol file. The `-m` option of the TIGA environment variable is not set or the directory does not contain `TIGA340.SYM` or the file is corrupt. In the latter case, recopy this file from your backup disk.

Syntax `void flush_extended()`

Type Core

Description The **flush_extended** function performs two operations: first, it flushes the TIGA extended primitives and the installed user functions (both direct mode and C-packet) on the TMS340 side. Second, it removes the symbol table information stored on the host side. You can then install a new set of user functions.

For more details on extensibility and the use of this function, refer to Chapter 4.

Syntax

```
void frame_oval(w, h, xleft, ytop, dx, dy)
    short w, h;          /* width, height of rect.          */
    short xleft, ytop;   /* coordinates at top left corner  */
    short dx, dy;        /* width, height of frame border   */
```

Type Extended

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

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

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

The thickness of the frame in the X and Y dimensions is defined by two additional arguments:

- `dx` specifies the horizontal distance between the outer and inner ellipses.
- `dy` specifies the vertical distance between the outer and inner ellipses.

Example See **patnfill_piearc.**

Syntax

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

Type

Extended

Description

The **frame_rect** function solid fills a rectangular shaped frame with the current foreground color. The frame consists of a filled region between two concentric rectangles. The portion of the screen enclosed by the frame is not altered.

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

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

The thickness of the frame in the X and Y dimensions is defined by two additional arguments:

- `dx` specifies the horizontal distance between the outer and inner rectangles.
- `dy` specifies the vertical distance between the outer and inner rectangles.

Syntax

```
int function_implemented(function_code)
    short function_code;
```

Type Core

Description The **function_implemented** function queries whether a function is implemented or not. Functions in TIGA have an associated `function_code;` some 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
set_palet
get_palet
set_palet_entry
get_palet_entry
set_transp
gsp2cop
init_palet
```

The function codes themselves are contained in the main TIGA insert file, which contains 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 <typedefs.h>
#include <tiga.h>
#include <extend.h>

CONFIG config;

main()
{
  short green_index;
  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    /* 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);
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
void get_colors(fcolor, bcolor)
   short far *fcolor;
   short far *bcolor;
```

Type Core

Description The **get_colors** function returns both the foreground and background colors.

Syntax

```
/*-------------------------------------------------------------------*/
/* MODEINFO structure definition with the current mode of operation  */
/*-------------------------------------------------------------------*/

        typedef struct
        {
            long disp_pitch;
            short disp_vres;
            short disp_hres
            short screen_wide
            short screen_high;
            short disp_psize;
            long pixel_mask;
            short palet_gun_depth;
            long palet_size;
            short palet_inset;
            short num_pages;
            short num_offscrn_areas;
            long wksp_addr;
            long wksp_pitch;
        }MODEINFO;
/*-------------------------------------------------------------------*/
/* CONFIG structure definition of the current hardware configuration */
/*-------------------------------------------------------------------*/

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

        void get_config(config)
            CONFIG far *config;
```

Type Core

Description The **get_config** function returns a structure containing all board- and mode-specific information. Note that it is very likely that the structure described above will change in subsequent revisions. Therefore, it is recommended that the elements of the structure be referenced symbolically by their field name, rather than as an offset to the start of the structure. Insert files are available to do this.

The fields are as follows:

`version_number`	TIGA revision number, assigned by Texas Instruments.
`comm_buff_size`	Size, in bytes, of the communications buffer; application needs to ensure that the data sent does not overflow this buffer, for commands which do not check the size of the downloaded data.
`sys_flags`	Bits 0 — 7 indicate Floating Point Units (FPUs) are present, in order to be compatible with the TMS34020 Coprocessor ID codes. Bits 8—15 are reserved.
`device_rev`	This function invokes the TMS340's REV instruction and return its result here.
`num_modes`	Number of extended modes for boards that allow the switching between different display setups.
`current_mode`	Mode number corresponding to the current operating mode.
`program_mem_start`	Start address of program memory.
`program_mem_end`	End address of program memory.
`display_mem_start`	Start address of display memory.
`display_mem_end`	End address of display memory.
`stack_size`	Default stack size can be modified using **gsp_minit**.
`share_mem_size`	Size (in bytes) of shared memory that is available for the application to use.
`share_host_addr`	If share_size is nonzero, this is the start address in host memory of the shared memory; otherwise it is undefined.
`share_gsp_addr`	If share_size is nonzero, this is the start address in TMS340 memory of the shared memory; otherwise, it is undefined.

`disp_pitch`	Display pitch: linear difference between two scan lines in bits.
`disp_vres`	Vertical resolution in scan lines.
`disp_hres`	Horizontal resolution in pixels.
`screen_wide`	Contains the width of the monitor in centimeters. For systems where these dimensions are unknown, set to zero.
`screen_high`	Contains the height of the monitor in centimeters. For systems where these dimensions are unknown, set to zero.
`disp_psize`	Pixel size.
`pixel_mask`	Contains a mask of the bits used in a pixel. It normally contains the value of *2 to the power disp_psize minus 1*, indicating that every bit of pixel data is pertinent. On some boards the frame buffer may be arranged by 8 (disp_psize = 8) but with only 6 bits implemented. In that case pixel mask would contain the value 63 (hexadecimal 3F).
`palet_gun_depth`	Number of bits per gun in palette.
`palet_size`	Number of entries in the palette.
`palet_inset`	For most systems this field is set to 0. For TMS34070-based boards that store the palette in the frame buffer, this is the offset from the start of the scan line to the first pixel data.
`num_pages`	Number of display pages in multi-buffered systems.
`num_offscrn_areas`	This is the number of offscreen memory blocks available. If nonzero, then it is used to allocate space for the offscreen array, which can be obtained from the TMS340 via a call to the **get_offscreen_memory** function.
`wksp_addr`	Starting linear address in memory of offscreen workspace area.
`wksp_pitch`	Pitch of offscreen workspace area. If wksp_pitch=0, then no offscreen workspace is currently allocated.

Example See **draw_line.**

Syntax `int get_curs_state()`

Type Core

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

Example See cursor manipulation in **set_curs_shape.**

Syntax

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

Type Core

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

Example See cursor manipulation in **set_curs_shape.**

Syntax

```
typedef struct
{
    long    xyorigin;
    long    pensize;
    long    patnaddr;
    long    srcbm;
    long    dstbm;
    unsigned long stylemask
}ENVIRONMENT;

void get_env(env)
    ENVIRONMENT far *env;
```

Type Extended

Description The **get_env** function takes as its argument a pointer to the ENVIRONMENT structure containing the graphics environment variables. Although there are functions to manipulate these variables individually, if required, this function can be used to return the entire environment. Note that the structure described above may change in subsequent revisions. Therefore, it is recommended that the elements of the structure be referenced symbolically by their field name, rather than as an offset to the start of the structure. Insert files are available to do this.The fields of this structure are as follows:

`xyorigin`	Current drawing origin in y::x format set by **set_draw_origin**.
`pensize`	Current pen size arranged in y::x format, set by **set_pensize**.
`patnaddr`	TMS340 address of current pattern, set by **set_patnaddr**.
`srcbm`	TMS340 address of current source bitmap structure, set by **set_srcbm**.
`dstbm`	TMS340 address of current destination bitmap structure, set by **set_dstbm**.
`stylemask`	Current line style mask used by **styled_line** function.

Syntax

```
typedef struct
{
   char facename[32];
   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;   /* character width (0=proportional)   */
   short charhigh;   /* character height                   */
   short ascent;     /* ascent (how far above base line)   */
   short descent;    /* descent (how far below base line)  */
   short leading;    /* ldng. (row bottom to next row top) */
   long fontptr;     /* font address in GSP memory         */
   short id;         /* id of font (set at install time)   */
}FONTINFO;

int get_fontinfo(id, pFontInfo)
   short id;
   FONTINFO far *pFontInfo;
```

Type Core

Description The **get_fontinfo** function copies a structure of type FONTINFO, which describes the physical characteristics of the installed font referenced by `id` into the structure pointed to by `pFontInfo`. A nonzero value is returned if the structure is successfully copied; zero otherwise. An `id` of zero returns the FONTINFO structure for the system font, which does not need to be installed. If `id` is specified as –1, the FONTINFO of the currently selected font is returned.

Syntax

```
void get_isr_priorities(numisrs, ptr)
   short numisrs;     /* number of isrs                    */
   short far *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 calling this function, 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, refer to Chapter 4.

Syntax

```
int get_modeinfo(index, modeinfo)
    short index;
    MODEINFO far *modeinfo;
```

Type Core

Description The **get_modeinfo** function returns a structure containing a possible board configuration supported by the current board and monitor. The MODEINFO structure is described in detail in the **get_config** function. The index parameter is used to cycle through the different modes by setting it to 0, 1, 2, etc. It returns the MODEINFO structure for modes 0, 1, 2, etc. If an invalid index is entered, the function returns false (zero); otherwise, it returns true. The total number of possible modes can be found from the CONFIG structure using the **get_config** function.

Syntax

```
long get_nearest_color(r, g, b, i)
    char r;
    char g;
    char b;
    char i;
```

Type Core

Description The **get_nearest_color** function searches the current palette and detects the closest color value to that specified by the parameters. For a monochrome palette, it is simply the first index closest to `i`. For color palettes, the function is more complicated. Weighting values are given to each index that are the sum of the differences between the parameter `r` and the color's red value, and the difference between the parameter `g` and the color's green value, etc. Then the index with the smallest weight value is returned. See also Appendix B.7 for details on color selection.

Example See **function_implemented.**

Syntax

```
typedef struct
{
    long  addr;
    short xext;
    short yext;
}OFFSCREEN_AREA;

void get_offscreen_memory(num_blocks, offscreen)
    short num_blocks;
    OFFSCREEN_AREA far *offscreen;
```

Type Core

Description The **get_offscreen_memory** function returns the description of the offscreen memory blocks found in the system for the application to use. These blocks generally consist of display memory not being used either for the frame buffer or for an alternate page of frame buffer in multiple buffer systems. The number of blocks is in the structure returned in **get_config**. The application must reserve enough room for that amount of offscreen entries by first calling Microsoft's C **malloc**. The address returned by **malloc** should be submitted as the parameter to this function, as well as the number of blocks to be returned.

The structure returned consists of the start address of the TMS340 offscreen block, plus `xext` and `yext` in pixels.

These offscreen blocks can be used as temporary workspace for functions such as **set_wksp**, **seed_fill** and **zoom_rect**. Note that no memory management is performed on these blocks. The application must ensure the validity of the data stored there. Note also that the offscreen memory blocks are completely separate from those used by the memory management functions such as **gsp_malloc.**

If an offscreen memory block is used as the default offscreen workspace, it is guaranteed to be the first block returned via the **get_offscreen_memory** function.

Example

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

CONFIG config;

short arrow_shape[] =
{
  0x0003, 0x0007, 0x000F, 0x001F, 0x003F, 0x007F, 0x00FF, 0x01FF,
  0x03FF, 0x01FF, 0x007F, 0x00F7, 0x00F2, 0x01E0, 0x01E0, 0x00C0
};
#define  ARROW_W    16
#define  ARROW_H    16

main()
{
  int i;
  PTR arrow_addr;
  long arrow_size;
  OFFSCREEN_AREA far *offscreen;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      arrow_addr = 0;
      /* check if any offscreen areas are big enough      */
      if (config.mode.num_offscrn_areas)
      {
        /* malloc space in host memory to hold offscreen */
        /* structure                                      */
        offscreen = (OFFSCREEN_AREA *)malloc
        (config.mode.num_offscrn_areas
        *sizeof(OFFSCREEN_AREA));
        /* get the offscreen memory data structure        */
        get_offscreen_memory(config.mode.num_offscrn_areas,
                             offscreen);
```

```
            /* define size needed to store the arrow in bits */
            arrow_size = ARROW_W * ARROW_H;
            for (i = 0; (i < config.mode.num_offscrn_areas) &&
                 (!arrow_addr); i++)
            {
              if ((offscreen[i].xext * config.mode.disp_psize)
                    >arrow_size)
                    arrow_addr=offscreen[i].addr;
            }
          }
          /* if no available offscreen memory, use gsp_malloc */
          if (!arrow_addr)
                arrow_addr = gsp_malloc((arrow_size+7)/8);
          /* transfer shape data from host to gsp             */
          host2gsp((uchar far *) arrow_shape, arrow_addr,
                   (arrow_size+7)/8, 0);
          /* set the source bitmap to the arrow shape         */
          set_srcbm(arrow_addr, ARROW_W, ARROW_W, ARROW_H, 1);
          /* blit the arrow to top-left corner of the screen, */
          /* performing expand of 1 to n bits-per-pixel       */
          set_colors(LIGHT_GRAY, DARK_GRAY);
          bitblt(ARROW_W, ARROW_H, 0, 0, 0, 0);
        }
        set_videomode(PREVIOUS, INIT);
      }
    }
```

Syntax

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

void get_palet(palet_size, palet)
    short palet_size;
    PALET far *palet;
```

Type Core

Description The **get_palet** function reads an entire palette into the `palet` array. The `palet_size` parameter should be the same as the entry in the CONFIG structure to return the entire palette into the `palet` array defined in host memory.

Note that the `palet` values returned are the physical colors used in the palette on the board. If a palette hexadecimal entry is set by the **set_palet** or **set_palet_entry** functions to

Red = FF

Green = FF

Blue = FF

Intensity = 0F

With the actual color palet using only 4 bits per gun, the hexadecimal values read by a call to **get_palet** or **get_palet_entry** are

Red = F0

Green = F0

Blue = F0

Intensity = 00

See also Appendix B.7 for details on color selection.

Example See call to **get_palet_entry**.

Syntax

```
int get_palet_entry(index, r, g, b, i)
    long index;
    char far *r;
    char far *g;
    char far *b;
    char far *i;
```

Type Core

Description The **get_palet_entry** routine returns the `r`, `g`, `b`, and `i` entries for a given entry in the palette. If the index is in the valid range, this function returns true (nonzero) and the palette entry. If the index is invalid, the values returned are also invalid, and the function returns false (zero).

Example

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

CONFIG config;

main()
{
  char  r, g1, g2, b, i;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      if (function_implemented(SET_PALET_ENTRY))
      {
        /* set two logical colors, shades of green       */
        set_palet_entry(1, 0, 0xFF, 0, 0);
        set_palet_entry(2, 0, 0xF0, 0, 0);
        /* get the physical colors back                  */
        get_palet_entry(1, &r, &g1, &b, &i);
        get_palet_entry(2, &r, &g2, &b, &i);
        /* if green values are the same,use blue instead */
        if (g1 == g2)
        {
          set_palet_entry(2, 0, 0, 0xFF, 0);
        }
        /* fill some rects based on these colors         */
        set_fcolor(1);
        fill_rect(config.mode.disp_hres>>1,
                  config.mode.disp_vres>>1, 0, 0);
        set_fcolor(2);
        fill_rect(config.mode.disp_hres>>1,
                  config.mode.disp_vres>>1,
                  config.mode.disp_hres>>1,
                  config.mode.disp_vres>>1);
        /* restore the palet                             */
        init_palet();
      }
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type Extended

Description The **get_pixel** function returns the value of the pixel at coordinates (`x,y`). The coordinates are 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 MSBs are 0s).

Syntax `long get_pmask();`

Type Core

Description The **get_pmask** function returns the value of the plane mask (GSP PMASK register). Although only the 16 LSBs of the PMASK register are implemented in the TMS34010, the plane mask is 32 bits to provide upward compatibility with future TMS340 processors.

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

Syntax `int get_ppop()`

Type Core

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

The PPOP code determines the manner in which pixels are combined (logically or arithmetically) during drawing operations. A new PPOP code can be selected with the **set_ppop** function. Legal PPOP codes are in the range 0 to 21.

Table 3–1. Pixel Processing Options

Code	Replace Destination Pixel with:	Code	Replace Destination Pixel with:
0	source	11	NOT source AND destination
1	source AND destination	12	all 1s
2	source AND NOT destination	13	NOT source or destination
3	all 0s	14	source NAND destination
4	source OR NOT destination	15	NOT source
5	source EQU destination	16	source + destination
6	NOT destination	17	ADDS (source, destination)
7	source NOR destination	18	destination - source
8	source OR destination	19	SUBS (destination - source)
9	destination	20	MAX (source, destination)
10	source XOR destination	21	MIN (source, destination)

The effects of the 22 different codes are described in the *TMS34010 User's Guide*.

Example See call to **set_ppop** in **draw_line.**

Syntax

```
int get_textattr(pControl, count, arg)
    char far *pControl;
    short count;
    short far *arg;
```

Type Extended

Description The **get_textattr** function gets text rendering attributes. `pControl` is a control string specifying the attributes to be retrieved. Values associated with each requested attribute are stored in order in the array specified by `arg`. The number of attributes in the control string is passed in parameter `count`. The number of attributes successfully assigned is returned. This is the current list of valid attributes:

Attribute	Description	Option Value
%a	alignment	0 = topleft,1=baseline
%e	additional intercharacter spacing	16-bit signed integer

Syntax `int get_transp();`

Type Core

Description The **get_transp** function returns the state of the transparency enable bit (the T bit from the TMS340's control register). A value of true (nonzero) is returned if transparency is enabled; otherwise, false (zero) is returned.

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

Syntax

```
unsigned long get_vector(trapnum)
   unsigned short trapnum;
```

Type

Core

Description

The **get_vector** function returns the address currently in the trap vector specified by `trapnum`. This function should be used whenever it is necessary to read a trap vector address.

Syntax `int get_videomode();`

Type Core

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

Syntax `int get_windowing();`

Type Core

Description The **get_windowing** function returns the 2-bit windowing code contained in the control I/O register. The windowing codes are

00 No windowing

01 Interrupt request on write in window

10 Interrupt request on write outside window

11 Clip to window

For a more detailed description of the windowing operation, refer to the *TMS34010 User's Guide.*

Syntax

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

Type Core

Description The **get_wksp** function returns the parameters defining the current offscreen workspace. The function returns false(zero) if no offscreen workspace is currently allocated. True (nonzero) is returned if a valid offscreen workspace is present. If true is returned, then the address and pitch of the offscreen workspace is returned through the `addr` and `pitch` pointer parameters, respectively.

When using functions, such as **fill_polygon**, that require an offscreen workspace, check for a valid offscreen workspace by calling **get_wksp**. If none is present, then allocate one using **gsp_malloc** and update the workspace parameters via the **set_wksp** function.

Example See **fill_polygon**.

Syntax

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

Type Core

Description The **gsp2cop** function copies data from TMS340 space into the coprocessor space with ID `copid` (a number from 0 — 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

```
void gsp2gsp(addr1, addr2, length)
   long addr1;
   long addr2;
   long length;
```

Type Core

Description The **gsp2gsp** function copies `length` bytes from TMS340 memory to TMS340 memory. It handles overlapping regions. There is no restriction on the alignment of the address.

Syntax

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

Type Core

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).

Syntax

```
void gsp2hostxy(saddr, sptch, daddr, dptch, sx, sy, dx, dy,
                xext, yext, psize, swizzle)
   long 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

Core

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, `spitch`, is the difference in the linear addresses between two pixels in the same column and adjacent rows of the bitmap in TMS340 memory. `dpitch` 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), though a long variable, must be less than 16 bits.
- 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

```
long 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 to it. 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**, **gsp_minit,** and **gsp_realloc**.

Syntax

```
void gsp_execute(entry_point)
   unsigned long entry_point;
```

Type

Core

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 start 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 (cd_is_alive())
    {
      if (entry = loadcoff(argv[1]))
        gsp_execute(entry);
      else
        printf("Error in load\n");
    }
    else printf("TIGACD not running\n");
  }
  else printf("Usage: load <coff filename>\n");
}
```

Syntax

```
int gsp_free(ptr)
    long ptr;
```

Type Core

Description The **gsp_free** function routine 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.

Example See **draw_ovalarc.**

Syntax

```
long 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 to it. 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**.

Example

See **fill_polygon.**

Syntax `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 download an object, for example, a bitmap.

Syntax

```
void gsp_minit(stack_size)
   long stack size;
```

Type

Core

Description

The **gsp_minit** function reinitializes and frees all pointers in the TMS340 dynamic memory in the heap pool. Any previously allocated blocks are no longer allocated, and all pointers to such blocks become invalid after this procedure is called. It can also be used to modify the `stack_size`. The default stack size is defined in the CONFIG structure. Supplying an argument of –1 selects this stack size. Select a larger/smaller stack by supplying the stack size in bits.

Be careful using this function because all allocated blocks of memory are freed, possibly including the background save area for the cursor (if stored in heap). Disable the cursor prior to 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. **gsp_minit** 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

```
long gsp_realloc(ptr, size)
   long ptr;  /* pointer to object to change          */
   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 since 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

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

Type Core

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 non-zero, the data is swizzled before it is written to the TMS340 (that is, the order of the bits in each byte is reversed). `hptr` is a pointer to host memory and must be declared as a long pointer (that is, segment:offset format). `gptr` is a pointer to TMS340 memory. It must be byte aligned (that is, 3 LSBs must be 0).

Example See **get_offscreen_memory.**

Syntax

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

Type Core

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, `spitch`, 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 `dpitch` is the same for TMS340 memory.

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 `void init_palet()`

Type Core

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

index 0 = Black	index 8 = Dark Gray
index 1 = Blue	index 9 = Light Blue
index 2 = Green	index 10 = Light Green
index 3 = Cyan	index 11 = Light Cyan
index 4 = Red	index 12 = Light Red
index 5 = Magenta	index 13 = Light Magenta
index 6 = Brown	index 14 = Yellow
index 7 = Light Gray	index 15 = White

The palette may well contain more than 16 entries. If so, this function replicates these 16 colors through the rest of the palette.

Example See **get_palet_entry.**

Syntax `void init_text()`

Type Core

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

Syntax

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

Type Core

Description The **install_alm** function installs the absolute load module (specified by the function `alm_name`) into theTIGA graphics manager and returns a module identifier which is used to invoke the extensions specified in the TIGAEXT section.

If the module contains interrupt service routines, they will be 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 4.

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 routine number and command type using the bitwise-OR operator (|).

These are the error codes returned by this function:

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in main TIGA directory; either the TIGA environment variable `-m` option is not set, or that directory does not contain `TIGALNK.EXE`.
–3	**Out of Memory** – Not enough host memory to run `TIGALNK` or not enough TMS340 memory to store ALM.
–4	**Communications Driver not Running** – Run `TIGACD`
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`
–8	**Missing ALM** – Either 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.

Example See Section 4.3.2 on page 4-8.

Syntax

```
short install_font(pFont)
   unsigned long pFont;
```

Type Extended

Description The **install_font** function installs the font pointed to by `pFont` into the font table and returns an identifier (ID) for the font that can be used to reference the font in subsequent text operations.

Note that `pFont` is a pointer to a font already located in TMS340 memory. The **install_font** function merely adds the address of the font to the font table. The font must first be loaded from disk by the host and downloaded intoTMS340 memory. This is shown in the example.

The ID returned is nonzero if the installation was successful. If unsuccesful, the reserved ID for the system font (zero) is returned. For further details on the font format see Appendix A.

Example

```
#include <tiga.h>
#include <typedefs.h>
#include <extend.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    */
int loadinst_font(name)
char *name;
{
  FILE *fp;
  FILEHDR fh;
  FONT *hpTmp;
  int  id = 0;
  PTR  gpTmp;

  /* 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 `int install_primitives()`

Type Core

Description The **install_primitives** function is similar to a call to **install_rlm** and is used to download extended primitives such as **draw_line** etc. Before calling an extended primitive download them either by this command, or by installing a dynamic load module, that includes the primitives. For more details on extensibility and the use of this function, refer to Chapter 4.

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

This function returns these error codes:

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in main TIGA directory, either the TIGA environment variable –m option is not set, or that directory does not contain `TIGALNK.EXE`.
–3	**Out of Memory** – Not enough host memory to run `TIGALNK` or not enough TMS340 memory to store RLM.
–4	**Communications Driver not Running** – Run `TIGACD`.
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`
–7	**Symbol File Error** – I/O error obtained in accessing symbol file. The –m option of the TIGA environment variable is not set or the directory does not contain `TIGA340.SYM`, or the file is corrupt. In the latter case, recopy this file from your backup disk.
–10	**Symbol Reference** – An unresolved symbol was referenced by the RLM. Determine the name either by producing a link map for the RLM or by invoking `TIGALNK` from the command line.
–12	**Symbol Table Mismatch** – The modules installed in the symbol table do not match those the TIGA graphics manager has installed. Reinitialize the modules by a call to **flush_extended** and install them again.

Example

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

CONFIG config;

main()
{
  /* using INIT_GLOBALS does not initialize the heap,      */
  /* if the primitives were installed they still are       */
  if (set_videomode(TIGA, INIT_GLOBALS | CLR_SCREEN))
  {
    /* install the optional TIGA extended primitives       */
    if (install_primitives() < 0)
    {
      printf("Cannot install extended primitives\n");
      exit(0);
    }
    get_config(&config);
    set_fcolor(LIGHT_GRAY);
    draw_line(0, 0,   config.mode.disp_hres,
              config.mode.disp_vres);
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type Core

Description The **install_rlm** function installs the relocatable load module (specified by `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 module 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 routine number and command type using the bitwise-OR operator (|).

For more details on extensibility and the use of this function refer to Chapter 4.

This function returns these error codes:

Error Code	Description
–1	**System Error** – Could not find `TIGALNK` in main TIGA directory, either the TIGA environment variable –m option is not set or that directory does not contain `TIGALNK.EXE`.
–3	**Out of Memory** – Not enough host memory to run `TIGALNK` or not enough TMS340 memory to store RLM.
–4	**Communications Driver not Running** – Run `TIGACD`.
–5	**Graphics Manager not Running** – Run `TIGALNK -lx`
–6	**Missing RLM** – 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.
–7	**Symbol File Error** – I/O error obtained in accessing symbol file. The –m option of the TIGA environment variable is not set or the directory does not contain `TIGA340.SYM` or the file is corrupt. In the latter case, recopy this file from your backup disk.
–10	**Symbol Reference** – An unresolved symbol was referenced by the RLM. Determine the name either by producing a link map for the RLM or by invoking `TIGALNK` from the command line.

Error Code	Description (continued)
–12	**Symbol Table Mis-match** – The modules installed in the symbol table do not match those the TIGA graphics manager has installed. Reinitialize the modules by a call to **flush_extended** and install them again.

Example See Section 4.3.1 on page 4-7.

Syntax

```
void  install_usererror(function_name)
   void (far *function_name)();
```

Type

Core

Description

The **install_usererror** function installs a user error function that is called when an error is found in the host communications. The default usererror function simply prints a message to the screen when an error occurs. The user 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;
```

Current error codes:

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.

Example

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

CONFIG config;

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

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);
}
```

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

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      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 GSP 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 GSP          */
      pen_polyline_a(nofpts, lotofpts);
      /* wait for GSP side to finish (to producetimeouts) */
      synchronize();
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type Core

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

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

Example See **fill_polygon.**

Syntax

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

Type Core

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 in the parameter. It returns false (zero) 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 `short page_busy()`

Type Core

Description The **page_busy** function is used in conjuction with the **page_flip** function to determine the status of page flipping. **page_busy** returns true (nonzero) if **page_flip** is called and the pages have not been flipped. Otherwise, false (zero) is returned.

Example See **page_flip**.

Syntax

```
int page_flip(display, drawing)
    short display;
    short drawing;
```

Type Core

Description The **page_flip** function can be used if the `num_pages` entry in the CONFIG structure (described in Appendix A) is greater than 1, indicating that multiple frame buffers are available in a particular configuration. This function sets the display page to display a particular frame buffer and sets the drawing page for subsequent drawing operations.

The switching of display and drawing pages is performed at the start of vertical blanking.

If the function completes normally, it returns true (nonzero). If the display or drawing page is invalid, the function aborts and returns false (zero).

Example

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

#define TRUE 1
#define BALL_WT 16
#define BALL_HT 16
#define XMIN 0
#define YMIN 0
#define XMAX (config.mode.disp_hres-BALL_WT)
#define YMAX (config.mode.disp_vres-BALL_HT)
#define NUM_SPRITES 15

typedef struct
{
  short   x;
  short   y;
} SPRITE;

CONFIG config;
MODEINFO modeinfo;

short display_page = 0;
static SPRITE sprite[NUM_SPRITES];
static short dx[NUM_SPRITES];
static short dy[NUM_SPRITES];

flip_page()
{
  display_page ^= 1;
  page_flip(display_page, 1-display_page);
}

init_logic()
{
  register int i;
  srand(9);
  for (i = 0; i < sizeof(sprite)/sizeof(SPRITE); i++)
  {
    sprite[i].x = rand() % (config.mode.disp_hres - 32);
    sprite[i].y = rand() % (config.mode.disp_vres - 32);
    dx[i] = (rand() % 6) + 1;
    dy[i] = (rand() % 5) + 1;
  }
}
```

```
build_screen()
{
  register int i;

  set_fcolor(0);
  clear_page(BLACK);
  for (i = 0;i < sizeof(sprite)/sizeof(SPRITE);i++)
  {
    set_fcolor(i+1);
    fill_oval(16,16,sprite[i].x,sprite[i].y);
  }
}

animate()
{
  register int i;
  short tx, ty, tdx, tdy;

  for (i = 0; i < sizeof(sprite)/sizeof(SPRITE); i++)
  {
    tx = sprite[i].x;
    ty = sprite[i].y;
    tdx = dx[i];
    tdy = dy[i];
    if (tx >= XMAX && tdx > 0) tdx = -tdx;
    else if (tx <= XMIN && tdx < 0) tdx = -tdx;
    else if (ty >= YMAX && tdy > 0) tdy = -tdy;
    else if (ty <= YMIN && dy[i] < 0) tdy = -tdy;
    sprite[i].x += ( dx[i] = tdx );
    sprite[i].y += ( dy[i] = tdy );
  }
  while (page_busy());
  build_screen();
  flip_page();
}
```

```
main()
{
  int mode, oldmode;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      oldmode = config.current_mode;
      /* look for a mode with more than 1 display page    */
      mode = 0;
      do get_modeinfo(mode, &modeinfo);
      while ((modeinfo.num_pages == 1) &&
             (++mode < config.num_modes));
      if (modeinfo.num_pages > 1)
      {
        /* set configuration to multiple pages mode       */
        set_config(mode, TRUE);
        /* update config structure for current mode       */
        get_config(&config);
        flip_page();
        init_logic();
        do animate();
        while (!kbhit());
        getch();
      }
      if (mode != oldmode)
      {
        /* restore old mode                               */
        set_config(oldmode, TRUE);
        clear_frame_buffer(BLACK);
      }
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
typedef struct
{
    short x;
    short y;
}POINTS;

void patnfill_convex(n, points);
    short  n;
    POINTS far *points;
```

Type Extended

Description The **patnfill_convex** function fills a convex polygon with a pattern, given a list of points representing the vertices. To be drawn correctly, the polygon must be convex; that is, it should contain no concavities. A polygon must have at least three vertices to be visible. To ensure that the polygon is closed, the first and last vertices should contain the same coordinates. The polygon is pattern-filled with the current pattern, which is drawn in the current foreground and background colors.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

The **patnfill_convex** function automatically culls back faces to support 3D applications. A polygon is drawn only if its front side is visible, that is, if it is facing toward the screen. If the vertices are specified in counterclockwise order, the polygon is assumed to be facing away from the screen and is therefore not drawn.

The back face test is done by first comparing vertices n – 2, n – 1, and 0 to determine whether the polygon vertices are specified in clockwise (front face) or counterclockwise (back face) order. This test relies on the polygon containing no concavities. If the three vertices are 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 not colinear. If all the vertices are colinear, the polygon is invisible.

This function is similar to the **patnfill_polygon** function but is specialized for rapid drawing of convex polygons.

The argument `points` can be of any length. The application can easily overflow the command buffer which is used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_con-**

fig function. It is up to the application to check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **patnfill_convex_a** with the same parameterization is also supplied to check the size of the data to be sent. If the command buffer overflows **patnfill_convex_a** will attempt to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size of the data being transferred; however, the invocation of the function takes longer becuase the length of the data must be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function).

Syntax

```
void patnfill_oval(w, h, xleft, ytop)
   short w, h;       /* width, height of enclosing rect.  */
   short xleft, ytop;/* XY coordinates of top left corner */
```

Type

Extended

Description

The **patnfill_oval** function fills an ellipse with the current pattern in the current foreground and background colors. The ellipse is in standard position, with its major and minor axes parallel to the coordinate axes.

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

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

Example

See similar call to **patnfill_piearc.**

Syntax

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

Type Extended

Description The **patnfill_piearc** function fills a pie-shaped wedge with a pattern. The wedge is bounded by an arc and two straight edges. The arc is taken from an ellipse in standard position, with its major and minor axes parallel to the coordinate axes. The two straight edges are defined by lines connecting the end points of the arc with the center of the ellipse. The arc is filled with the current pattern in the current foreground and background colors.

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

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

The last two arguments define the limits of the arc:

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

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

Example

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

CONFIG config;

/* number of installed patterns                                    */
#define PMAX 9
typedef short PAT_ARY[16];
PAT_ARY patterns[PMAX] =
{
   /* PATTERN # 0                                                  */
   { 0x0000, 0x3FFC, 0x7FFE, 0x0006, 0x0006, 0x1FC6, 0x3FE6,0x3066,
   0x3066, 0x33E6, 0x31C6, 0x3006, 0x3006, 0x3FFE, 0x1FFC, 0x0000},
   /* PATTERN # 1                                                  */
   {0x0000, 0x0080, 0x0080, 0x0080, 0x01C0, 0x01C0, 0x7FFF, 0x1FFC,
    0x0FF8, 0x03E0, 0x03E0, 0x07F0, 0x0630, 0x0C18, 0x0808, 0x0000},
   /* PATTERN # 2                                                  */
   { 0x0000, 0x0000, 0x0E38, 0x1F7C, 0x3FFE, 0x3FFE, 0x3FFE, 0x3FFE,
    0x1FFC, 0x0FF8, 0x07F0, 0x03E0, 0x01C0, 0x0080, 0x0000, 0x0000 },
   /* PATTERN # 3                                                  */
   { 0x0000, 0x01C0, 0x19CC, 0x188C, 0x0490, 0x02A0, 0x31C6, 0x3FFE,
    0x31C6, 0x02A0, 0x0490, 0x188C, 0x19CC, 0x01C0, 0x0000, 0x0000 },
   /* PATTERN # 4                                                  */
   { 0x0420, 0x0420, 0x3FFC, 0x2424, 0x2424, 0xFC3F, 0x2004, 0x2004,
    0x2004, 0x2004, 0xFC3F, 0x2424, 0x2424, 0x3FFC, 0x0420, 0x0420 },
   /* PATTERN # 5                                                  */
   { 0x0101, 0x0101, 0x8282, 0x7C7C, 0x1010, 0x1010, 0x2828, 0xC7C7,
    0x0101, 0x0101, 0x8282, 0x7C7C, 0x1010, 0x1010, 0x2828, 0xC7C7 },
   /* PATTERN # 6                                                  */
   { 0x0000, 0x0000, 0x0000, 0x1FF0, 0x1FF0, 0x0AB0, 0x1570, 0x0AB0,
    0x1570, 0x0AB0, 0x1570, 0x0AB0, 0x0000, 0x0000, 0x0000, 0x0000 },
   /* PATTERN # 7                                                  */
   { 0x0000, 0xFE7F, 0xFE7F, 0xFE7F, 0xFE7F, 0xFE7F, 0xFE7F, 0x0000,
    0x0000, 0x7FFE, 0x7FFE, 0x7FFE, 0x7FFE, 0x7FFE, 0x7FFE, 0x0000 },
   /* PATTERN # 8                                                  */
   { 0xF007, 0xF803, 0x9C03, 0x0E07, 0x070E, 0x039C, 0x03F8, 0x07F0,
    0x0FE0, 0x1FC0, 0x39C0, 0x70E0, 0xE070, 0xC039, 0xC01F, 0xE00F },
};
static short angles[] ={33, 22, 50, 16, 24, 60, 75, 55, 58};
/* Initialize pattern structure                                    */
PATTERN patn =
{
   16,16,1,0L /* width, height, depth, data ptr                    */
};
```

```
main()
{
  short w, h, x, y, theta, arc, i;
  PTR   gsp_patterns;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      /* allocate space for patterns in GSP memory        */
      gsp_patterns = gsp_malloc(PMAX * sizeof(PAT_ARY));
      /* download patterns from host to gsp               */
      host2gsp ((char far *) patterns, gsp_patterns,
        PMAX * sizeof(PAT_ARY), 0);
      /* setup up with rectangle for the piechart         */
      w = config.mode.disp_hres>>1;
      h = config.mode.disp_vres>>1;
      x = config.mode.disp_hres>>2;
      y = config.mode.disp_hres>>2;
      set_bcolor(BROWN);
      set_fcolor(GREEN);
      theta = angles[0];
      /* draw a piechart with pies filled with patterns   */
      for (i = 1; i < 9; ++i)
      {
        arc = angles[i];
        patn.data = gsp_patterns + ((long)i<<8);
        set_patn((char far*)&patn);
        patnfill_piearc(w, h, x, y, theta, arc);
        theta += arc;
      }
      set_fcolor(CYAN);
      frame_oval(w, h, x, y, 2, 2);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
typedef struct
{
   short x;
   short y;
}POINTS;

void patnfill_polygon(n, points)
   long  n;
   POINTS far *points;
```

Type Extended

Description The **patnfill_polygon** function fills a polygon with a pattern, given a list of endpoints of the polygon. No restrictions are placed on the shape of the polygons filled by this function: edges can cross each other, filled areas can contain holes, and two or more filled regions can be disconnected from each other. The polygon is filled with the current pattern using the current foreground and background colors. To ensure that the polygon is closed, the first and last vertices should have the same coordinates.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

This function also takes as an implied argument a 1-bit representation of the frame buffer, which it uses as a temporary workspace. This workspace must be set up prior to invoking the **patnfill_polygon** function (via a call to the **set_wksp** function).

The argument `points` can be of any length. The application can easily overflow the command buffer that is used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. It is up to the application to check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point, **patnfill_polygon_a** with the same parameterization, is also supplied to check the size of the data to be sent. If the command buffer overflows, **patnfill_polygon_a** attempts to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size

of the data being transferred. There is a cost, however, in that the invocation of the function takes longer since the length of the data has to be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function)

Example See call to **fill_polygon**.

Syntax

```
void patnfill_rect(w, h, xleft, ytop)
   short w, h;          /* width and height of rectangle   */
   short xleft, ytop;   /* XY coord at top left corner     */
```

Type

Extended

Description

The **patnfill_rect** function fills a rectangle with the current pattern in the current foreground and background colors. Four arguments define the rectangle:

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

Example

See call to **fill_rect** in **fill_polygon** example.

Syntax

```
void patnframe_oval(w, h, xleft, ytop, dx, dy)
    short w, h;          /* width, height of rect.          */
    short xleft, ytop;   /* coordinates at top left corner  */
    short dx, dy;        /* width, height of frame border   */
```

Type Extended

Description The **patnframe_oval** function fills an ellipse-shaped frame with a pattern. The frame consists of a filled region between two concentric ellipses. The frame is filled with the current pattern in the current foreground and background colors. The portion of the screen enclosed by the frame is not altered.

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

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

The thickness of the frame in the X and Y dimensions is defined by two additional arguments:

- `dx` specifies the horizontal distance between the outer and inner ellipses.
- `dy` specifies the vertical distance between the outer and inner ellipses.

Example See similar call to **frame_oval** in **patnfill_piearc** example.

Syntax

```
void patnframe_rect(w, h, xleft, ytop, dx, dy)
   short w, h;          /* width, height of rect.          */
   short xleft, ytop;   /* coordinates at top left corner  */
   short dx, dy;        /* width, height of frame border   */
```

Type Extended

Description The **patnframe_rect** function fills a rectangular frame with a pattern. The frame consists of a filled region between two concentric rectangles. The frame is filled with the current pattern in the current foreground and background colors. The portion of the screen enclosed by inner edge of the frame is not altered.

The first four arguments define the outer rectangle:

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

The thickness of the frame in the X and Y dimensions is defined by two additional arguments:

- `dx` specifies the horizontal distance between the outer and inner rectangles.
- `dy` specifies the vertical distance between the outer and inner rectangles.

Syntax

```
void patnpen_line(x1, y1, x2, y2)
   short  x1, y1;  /* starting coordinates                    */
   short  x2, y2   /* ending coordinates                      */
```

Type

Extended

Description

The **patnpen_line** function uses the pen to draw a patterned line. Arguments `x1` and `y1` specify the starting coordinates of the line, and `x2` and `y2` specify the ending coordinates.

The pen is a rectangle whose width and height can be modified by means of the **set_pensize** function. At each point on the line drawn by the **patnpen_line** function, the pen is located with its top left corner touching the line. The area covered by the pen is filled with the current pattern in the current foreground and background colors.

Example

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

CONFIG config;

/* number of installed patterns */
#define PMAX 8
typedef short PAT_ARY[16];
PAT_ARY patterns[PMAX] =
{
  /* PATTERN # 0                                                    */
  { 0xAAFA, 0xFF77, 0xFFFF, 0xFF77, 0xAAFA, 0x7070, 0xF8F8, 0x7070,
  0xFAAA, 0x77FF, 0xFFFF, 0x77FF, 0xFAAA, 0x7070, 0xF8F8, 0x7070 },
  /* PATTERN # 1                                                    */
  { 0x00C0, 0x0030, 0x1F88, 0x2044, 0x4024, 0x4E24, 0x9124, 0xA1C4,
  0x2385, 0x2489, 0x2472, 0x2402, 0x2204, 0x11F8, 0x0C00, 0x0300 },
  /* PATTERN # 2                                                    */
  { 0x93C9, 0x0E70, 0x1C38, 0xB99D, 0xF24F, 0x6666, 0xCDB3, 0x4DB2,
  0x4DB2, 0xCDB3, 0x6666, 0xF24F, 0xB99D, 0x1C38, 0x0E70, 0x93C9 },
  /* PATTERN # 3                                                    */
  { 0x0000, 0x7FFE, 0x6426, 0x524A, 0x4992, 0x6666, 0x566A, 0x4992,
  0x4992, 0x566A, 0x6666, 0x4992, 0x524A, 0x6426, 0x7FFE, 0x0000 },
  /* PATTERN # 4                                                    */
  { 0x0441, 0x8AA2, 0x5114, 0x2388, 0x5114, 0x8442, 0x0EE1, 0xA54A,
  0x739C, 0xA54A, 0x0EE1, 0x8442, 0x5114, 0x2388, 0x5114, 0x8AA2 },
  /* PATTERN # 5                                                    */
  { 0x0000, 0x601E, 0x7C31, 0x2733, 0x29B3, 0x34B0, 0x12B0, 0x18B0,
  0x0F70, 0x00B2, 0x1FF4, 0x7FFC, 0x601C, 0x403C, 0x5842, 0x3800 },
  /* PATTERN # 6                                                    */
  { 0x1111, 0x2222, 0x4444, 0x8888, 0x1111, 0x2222, 0x4444, 0x8888,
  0x1111, 0x2222, 0x4444, 0x8888, 0x1111, 0x2222, 0x4444, 0x8888 },
  /* PATTERN # 7                                                    */
  { 0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111,
  0x8888, 0x4444, 0x2222, 0x1111, 0x8888, 0x4444, 0x2222, 0x1111 },
};
/* Initialize pattern structure                                     */
PATTERN pattern =
{
  16,16,1,0L /* width, height, depth, data ptr                     */
};
```

```
main()
{
  long  x, y;
  long  r, t;
  short fcolor, colormax, patn;
  PTR   gsp_patterns;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      set_draw_origin(config.mode.disp_hres>>1,
      config.mode.disp_vres>>1);
      set_pensize(4, 3);
      /* allocate space for patterns in GSP memory        */
      gsp_patterns = gsp_malloc(PMAX * sizeof(PAT_ARY));
      /* download patterns from host to gsp               */
      host2gsp ((char far *) patterns, gsp_patterns,
                PMAX * sizeof(PAT_ARY), 0);
      /* initialize colormax variable                     */
      colormax = (1<<config.mode.disp_psize) - 1;
      fcolor = 1;
      patn   = 0;
      /* draw a swirling patterned shape                  */
      for (r = 20; r < (config.mode.disp_vres>>1);
            r += r >> 4)
      {
        x = 0;
        y = -r << 16;
        for (t = 0; t < 201; ++t)
        {
          set_fcolor(fcolor);
          if (++fcolor == colormax) fcolor = 1;
          if (++patn   == PMAX) patn   = 0;
          pattern.data = gsp_patterns+((long)patn<<8);
          set_patn(&pattern);
          patnpen_line(x-(x>>2)>>16, y-(y>>2)>>16, x>>16,
                       y>>16);
          x -= y >> 5;
          y += x >> 5;
        }
      }
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

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

Type Extended

Description The **patnpen_ovalarc** function uses the pen to draw a patterned arc of an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the **set_pensize** function. At each point on the arc drawn by the **patnpen_ovalarc** function, the pen is positioned so that its top left corner touches the arc. The area covered by the pen is filled with the current pattern in the current foreground and background colors.

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

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

The last two arguments define the limits of the arc:

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

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

Example See **patnpen_line** for use of patterned pen and **draw_ovalarc** for use oval arcs.

Syntax

```
void patnpen_piearc(w, h, xleft, ytop, theta, arc)
    short w, h;          /* width and height                */
    short xleft, ytop;   /* top left corner                 */
    short theta;         /* starting angle (degree)         */
    short arc;           /* angle extent (degree)           */
```

Type Extended

Description The **patnpen_piearc** function uses the pen to draw a patterned, pie-shaped wedge from an ellipse. The wedge is formed by an arc of the ellipse, and by two straight lines that connect the two end points of the arc with the center of the ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of **set_pensize** function. At each point on the arc drawn by the **patnpen_piearc** function, the pen is positioned so that its top left corner touches the arc. The two lines from the center are drawn in similar fashion. The area covered by the pen is filled with the current pattern in the current foreground and background colors.

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

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

The last two arguments define the limits of the arc:

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

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

Example See **patnpen_line** for use of patterned pen and similar call to **patnfill_piearc.**

Syntax

```
void patnpen_point(x ,y)
   short x, y; /* pen coordinates */
```

Type Extended

Description The **patnpen_point** function uses the pen to draw a patterned point. The resulting figure is a rectangle the width and height of the pen, filled with the current pattern in the current foreground and background colors. The top left corner of the rectangle is located at coordinates `(x,y)`.

Example See **patnpen_line** for use of patterned pen and **draw_point** for drawing single pixels.

Syntax

```
typedef struct points_struct
{
    short  x;
    short  y;
} POINTS;

void patnpen_polyline(n, points)
    short  n;
    POINTS far *points;
```

Type Extended

Description The **patnpen_polyline** function uses the current pen to draw `n` patterned lines whose endpoints are supplied in an array of structures, described in the syntax.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

The argument `points` can be of any length. The application can easily overflow the command buffer that is used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. The application must check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **patnpen_polyline_a** with the same parameterization, is also supplied to check the size of the data to be sent. If the command buffer overflows **patnpen_polyline_a** attempts to allocate a temporary buffer in heap. In this way, the application is freed from having to check the size of the data being transferred; however, the invocation of the function takes longer because the length of the data must be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function)

Example See **patnpen_line** for use of patterned pen and similar call to **draw_polyline** for use of polylines.

Syntax

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

Type Core

Description The **peek_breg** function returns the contents of a B-file register. Argument `breg` is a register number in the range 0 to 15. The function ignores all but the 4 LSBs of `breg`. The return value is 32 bits.

Syntax

```
void pen_line(x1, y1, x2, y2)
   short x1, y1;   /* starting coordinates                    */
   short x2, y2;   /* ending coordinates                      */
```

Type

Extended

Description

The **pen_line** function uses the pen to draw a line. Arguments `x1` and `y1` specify the starting coordinates of the line; `x2` and `y2` specify the ending coordinates.

The pen is a rectangle whose width and height can be modified by means of the **set_pensize** function. At each point on the line drawn by the **pen_line** function, the pen is located with its top left corner touching the line. The area covered by the pen is solid-filled in the current foreground color.

Example

See **install_usererror** for similar call to **pen_polyline**.

Syntax

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

Type Extended

Description The **pen_ovalarc** function uses the pen to draw an arc taken from an ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the **set_pensize** function. At each point on the arc drawn by the **pen_ovalarc** function, the pen is located with its top left corner touching the arc. The area covered by the pen is solid-filled in the current foreground color.

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

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

The last two arguments define the limits of the arc:

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

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

Example See **install_usererror** for use of drawing pen and **draw_ovalarc** for use of oval arcs.

Syntax

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

Type Extended

Description The **pen_piearc** function uses the pen to draw a pie-shaped wedge from an ellipse. The wedge is formed by an arc of the ellipse and by two straight lines that connect the two end points of the arc with the center of the ellipse. The ellipse is in standard position, with the major and minor axes parallel to the coordinate axes.

The pen is a rectangle whose width and height can be modified by means of the **set_pensize** function. At each point on the arc drawn by the **pen_piearc** function, the pen is located with its top left corner touching the arc. The area covered by the pen is solid-filled in the current foreground color.

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

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

The last two arguments define the limits of the arc:

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

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

Example See **install_usererror** for use of drawing pen and **patnfill_piearc** for use of pie arcs.

Syntax

```
void pen_point(x, y)
   short x, y;      /* pen coordinates */
```

Type Extended

Description The **pen_point** function uses the pen to draw a point. The resulting figure is a rectangle the width and height of the pen and solid-filled with the current foreground color. The top left corner of the rectangle is located at coordinates (`x,y`).

Example See **install_usererror** for use of drawing pen and **draw_point** for drawing single pixels.

Syntax

```
typedef struct
{
    short  x;
    short  y;
}POINTS;

void pen_polyline(n, points);
    short  n;
    POINTS far *points;
```

Type Extended

Description The **pen_polyline** function uses the current pen to draw lines whose endpoints are supplied in an array of structures, described in the syntax.

The function requires two input arguments:

- The first argument, `n`, defines the number of vertices in the polygon.
- The second argument, `points`, is an array in which each pair of adjacent 16-bit quantities contains the X and Y coordinates, respectively, of a vertex.

The argument `points` can be of any length. The application can easily overflow the command buffer used by the host processor to send the function parameters to the TMS340. The size of the command buffer is in the CONFIG structure (described in Appendix A) returned by the **get_config** function. The application must check that the data sent will not overflow this buffer.

The number of points that can be sent is given by the following formula:

$$n < \frac{\text{com_buffer_size (in bytes)} - 10}{4}$$

An alternate entry point **pen_polyline_a** with the same parameterization, is supplied to check the size of the data to be sent. If the command buffer overflows **pen_polyline_a** will attempt to allocate a temporary buffer in heap. In this way the application is freed from having to check the size of the data being transferred. There is a cost, however, in that the invocation of the function will take longer since the length of the data has to be parsed. If there is not enough room to store the temporary buffer in TMS340 memory, the error function is invoked (which can be trapped by the **install_usererror** function).

Example See **install_usererror**.

Syntax

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

Type Core

Description The **poke_breg** function stores the 32-bit value in a B-file register. Argument `breg` is any register number between 0 and 15.

Syntax

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

Type Core

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

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

Syntax

```
int seed_fill(xseed, yseed, buffer, maxbytes)
    short xseed, yseed; /* coordinates of seed pixel       */
    long buffer;        /* GSP pointer to working storage  */
    short maxbytes;     /* size of buffer in bytes         */
```

Type Extended

Description The **seed_fill** function fills a connected region of pixels starting at a specified seed pixel. The seed color is the color of the specified seed pixel at the time the function is called. The connected region is solid-filled with the current foreground color.

- The first two arguments, `xseed` and `yseed`, specify the coordinates of the seed pixel.

The last two arguments specify a buffer used as a working storage during the seed fill.

- Argument `buffer` is a buffer large enough to contain the temporary data that the function uses.
- Argument `maxbytes` is the number of 8-bit bytes available in the buffer array.

Storage requirements can be expected to increase with the complexity of the connected region being filled.

Note:

The argument `buffer` is a pointer in TMS340 memory. This area must have previously been allocated using **gsp_malloc**.

The connected region filled by the function always includes the seed pixel. To be considered part of the connected region, a pixel must both match the seed color and be horizontally or vertically adjacent to another pixel that is part of the connected region. (A diagonally adjacent neighbor is not sufficient.)

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

- The seed pixel matches the current foreground color.
- The seed pixel lies outside the current window register values.
- If at any point the storage buffer space specified by maxbytes is insufficient to continue.

If the function aborts, it returns the value false (zero); otherwise, it completes normally and returns true (nonzero).

Example See **draw_ovalarc.**

Syntax

```
int seed_patnfill(xseed, yseed, buffer, maxbytes)
    short xseed, yseed; /* coordinates of seed pixel       */
    long buffer;        /* GSP pointer to working storage  */
    short maxbytes;     /* size of buffer in bytes         */
```

Type Extended

Description The **seed_patnfill** function fills a connected region of pixels with a pattern starting at a specified seed pixel. The seed color is the color of the specified seed pixel at the time the function is called. The connected region is filled with the current pattern in the current foreground and background colors.

- The first two arguments: `xseed, yseed`, specify the coordinates of the seed pixel.

The last two arguments specify a buffer used as a working storage during the seed fill.

- Argument `buffer` is a buffer large enough to contain the temporary data that the function uses.
- Argument `maxbytes` is the number of 8-bit bytes available in the buffer array.

Storage requirements can be expected to increase with the complexity of the connected region being filled.

Note:

The argument `buffer` is a pointer in TMS340 memory. This area must have previously been allocated using **gsp_malloc**.

The connected region filled by the function always includes the seed pixel. To be considered part of the connected region, a pixel must both match the seed color and be horizontally or vertically adjacent to another pixel that is part of the connected region. (A diagonally adjacent neighbor is not sufficient.)

The **seed_patnfill** function aborts (returns immediately) if any of these conditions is detected:

- The seed pixel matches the current foreground or background color.
- The seed pixel lies outside the current window register values.
- If at any point the storage buffer space specified by `maxbytes` is insufficient to continue.

If the function aborts, it returns the value false (zero); otherwise, it completes normally and returns true (nonzero).

Example See similar call to **seed_fill** in **draw_ovalarc.**

Syntax

```
int select_font( id )
    short id;
```

Type Extended

Description The **select_font** function selects a previously installed font for use by the text functions. The input argument, `id`, must be either zero, indicating the system OEM font, or a value returned by the **install_font** function.

A nonzero value is returned if the selection was successful, and a zero value if the font referred to by `id` is not installed in the font table.

Syntax

```
void set_bcolor(color)
    long color;
```

Type Core

Description The **set_bcolor** function sets the COLOR0 B-file register to the color index replicated by the pixel size of the current configuration.

Example See **patnfill_piearc**.

Syntax

```
void set_clip_rect(w,h,xleft,ytop)
   short w,h;           /* width, height of clipping rect  */
   short xleft,ytop;    /* coordinates of top left corner  */
```

Type Core

Description The **set_clip_rect** function sets the current clipping rectangle by updating the B-file registers WSTART(B5) and WEND(B6) to the dimensions specified by the parameters passed. The parameters `xleft` and `ytop` are relative to the current drawing origin.

Example See **draw_oval.**

Syntax

```
void set_colors(fcolor, bcolor)
   long fcolor;
   long bcolor;
```

Type Core

Description The **set_colors** function sets both the COLOR1 and COLOR0 B-file registers to the color indices replicated by the pixel size of the current configuration.

Example See **get_offscreen_memory**.

Syntax

```
int set_config(graphics_mode, init_draw)
    short graphics_mode;
    short init_draw;
```

Type Core

Description The **set_config** function sets the graphics mode selected by the index passed to the default graphics mode, in which the board remains until a subsequent call to **set_config** is made. Different modes can have different display resolutions, pixel sizes etc. The **get_config** function can be used to determine the number of board configurations available on a given board and the parameters for each configuration. The **get_modeinfo** function can be used to determine the particular configuration parameters for a given board. To select a particular mode, this function is invoked with an argument of the desired mode number.

If the `graphics_mode` argument is valid, the new graphics mode is set up and the function returns true (nonzero). If the argument is invalid, the function performs no operation and returns false (zero).

This function initializes the following TMS340 registers:

HESYNC to **DPYCTL I/O registers**	Initialized for the current resolution.
PSIZE I/O register	Set to the current pixel size.
CONVDP I/O register	Set to the left-most-one of the display pitch.
DPYTAP I/O register	Initialized for the board.
DPTCH B-file register	Set to the display pitch.
OFFSET B-file register	Set to the offset of the current drawing page.
WSTART B-file register	Set to 0.
WEND B-file register	Set to the `disp_vres:disp_hres` value.

If there is an on-board palette, it is reset to the default colors (see **init_palet**).

The **set_config** function also sets the display and drawing pages to page 0 (for multiple-paged frame buffers).

Furthermore if the `init_draw` flag is set to true (nonzero), the following drawing environments are initialized:

- The transparency is disabled (in CONTROL I/O register)
- Window clipping is set (in CONTROL I/O register)
- Pixel Processing is set to replace (in CONTROL I/O register)
- PMASK I/O register is set to 0
- Foreground color is set to light grey and the background color to black
- Source and destination bitmaps are set to the screen
- Drawing origin is set to 0,0
- Pen width and height are set to 1
- Current pattern address is set to 0
- All installed fonts are removed and the current selected font is set to the system font
- Graphics cursor is set to the center of the screen; it is turned off and its shape is set to the default shape
- Temporary workspace is initialized (see **set_wksp**)

Syntax

```
typedef struct
{
    short hot_x;
    short hot_y;
    short width;
    short height;
    short pitch;
    long color;
    short mask_rop;
    short shape_rop;
    PTR data;
}CURSOR;

void set_curs_shape(shape);
    long shape;
```

Type Core

Description The **set_curs_shape** function initializes the global pointer to the cursor shape with the parameter passed to it (which is a pointer in TMS340 memory). Prior to calling this function, both the cursor shape 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 loading the structure. 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 parameter of 0.

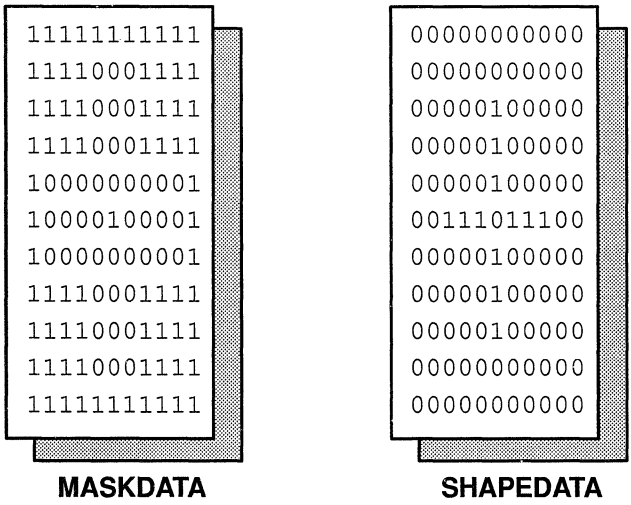
In the **set_curs_xy** function, (x, y) is the position of the top-left pixel of the cursor if `hot_x` and `hot_y` are zero. These values are subtracted from the current cursor position to give the top-left hand corner of the cursor starting drawing point. For example, in a simple crosshairs cursor of 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 data, the background and foreground colors are always 0 and 0FFFFFFFFh, respectively. The color of the cursor shape is determined by the color element in 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). 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). 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 masks for a simple crosshair cursor:



MASKDATA **SHAPEDATA**

Example

```
#include <dos.h>
#include <conio.h>
#include <typedefs.h>
#include <tiga.h>
#include <extend.h>
#define ESC 0x1B

CONFIG config;

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
};

CURSOR far pencil =
{0x0000, 0x001B, 0x0011, 0x001C, 0x0020, 0x0FL, 1, 8, 0L};

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

union REGS regs;
```

```
/* checks for an installed mouse driver                    */
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 GSP 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;
}
```

```
install_cursor(type)
short type;   /* 0=default(arrow), 1=user(pencil)          */
{
  /* Address of user cursor in GSP mem                     */
  static PTR UserCurs = 0L;

  if(type)
  {
    /* User cursor type specified                          */
    if( UserCurs == 0L )
    {
      /* Execute this block 1st time only                  */
      unsigned short num_bytes;

      /* download cursor shape data to GSP                 */
      num_bytes=((pencil.height * pencil.pitch) << 1) >> 3;
      pencil.data=(PTR)gsp_malloc(num_bytes);
      host2gsp (PencilData, pencil.data, num_bytes, 0);
      /* download cursor structure to GSP                  */
      num_bytes=sizeof(CURSOR);
      UserCurs=(PTR)gsp_malloc(num_bytes);
      host2gsp (&pencil, UserCurs, num_bytes, 0);
    }
    set_curs_shape( UserCurs );
  }
  else
    set_curs_shape((PTR)0);    /*Use default if type==0   */
}

main()
{
  char key;
  short CursorType=1;

  if (check_mouse())
  {
    if (set_videomode(TIGA, INIT | CLR_SCREEN))
    {
      if (install_primitives() >= 0)
      {
        get_config(&config);
        printf("Press...\n");
        printf("   ESC to quit\n");
        printf("   SPACE to toggle cursor shapes\n");
        printf("   LEFT mouse button to draw points\n");
              /* assign a new cursor shape              */
        install_cursor(CursorType);
```

```
            /* initialize mouse to the center of the screen   */
            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 ' ' :
                    install_cursor(CursorType^=1);
                    break;
                  case ESC :
                    set_curs_state(0); /* Turn cursor off   */
                    set_videomode(PREVIOUS,INIT);
                    exit(0);
                }
            }
          }
          set_videomode(PREVIOUS,INIT);
        }
      }
      else printf("Mouse driver required for this example\n");
    }
```

set_curs_state *Set Current Cursor State*

Syntax

```
void set_curs_state(enable)
    short enable;
```

Type Core

Description The **set_curs_state** function displays the cursor (if `enable` is nonzero) or removes it from the screen.

Example See **set_curs_shape**.

Syntax

```
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 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

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

Type Extended

Description The **set_draw_origin** function sets the drawing origin for all subsequent drawing functions. All coordinates specified for all drawing functions are relative to the drawing origin.

Example See **styled_line**.

Syntax

```
void set_dstbm(addr,pitch,xext,yext,psize)
    long addr;
    short pitch;
    short xext;
    short yext;
    short psize;
```

Type Extended

Description The **set_dstbm** function sets the TMS340's destination bitmap for subsequent **bitblt** or drawing functions. Invoking the function with an address of 0 sets the destination bitmap to the screen.

The pitch of the destination bitmap must be a multiple of 16. Currently, no clipping is performed in a linear bitmap; the calling program must do this. Future TIGA revisions will provide this capability.

Example

```
#include <typedefs.h>
#include <tiga.h>
#include <extend.h>
CONFIG config;

/* function to save portion of screen defined by arguments*/
/* into a linear bitmap and return the address            */
unsigned long save_offscreen(width, height, x_left, y_top)
short width, height, x_left, y_top;
{
  unsigned long address;
  address = (PTR) gsp_malloc(((long)width * (long)height *
                             (long) config.mode.disp_psize)/8);
  if (address)
  {
    /* turn off transparency, otherwise pixels of 0 color */
    /* will not be copied into the destination bitmap     */
    transp_off();
    set_srcbm(01,0,0,0,0);
    set_dstbm(address, width * config.mode.disp_psize,
    width, height, config.mode.disp_psize);
    bitblt(width, height, x_left, y_top, 0, 0);
    set_dstbm(01,0,0,0,0);
  }
  return(address);
}

/* function to restore to the screen a pre-saved          */
/* rectangular region in heap pointed to by address       */
restore_onscreen(address, width, height, x_left, y_top)
unsigned long  address;
short width, height, x_left, y_top;
{
  if (address)
  {
    /* turn off transparency, otherwise pixels of 0 color */
    /* will not be copied into the destination bitmap     */
    transp_off();
    set_srcbm(address, width * config.mode.disp_psize,
    width, height, config.mode.disp_psize);
    set_dstbm(01,0,0,0,0);
    bitblt(width, height, 0, 0, x_left, y_top);
    set_srcbm(01,0,0,0,0);
  }
}
```

```
main()
{
  short w, h, x, y, i;
  PTR   save;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      clear_screen(LIGHT_MAGENTA);
      w = config.mode.disp_hres>>4;
      h = config.mode.disp_vres>>4;
      x = config.mode.disp_hres>>2;
      y = config.mode.disp_vres>>1;
      /* save a portion of the screen                          */
      save = save_offscreen(w, h, x, y);
      /* clear screen to the current background color          */
      clear_screen(-1);
      x = y = 0;
      /* restore saved region to various places on screen */
      for (i = 0; i < 8; i++)
      {
        restore_onscreen(save, w, h, x, y);
        x += config.mode.disp_hres>>3;
        y += config.mode.disp_vres>>3;
      }
      gsp_free(save);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax

```
void set_fcolor(color)
   long color;
```

Type Core

Description The **set_fcolor** function sets the COLOR1 B-file register to the color index replicated by the pixel size of the current configuration.

Example See **fill_polygon.**

Syntax

```
void 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**.

The `level` parameter indicates the interrupt level where the interrupt routine was installed. The `priority` is returned by the **get_isr_priorities** function when the interrupt is installed, 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.

The `scan_line` parameter is valid only for display interrupts (interrupt level 10). It is used to set the line at which the interrupt occurs. If the `scan_line` parameter 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 re-specifying the `scan_line` parameter.

For more details on extensibility and the use of this function, refer to Chapter 4.

Example

See Section 4.9.

Syntax

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

void set_palet(count, index, palet)
    long count;
    long index;
    PALET far *palet;
```

Type Core

Description The **set_palet** function loads the palette on the board using the `palet` array. The number of palette entries to be loaded is passed in argument `count`. Argument `index` designates where the palette loading is to start. This allows for a palette to be loaded one piece at a time, rather than all at once. The range that is being loaded is checked by this function to ensure that it does not overflow the palette; thus, if the starting `index` plus the number of entries (`count`) is greater than the palette size, the `count` value is decreased by the appropriate amount.

In the PALET structure, `r`, `g`, and `b` refer to the red, green, and blue values for the color entry (0 being off and all 1s being full on); `i` is an intensity level for monochrome displays. See also Appendix B.7 for details on color selection.

On systems that store the palette data in display memory (such as those using the TMS34070), this function reloads every palette data area in the frame buffer. Thus, if the system contains multiple display pages, every page is loaded by this function.

Example See similar call to **set_palet_entry**.

Syntax

```
int set_palet_entry(index, r, g, b, i)
    long index;
    char r;
    char g;
    char b;
    char i;
```

Type Core

Description The **set_palet_entry** function initializes one entry `index` in a palette. If the function aborts, it returns the value false (zero); otherwise, it completes normally and returns true (nonzero).

In the palette structure, `r, g, b` refer to the red, green and blue values for the color entry (0 being off and all 1s being full on), and `i` is an intensity level for monochrome displays. See also Appendix B.7 for details on color selection.

On systems that store the palette data in display memory (such as those using the TMS34070) this function reloads the palette entry in every palette data area in the frame buffer. Thus if the system contains multiple display pages, every page is loaded by this function.

Example See **get_palet_entry**.

Syntax

```
typedef structure

{
  ushort width;
  ushort height;
  ushort depth;
  PTR data;
} PATTERN;

void set_patn(p)
  PATTERN far *p;
```

Type Extended

Description The **set_patn** function sets the current drawing pattern as specified by the PATTERN structure pointed to by `p`. Currently, only 16 x 16 x 1 patterns are supported in TIGA's extended graphics primitives. The application must first install its pattern data using **gsp_malloc** to allocate TMS340 storage memory and then use **host2gsp** to transfer them to TMS340 memory. A pointer to the desired pattern in TMS340 memory is then set in the PATTERN structure before calling **set_patn.**

Example See **patnfill_piearc.**

Syntax

```
void set_pensize(w, h)
   short  w, h;  /* pen width and height */
```

Type Extended

Description The **set_pensize** function specifies the width and height of a rectangular pen. These dimensions are used for any subsequent drawing operations with the pen.

Example See **install_usererror.**

Syntax

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

Type Core

Description The **set_pmask** function specifies the plane mask that is used in subsequent drawing operations. The mask determines which bits in a pixel can be modified during drawing operations. The 0s in the mask enable modification of the corresponding bit planes. The 1s in the mask write-protect the corresponding planes.

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

Syntax

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

Type Core

Description The **set_ppop** function defines the pixel processing operation for subsequent drawing operations. The specified Boolean or arithmetic operation determines the manner in which source and destination pixel values are combined. The range for the `ppop_code` argument is 0 to 21. The codes are described in the following table:

Table 3–2. Pixel Processing Options

Code	Replace Destination Pixel with:	Code	Replace Destination Pixel with:
0	source	11	NOT source AND destination
1	source AND destination	12	all 1s
2	source AND NOT destination	13	NOT source or destination
3	all 0s	14	source NAND destination
4	source OR NOT destination	15	NOT source
5	source EQU destination	16	source + destination
6	NOT destination	17	ADDS (source, destination)
7	source NOR destination	18	destination - source
8	source OR destination	19	SUBS (destination - source)
9	destination	20	MAX (source, destination)
10	source XOR destination	21	MIN (source, destination)

The *TMS34010 User's Guide* (literature number SPVU001) describes the details of these operations.

Example See **draw_line** and **zoom_rect**.

Syntax

```
void set_srcbm(addr,pitch,xext,yext,psize)
    long addr;
    short pitch;
    short xext;
    short yext;
    short psize;
```

Type Extended

Description The **set_srcbm** function sets the TMS340's source bitmap for subsequent **bitblt** functions. Invoking the function with the address of 0 sets the source bitmap to the screen.

The pitch of the source bitmap must be a multiple of 16. Currently no clipping is performed in a linear bitmap, the calling program must do this. Future TIGA revisions will provide this capability.

Example See **set_dstbm.**

Syntax

```
int set_textattr(pControl, count, arg)
   char far  *pControl;
   short count;
   short far  *arg;
```

Type Extended

Description The **set_textattr** function sets text rendering attributes. `pControl` is a control string specifying the attributes to be set. Values associated with attributes can be specified either as immediate values in the control string. or as arguments passed in the array `arg`. The number of attributes in the control string must be passed in parameter `count`.

When a values is passed as a string literal, it should be placed between the `%` character and the attribute. For example, to assign the value 1 to attribute `a` enter:

```
set_textattr("%1a", 1, 0);
```

To pass the same value as an argument, an asterisk is placed between the `%` character and attribute, and the value 1 is provided as the first argument (element 0) in the passed array:

```
short arg[1];
arg[0] = 1;
set_textattr("%*a", 1, arg[0]);
```

The number of attributes successfully set is returned. This is the current list of valid attributes:

Attribute	Description	Option Value
%a	alignment	0 = top left,1= baseline
%e	additional intercharacter spacing	16 bit signed integer

Syntax

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

Type Core

Description The **set_timeout** function sets the value of the timeout value (in milliseconds) that determines how long the host waits for a TMS340 function to complete prior to calling the error function with a timeout. 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

```
void set_transp(mode)
   short mode;
```

Type Core

Description The **set_transp** function, if implemented (on TMS34020 systems only), changes the transparency mode. Currently, these are the modes supported are:

mode = 0	Transparency on source equals zero
mode = 1	Transparency on source equals COLOR0
mode = 2	Transparency on result equals zero
mode = 3	Transparency on result equals COLOR0

On TMS34010 systems, only mode 2 is supported.

Syntax

```
unsigned long set_vector(trapnum, newaddr)
    unsigned short trapnum;
    unsigned long newaddr;
```

Type Core

Description The **set_vector** function writes the specified 32-bit address contained in `newaddr` into the trap vector specified by `trapnum`. The address originally in this trap is returned. This function should be used whenever it is necessary to modify a trap vector address.

Syntax

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

Type Core

Description The **set_videomode** function sets up the video mode to be used. Every TIGA application should start with a call to this function with a mode of TIGA and the initialize flag set (non-zero) prior to invoking any other TIGA function. This function sets up the mode to be used in a particular application. These are the valid modes:

TIGA	EGA
MDA	VGA
HERCULES	AI_8514
CGA	PREVIOUS

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 IBM emulation (since in most cases the board on which TIGA is being run is the primary video board). The mode selected could be `PREVIOUS` which restores the emulation mode from a global. 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 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** and that particular video mode is not supported by the board, the routine returns false (zero).

The `style` parameter is used to determine the manner in which the mode is set up in on entry. This list describes valid styles:

NO_INIT Used during an application to switch between TIGA and other emulation 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 value and the user error handler. The heap pool is retained, which keeps the downloadable extensions installed.

INIT Initializes global variables and dynamic memory (heap pool). This frees all allocated pointers and thus deletes all downloaded extensions.

The `style` parameter contains one further option, which can be selected by ORing with the above style parameter:

CLR_SCREEN This parameter should be used at initialization when you are using the INIT_GLOBALS or INIT styles. The screen is blanked while the video registers are initialized.

Syntax

```
void set_windowing(enable)
   short  enable;
```

Type

Core

Description

The **set_windowing** function sets the two-bit windowing code contained in the CONTROL I/O register. The windowing codes are

- 00 – No windowing
- 01 – Interrupt request on write in window.
- 10 – Interrupt request on write outside window.
- 11 – Clip to window.

For a more detatiled description of the windowing operation, refer to the *TMS34010 User's Guide* (literature number SPVU001). Care must be taken in using this function. The extended drawing primitives assume windowing option 3 (clip to window) is set. Setting the interrupt request modes (1 and 2) should only be done after downloading an interrupt service routine for the window violation interrupts (see Installing Interrupts in Chapter 4).

Syntax

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

Type

Core

Description

The **set_wksp** function sets the workspace environment variables to the specified address and pitch. The workspace memory is used by functions such as **fill_polygon** to draw a 1-bit-per-pixel image of the polygon before expanding it to the screen.

The area used for the workspace may be allocated from offscreen memory (a description of which is returned by the **get_offscreen_memory** function) or from from heap using **gsp_malloc**. It must be large enough to hold a 1-bit-per-pixel representation of the current screen coordinates with a power of 2 pitch.

Example

See **fill_polygon.**

Syntax

```
void styled_line(x1, y1, x2, y2, style, mode)
    short  x1, y1;     /* start coordinates                  */
    short  x2, y2;     /* end coordinates                    */
    long style;        /* 32-bit line style pattern          */
    short  mode        /* selects 1 of 4 drawing modes       */
```

Type Extended

Description The **styled_line** function uses Bresenham's algorithm to draw a styled line from point (`x1, y1`) to point (`x2, y2`). The line is a single pixel thick and is drawn in the specified line-style pattern.

- Arguments `x1` and `y1` specify the starting coordinates.
- Arguments `x2` and `y2` specify the ending coordinates.
- The last two arguments, `style` and `mode`, specify the line style and drawing mode.

Argument `style` is a long integer containing a 32-bit repeating line-style pattern. Pattern bits are used in the order 0,1,...,31, where 0 is the rightmost bit (the LSB). The pattern is repeated modulo 32 as the line is drawn. A bit value of 1 in the pattern specifies that the foreground color is used to draw the corresponding pixel. A value of 0 in one of the line style pattern bits means that the corresponding pixel is either drawn in background color (modes 1 and 3) or not drawn (modes 0 and 2). Four drawing modes are supported:

mode 0	Do not draw background pixels (leave gaps), and do load new line-style pattern from style argument.
mode 1	Draw background pixels in COLOR0, and load new lin-style pattern from style argument.
mode 2	Do not draw background pixels (leave gaps), and do re-use old line-style pattern (ignore style argument).
mode 3	Draw background pixels in COLOR0 and re-use old line-style pattern (ignore style argument).

The current style pattern used by the **styled_line** function is available by calling the **get_env** function. See the **get_env** function description for more information.

Example

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

CONFIG config;

main()
{
  long x1, y1, x2, y2, i, mask;

  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      /* draw spiral using styled line segments          */
      set_draw_origin(config.mode.disp_hres>>1,
      config.mode.disp_vres>>1);
      x2 = 0;
      y2 = -20 << 16;
      /* line style pattern                              */
      mask = 0x93E493E4;
      styled_line(0, 0, 0, 0, mask, 0);
      for (i = 1500; i > 0; --i)
      {
        x1  = x2;
        y1  = y2;
        x2 += y1 >> 4;
        y2 -= x1 >> 4;
        styled_line(x1>>16, y1>>16, x2>>16, y2>>16,
                    -1,2);
      }
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Syntax `void swap_bm()`

Type Extended

Description The **swap_bm** function swaps the pointers to the structures containing the source and destination bitmaps. This is useful for copying bitmap data offscreen, then at some later point, swapping the bitmaps and restoring the data back onscreen.

Example See **zoom_rect.**

Syntax `void synchronize()`

Type Core

Description The **synchronize** function ensures that the TMS340 completes all pending operations before it returns. TIGA supports two-processor execution and some conditions require them 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

```
int text_out(x, y, pString)
    short x, y;
    unsigned char far *pString;
```

Type Core

Description The **text_out** function renders the ASCII string pointed to by `pString` in the currently selected font using the current set of text drawing attributes. The string is a null terminated sequence of 8-bit ASCII character codes.

The starting position of the string is specified by arguments, `x` and `y`. Coordinate `x` is the position at the left edge of the string, and coordinate `y` is the position at either the top of the string, or the baseline, depending on the text alignment attribute set by a call to **set_textattr**.

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.

Syntax

```
int text_width(pString)
    unsigned char far *pString;
```

Type Extended

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.

Syntax `void transp_off()`

Type Core

Description The **transp_off** function disables transparency for subsequent drawing operations. The transparency mode set by default is mode 2, transparency on result equals zero (see **set_transp** function).That is, if the pixel operation involving source and destination pixels is 0, the destination pixel is not altered. This can be modified for TMS34020 systems by using the **set_transp** function.

Example See **set_dstbm.**

Syntax `void transp_on()`

Type Core

Description The **transp_on** function enables transparency for subsequent drawing operations. The transparency mode set by default is mode 2, transparency on result equals zero (see **set_transp** function).That is, if the pixel operation involving source and destination pixels is 0, the destination pixel is not altered. This can be modified for TMS34020 systems by using the **set_transp** function.

Example See call to **transp_off** in **set_dstbm.**

Syntax

```
void wait_scan(line)
   short line;   /* wait until this scan line is reached */
```

Type Core

Description The **wait_scan** function waits for a scan line on the CRT to be refreshed. This function does not return control to the calling routine until the specified line is scanned by the electron beam. Control is returned at the start of the horizontal blanking interval that follows the designated line. Scan lines are numbered in ascending order, starting with line 0 at the top of the screen. Only visible scan lines are counted.

This function can be used to synchronize drawing operations to the electron beam of a CRT display. For example, when drawing an animated sequence of frames, transitions from one frame to the next appear smoother if an area of the screen is not redrawn at the same time it is output to the CRT.

If argument line 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.

Syntax

```
void zoom_rect(ws, hs, xs, ys, wd, hd, xd, yd, linebuf)
    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          */
    short linebuf;  /* scan line buffer                     */
```

Type

Extended

Description

The **zoom_rect** function expands or shrinks a source rectangle on the screen or a linear bitmap to fit the dimensions of a destination rectangle on the screen. Horizontal zooming is accomplished by replicating or deleting rows of pixels. This type of function is sometimes referred to as a stretch blit.

The first four arguments define the source rectangle:

- The width `ws`
- The height `hs`
- The coordinates of the top left corner (`xs, ys`)

`ws` and `hs` must be non-negative.

The next four arguments define the destination rectangle:

- Width `wd`
- Height `hd`
- Coordinates of the top left corner of the rectangle (`xd, yd`)

`wd` and `hd` must be non-negative.

The final argument, `linebuf`, is a buffer large enough to contain one complete line of the display. The function uses the buffer as temporary working storage; The minimum buffer size (in bits) is the number of pixels per line times the number of bits per pixel. This buffer is a pointer to TMS340 memory and therefore must be allocated (by calling for example **gsp_malloc**) before invoking the function.

The **zoom_rect** function can be made to zoom from a linear bitmap by setting the source bitmap global using the **set_srcbm** function. This function only works if the pixel size of the source bitmap is the same as the screen. Thus, if the desired effect is to perform a **zoom_rect** on binary data, this must be done in two stages. First download the binary bitmap and use **bitblt** to expand this to the current pixel size to a linear bitmap. Then perform a **zoom_rect** on this linear bitmap onto the screen.

Example

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

CONFIG config;

short far arrow_shape[] =
{
  0x0003, 0x0007, 0x000F, 0x001F, 0x003F, 0x007F, 0x00FF,
  0x01FF, 0x03FF, 0x01FF, 0x007F, 0x00F7, 0x00F2, 0x01E0,
  0x01E0, 0x00C0
};

#define  ARROW_W    16
#define  ARROW_H    16

main()
{
  int  i;
  PTR  arrow_addr_bin, arrow_addr_col, buffer;
  long arrow_size;
  if (set_videomode(TIGA, INIT | CLR_SCREEN))
  {
    if (install_primitives() >= 0)
    {
      get_config(&config);
      /* set up linear bitmap with binary data for arrow  */
      arrow_size = ARROW_W * ARROW_H;
      arrow_addr_bin = gsp_malloc((arrow_size+7)/8);
      /* transfer shape data from host to gsp             */
      host2gsp(arrow_shape, arrow_addr_bin,
               (arrow_size+7)/8,0);
      /* set up a color bitmap for the arrow shape        */
      arrow_size *= config.mode.disp_psize;
      arrow_addr_col = gsp_malloc((arrow_size+7)/8);
      /* set the source bitmap to the binary arrow shape  */
      set_srcbm(arrow_addr_bin, ARROW_W, ARROW_W,
               ARROW_H,1);
      /* set destination bitmap to the color arrow shape  */
      set_dstbm(arrow_addr_col, ARROW_W *
               config.mode.disp_psize, ARROW_W, ARROW_H,
               config.mode.disp_psize);
      /* blit binary arrow and expand it to color arrow   */
      set_colors(LIGHT_CYAN, LIGHT_BLUE);
      bitblt(ARROW_W, ARROW_H, 0, 0, 0, 0);
```

```
        /* set the source bitmap to the color arrow          */
        swap_bm();
        /* set the destination bitmap to the screen          */
        set_dstbm(0l, 0, 0, 0, 0);
        /* set pixel processing to max as the results are    */
        /* normally better                                   */
        set_ppop(20);
        /* set up zoom-rect buffer to hold 1 scan-line       */
        buffer = gsp_malloc(config.mode.disp_hres *
                            config.mode.disp_psize);
        /* now zoom-rect the arrow to the screen             */
        zoom_rect(ARROW_W, ARROW_H, 0, 0, ARROW_W<<2,
                ARROW_H, 0, 0, buffer);
    }
    set_videomode(PREVIOUS, INIT);
  }
}
```

Chapter 4

Extensibility Through the User Library

Extensibility was a prime consideration in the design of the TIGA-340 Interface. TIGA-340 is extensible through the addition or manipulation of its **user library**, which is a collection of C-callable routines compiled or assembled with the TMS340 software toolset.

Graphics standards prior to TIGA have limited the software developer by providing a fixed set of graphics drawing primitives. In the rapidly changing graphics market, a fixed set of primitives is unacceptable. During the development of the TIGA graphics interface, incorporating extensibility into the standard was a major design goal. The result was the linking loader described in this chapter, which, along with a collection of routines, can be used by an application or device driver to install custom-made graphics primitives.

Section **Page**

4.1 Dynamic Load Module 4-2
4.2 Generating a Dynamic Load Module 4-4
4.3 Installing a Dynamic Load Module 4-7
4.4 Invoking Functions in a Dynamic Load Module 4-10
4.5 C-Packet Mode 4-13
4.6 Direct-Mode 4-18
4.7 Downloaded Function 4-32
4.8 Example Programs 4-36
4.9 Installing Interrupts 4-44
4.10 TIGA Linking Loader 4-47

4.1 Dynamic Load Module

The key to TIGA's extensibility is the Dynamic Load Module (DLM). This module is a collection of C or assembly routines written by the application or device driver programmer, and linked together to form the module. The DLM is downloaded at run time into TMS340 memory and integrated with the TIGA graphics manager. Once downloaded, each routine contained within the module is callable using the same conventions as the TIGA core or extended primitives.

TIGA currently supports two types of dynamic load modules:

- Relocatable load module (RLM), and
- Absolute load module (ALM).

The routines which compose a dynamic load module can be either standard C-type functions callable from either the host processor or from the TMS340, or interrupt service routines called on reception of an interrupt via the TIGA standard interrupt handler.

4.1.1 Relocatable Load Modules

Relocatable Load Modules (RLMs) are produced directly using 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 *TMS34010 Assembly Language Tools User 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 extended primitives, which are resolved when they are loaded. Furthermore, they contain all the necessary symbol information stored after loading a symbol table file so that subsequent RLMs that are loaded may reference the functions in another RLM. The installation of an RLM is performed by invoking the **install_rlm** function which, in turn, invokes `TIGALNK`, the linking loader.

4.1.2 Absolute Load Modules

Absolute Load Modules (ALMs) are created from relocatable load modules by calling the **create_alm** function. This function in turn calls the linking loader to link and save (instead of link and load) the resultant TMS340 memory image. `TIGALNK` uses the TIGA heap management routines to allocate a space in TMS340 memory where the ALM will be loaded. `TIGALNK` then links and relocates the module to the area starting address in heap. Thus, the ALM can only be loaded into this one area in memory. The heap area for the module is then freed by the **create_alm** routine. 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, this can be achieved

by always initializing heap prior to calling **create_alm** and then reinitializing heap when the module is installed. Heap initialization can be performed by calling **set_videomode** with an `INIT` style.

The reason for incorporating ALMs into TIGA is that installation of RLMs requires the application or device driver written for TIGA to call the **install_rlm** function, which in turn invokes the linking loader. This requires about 70—100K bytes of free main host memory, depending on the symbol table size of the module being installed. For many applications this is the most direct and flexible method for installation of functions, as the module can be relocated and the symbols accessed by subsequent module loads. However, for certain applications and application drivers, not enough memory is available to use this method. An example of this is the AutoCAD driver, ADI. By the time AutoCAD calls the ADI, all available PC memory has been appropriated, leaving no room for **install_rlm** to invoke the linking loader. Using the ALM, no memory is required. The **install_alm** function contains only a short piece of code to load the module into TIGA, since no external linking or relocation needs to be made. The **create_alm** function can be called when ADI is installed (at boot time). Because this is prior to AutoCAD being invoked, the PC memory is free to invoke `TIGALNK` to create the ALM file.

When loading an ALM, 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. A further restriction with ALMs is, since the symbol information is also no longer available within the file (as it is with RLMs), that modules loaded subsequently cannot reference functions in an ALM.

4.2 Generating a Dynamic Load Module

A TIGA dynamic load module consists of the following three parts:

- A collection of C and/or assembly routines, some (or all) of which are to become TIGA extensions or interrupt service routines.
- TIGAEXT section declaration. Required only if TIGA extensions are being declared.
- 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 *TMS34010 C Compiler Reference Guide* and the *TMS34010 Assembly Language Tools User's Guide.* If the user library is to contain functions written using TMS34010 assembly code then certain guidelines need 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 Section 5.4 of the *TMS34010 C Compiler Reference Guide.*

Dependent 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, the 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.

4.2.1 TIGAEXT Section

The TIGAEXT 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 declared 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
```

4.2.2 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 declared 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
   .long    1                    ;interrupt number 1;
   .long    _my_int10
   .long    10                   ;interrupt number 10;
   .text                         ;end section
```

Note that the TIGAEXT and TIGAISR sections must contain the exact number of declarations for the external functions to be installed. This is because `TIGALNK` uses the length of these sections to determine the number of declarations.

4.2.3 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`	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

Step1: Assemble and/or compile all of the source files:

```
cc myfuncs tigaext myints tigaisr ↵
```

This produces four object files:

```
myfuncs.obj
tigaext.obj
myints.obj
tigaisr.obj
```

Step 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_init00".** is displayed. It can be ignored.

4.3 Installing a Dynamic Load Module

To invoke the commands installed in a dynamic load module, it must first be installed 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. TIGALNK uses the environment variable to find the DLM. The actual installation procedure differs from RLM to ALM.

4.3.1 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 explains how to install the example described in the previous section, EXAMPLE.RLM.

Example 4–1.

```
#include <tiga.h>

main()
{
   short module;

   /* initialize TIGA */
   if (!set_videomode(TIGA, INIT))
   {
      printf("Fatal Error - TIGA not installed\n");
      exit(0);
   }
   /* attempt to install module */
   if ((module = install_rlm("EXAMPLE")) < 0 )
   {
      printf("Fatal Error - couldn't install Example RLM\n");
       printf("Error code = %d\n", module);
      exit(0);
   }
   /* code to invoke the module functions */
      :
      :
   set_videomode(PREVIOUS, INIT);
}
```

The **install_rlm** function is invoked with the filename of the RLM file. Either a full pathname can be given, or just the final part of the filename, when either the current directory is used or that directory set by the TIGA environment variable. A default extension of .RLM is assumed unless one is given. The **install_rlm** function will return either the module id for the RLM, which will be used when the functions are invoked, or an error code if some error occurred. Error codes are negative values, module identifiers are always positive (including zero).

4.3.2 Installing an Absolute Load Module

An absolute load module must first be created from a relocatable load module. Below is an example program written in Microsoft C that explains how to create an ALM from the example described in the previous section.

Example 4–2.

```
#include <tiga.h>

main()
{
   register return_code;

   if(!set_videomode(TIGA, INIT))
   {
      printf("Fatal Error, TIGA interface not installed.\n");
      exit(1);
   }
   /* attempt to create the module */
   return_code = create_alm("EXAMPLE", "EXAMPLE");
   if (return_code < 0)
   {
       printf("Fatal Error in creation of 'alm' file\n");
       printf("Error code = %d\n", return_code);
      exit(1);
   }
   /* further initialization code                                    */
      :
      :
   set_videomode(PREVIOUS, INIT);
}

init_driver()
{
   register return_code;

   if(!set_videomode(TIGA, INIT))
   {
      printf("Fatal Error, TIGA interface not
                installed.\n");
      return(0);
   }
   /* attempt to install the module                                  */
   return_code = install_alm("EXAMPLE");
   if (return_code < 0)
   {
      printf("Fatal Error in installation of 'alm' file\n");
      printf("Error code = %d\n", return_code);
      return(0);
   }
   /* code to invoke the module functions                            */
      :    :
   set_videomode(PREVIOUS, INIT);
}
```

The example assumes that at the time the program is run initially, TIGALNK 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. **create_alm** produces an ALM file only if it does not already exist. This generally saves the program the unnecessary time of recreating the ALM every time the program is run. If the application requires to create 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 performed after invoking the init_driver function. 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 re-initialized and the ALM is installed. **install_alm** does not invoke TIGALNK but uses a trivial loader to move code from the host PC file into TMS340 memory.

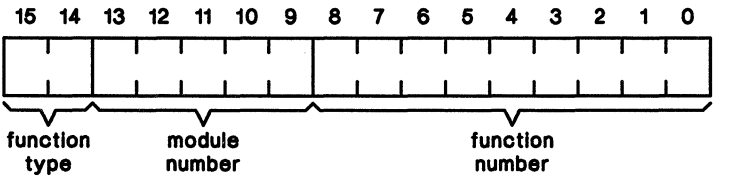
4.4 Invoking Functions in a Dynamic Load Module

The process of invoking a function in a DLM is done in two parts. The first part involves the selection of the function, which is described in this section. The second part is the actual invocation of the function and the passing of its parameters from the host to the TMS340. That part is described in subsequent sections.

4.4.1 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 split into the following fields, as Figure 4–1 shows:

Figure 4–1. 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 for TIGA core primitives.
   30 for TIGA extended primitives 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 will be 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 primitives are always installed at initialization time as module number 31. Likewise, the DLM that

contains the TIGA extended primitives 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 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 that they are declared in the TIGAEXT section within a module. For the example, described in Section 4.3 `my_func1` would be designated function number 0, and `my_func2` would be designated function number 1.

4.4.2 Using Macros in Command Number Definitions

The definition 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.H` include files to enable the command number of a function to be specified without referencing the individual bits in the command number. These are the macros:

```
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)
```

`CORE_CP` and `CORE_DM` select C-packet or direct-mode functions with a module number of 31 (for the TIGA core primitives). Similarly, `EXT_CP` and `EXT_DM` select C-packet or direct-mode functions with a module number of 30 (for the TIGA extended primitives). `USER_CP` and `USER_DM` are used for 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 an application hard-codes the bits in a command number, there is a risk of incompatibility with future versions of TIGA.

4.4.3 Passing Parameters to the TIGA Function

The invocation of a TIGA function can be done in two ways, depending on the type of function call that is made. These are C-packet or direct-mode calls.

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; thus it is very easy 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 be recompiled under the TMS340 C compiler. Any parameters it received on the host side will be passed from host to 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 local to the host. To do this, however, extra data must be sent along to the TMS340 describing the type and size of each parameter.

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 just 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 will be developed using C-packet initially. Those functions that are more time critical would be modified to use direct-mode. The changes to the source code of an extension to change it from C-packet to direct-mode are not that significant as will be seen from examples given later. The following sections give a complete description of C-packet and direct modes.

4.5 C-Packet Mode

To invoke a User extension using C-packet mode, three pieces of information need to be supplied:

- The type of call the function uses
- The function's command number
- Description of the function arguments

4.5.1 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 which pass pointers to data that is modified indirectly by the function called.

```
draw_a_line(x1, y1, x2, y2)      would use cp_cmd
poly_line(10, &point_list)       would use cp_cmd
i = read_point(x, y)             would use cp_ret
copy_mem( &src, &dst, len)       would use cp_alt
```

An additional set of entry points is used when the argument list has the potential of being 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, with the added capability of allocating additional space for large amounts of arguments data, at a cost of speed performance. These entry points should be avoided when the user knows that the argument length of the function in question will not exceed the maximum size dictated by the communication buffer's data size (which is a field of the CONFIG structure returned by **get_config**).

4.5.2 The Command Number

Section 4.4.1 on page 4-10 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.

4.5.3 Description of Function Arguments

To call the desired function, each of that function's arguments must be understood by the graphics manager, so 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 by the use of additional defines in the TIGA.H include file to represent the different data types. Below is a list of the currently supported data types:

_WORD(a)	Immediate WORD argument a
_SWORD(a)	Immediate signed WORD argument a
_DWORD(a)	Immediate Double WORD argument a
_BYTE_PTR(b,a)	BYTE array ptr a with b elements
_WORD_PTR(b,a)	WORD array ptr a with b elements
_DWORD_PTR(b,a)	DWORD array ptr a with b elements
_STRING(a)	Null-terminated string ptr 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

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 need 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.

4.5.4 C-Packet Examples

The exact argument list of the C-packet entry points is as follows:

```
entry_point_name(CMD_ID, num_packets, packet1, ... , packetn)
```

where:

```
cm_number              command number
npackets               number of C type packets
packet1...packetn      Packet data (see below)
```

Below are some examples of user extensions. These examples are not supplied TI-extended primitives.

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, `WORD,n` 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 function initializes the array pointed to by `&matrix` indirectly. (Use **cp_alt**)
- The function's command number was stored in `CMD_ID`.
- The function has 1 argument which points to a 4 x 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))
```

4.5.5 Overflow of the Command Buffer

When a command of any kind (primitive 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 very 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 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**) which 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 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 and thus some speed overhead is incurred using this technique. 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, which 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.

4.6 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 them 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 performance of invoking TIGA extensions from TMS340. They are not meant (although they could be used) for functions that are called from other downloaded functions from the TMS340 side. Such 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 has to ensure that the data being transferred does not overflow the command buffer.

A further difference between C-packet and direct mode was 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, specifying whether it is immediate or not, and its size. It is the direct-mode entry point itself that determines what format the parameters can be specified in, and, in turn, how these parameters are received in the TMS340 communication buffer. In the following sections is a list of the direct-mode entry points and the parameterization of their arguments.

4.6.1 Standard Command Entry Point

```
void dm_cmd(cmd_number, length, arg1,..,argn);
   short cmd_number;
   short length;
   short arg1...argn;
```

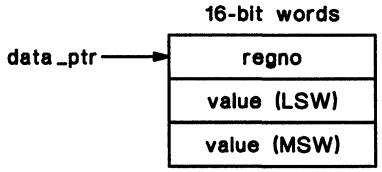
This command is the most commonly used for direct-mode commands in the TIGA system. It has a single `length` argument and an arbitrary-length list of immediate value arguments; it has no return value. The `length` specified is the number of 16-bit words that are sent; thus to send a *long*, `length` 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, since three16-bit words are pushed onto the stack (2 of them being the MSW and LSW of `value`).

```
#define poke_breg(regno,value)                  \
   dm_cmd(POKE_BREG,3,(short)(regno),(long)(value))
```

The data in the communication buffer looks as follows:

***Figure 4–2. 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
```

4.6.2 Standard Command Entry Point with Return

```
unsigned long dm_ret(cmd_number, length, arg1, ... , argn);
   short cmd_number;
   short length;
   short arg1...argn;
```

This command is similar to **dm_cmd** described in Section 4.6.1 on page 4-18. 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. The value is returned 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 primitive **get_nearest_color** uses this entry point. It sends 4 bytes of red, green, blue, and intensity (which are all promoted to shorts by the C-compiler), returning a long index into the palette.

```
dm_ret(GET_NEAREST_COLOR, 4, (short)(r), (short)(g)
                             (short)(b), (short)(i))
```

4.6.3 Standard Memory Send Command Entry Point

```
void dm_psnd(cmd_number, length, ptr)
   short        cmd_number;
   short        length;
   char far     *ptr;
```

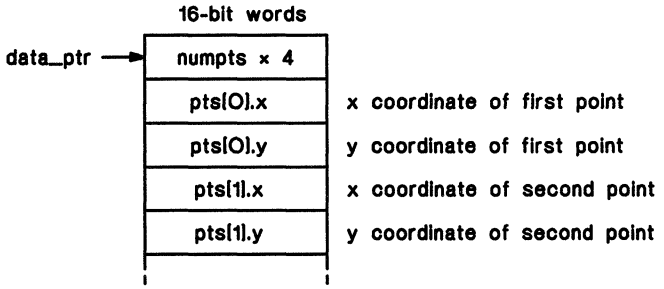
This command is used to call functions that require information in the form of an array or structure. Note that in this case the length specified is in bytes, not 16-bit words as in the previous two entry points. The `ptr` argument is a far pointer into host memory. The contents of this pointer are down loaded into the communication buffer.

The TIGA extended primitive **draw_polyline** uses this entry point. Notice that the `numpts` is multiplied by 4 since every point consists of two coordinates (x and y), each of which is 2 bytes long.

```
#define draw_polyline(numpts,pts)                              \
        dm_psnd(DRAW_POLYLINE,(short)(4*(numpts)),             \
                              (short far *)(pts))
```

The data in the communication buffer looks as follows:

***Figure 4–3. 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 primitive 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
```

4.6.4 Standard Memory Return Command Entry Point

```
unsigned long dm_pget(cmd_number, length, ptr)
   short        cmd_number;
   short        length;
   char far     *ptr;
```

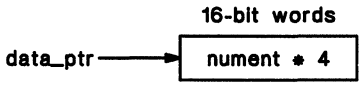
This command is used to call functions that return information in the form of an array or structure. The length (in *BYTES*) is sent as the first element in the command buffer and invokes the TMS340 function. The function writes the return data into the communication buffer at the word following the length.

The TIGA core primitive **get_palet** uses this entry point. Notice that the `nument` parameter is multiplied by 4 since each palet entry consists of a red, green, blue, and intensity byte.

```
#define get_palet(nument,pal)                    \
   dm_pget(GET_PALET,(short)(4*(nument)),(char far *)(pal))
```

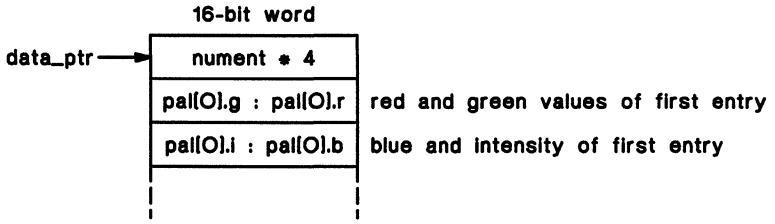
The data in the communication buffer contains the one word of data before the function is invoked:

***Figure 4–4. Data Structure Before Invoking* dm_pget**



Following the invocation of the buffer the communication buffer contains:

***Figure 4–5. Data Structure After Invoking* dm_pget**



4.6.5 Standard String Entry Point

```
void dm_pstr(cmd_number, ptr)
   short        cmd_number;
   char far     *ptr;
```

This command 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.

An example for this entry point can be found in the communication entry points tests in the TIGA release (in directory `tigapgms\tests\coms`).

4.6.6 Altered Memory Return Command Entry Point

```
unsigned long dm_palt(cmd_number, length, ptr)
   short        cmd_number;
   short        length;
   char far     *ptr;
```

This command is used to send and return 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 it completes the data is returned back into the host memory pointed to by `ptr`.

An example for this entry point can be found in the communication entry points tests in the TIGA release (in directory `tigapgms\tests\coms`).

4.6.7 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;
   char far     *send_ptr;
   short        return_length;
   char far     *return_ptr;
```

This command 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 Section 4.6.6 except that data is returned to a different area of host memory.

An example for this entry point can be found in the communication entry points tests in the TIGA release (in directory `tigapgms\tests\coms`).

4.6.8 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      */
   char far  *ptr1;      /* pointer data                      */
   short     cnt2;
   char far  *ptr2;
```

This command 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.

An example for this entry point can be found in the communication entry points tests in the TIGA release (in directory `tigapgms\tests\coms`).

4.6.9 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 */
   char far  *ptr1;           /* pointer data                 */
   short     cnt2;
   char far  *ptr2;
```

The command **dm_pret** is similar to **dm_pcmd** except that it returns a standard C value in the DX:AX registers.

An example for this entry point can be found in the communication entry points tests in the TIGA release (in directory `tigapgms\tests\coms`).

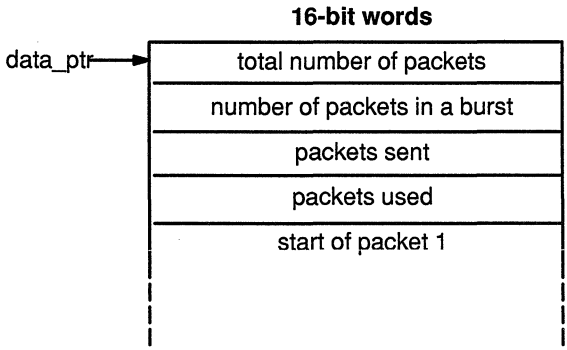
4.6.10 Poly Function Command

```
void dm_poly(cmd_number, packet_number, packet_size, packet_ptr)
   short      cmd_number;
   short      packet_number;
   short      packet_size;
   char far   *packet_ptr
```

This 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 is used for operations that require a large amount of data to be transferred and when a certain degree of parallelism is possible; that is, some of the data being sent can be processed while the rest is being sent down. For example, the ADI redraw function used by the TIGA AutoCAD driver uses this entry point to draw some vectors while others are being sent by the host.

The command buffer used by the communication driver to download the parameters is turned into a circular queue of packets. The command buffer contains the following:

***Figure 4–6. Data Structure of* dm_poly**



This 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 4–3.

```
;---------------------------------------------------------------;
;  TIGA - Graphics Manager function                             ;
;---------------------------------------------------------------;
; Usage:  Example GSP shell routine with dm_poly entry point.   ;
;---------------------------------------------------------------
;  Include GSP register definitions
        .copy       gspregs.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
```

4.6.11 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;       /* Number of immediate short
                                   words to send           */
  unsigned short sData;         /* First short word of data
                                   to send                 */
    :
  unsigned short ItemSz;        /* Size of items that follow
                                   (in bytes)              */
  unsigned short nItems;        /* # of items that follow  */
  char far *pData;              /* Pointer to data to send */
```

This entry point is similar to dm_poly; it is used for operations that require a large amount of data items to be transferred, and the TMS340 has the ability to operate on 1 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 using a specific set of rules. When the TMS340 function is called, it will receive a data pointer in TMS340 memory. The data at that address will consist of the immediate data values. The poly data that is sent in bursts by the host requires special processing and communication protocol in order to be received. In order to isolate this 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 2 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`	Command number of the polypixel function.
`2`	Specifies that two immediate arguments follow: `color` and `rop`.
`color`	First immediate argument.
`rop`	Second immediate value.
`4`	Each item is a point, which in this case is two words. The first specifies the X coordinate, the second specifies the Y. The size of the item is therefore 4 bytes.
`nPoints`	Specifies the number.
`pData`	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 GSP function which uses the
;  dm_ipoly host entry point.
;-------------------------------------------------------
;  Include GSP register definitions
        .copy       gspregs.inc
;  Include macros
        .mlib       gspmac.lib
;  Declare globals
        .globl      _PolyPixel
;  External References
        .globl      _srv_ipoly
;  Polypixel argument definition
aCOLOR  .set    0
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 340 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
```

4.7 Downloaded Function

User extended routines and interrupt service routines contained in a dynamic load module have the ability to access functions or globals which were previously installed into TIGA. This includes the core primitives and the TI extended primitives (provided that they have been installed by the application). Note that certain primitives are **host-only** primitives and cannot be invoked by a dynamically loaded routine. These are

create_alm
create_esym
field_extract
field_insert
flush_esym
flush_extended
get_isr_priorities
get_modeinfo
get_videomode
gsp2host
gsp2hostxy
host2gsp
host2gspxy
install_alm
install_primitives
install_rlm
install_usererror
set_config
set_timeout
set_videomode
synchronize

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 TMS340 primitive 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 primitives can be invoked from the TMS340 side since some require access to host side data structures, such as those concerned with the linking loader. Two include files containing the graphics manager core functions and extended functions (`gsptiga.h` and `gspextnd.h`) are supplied for this purpose. This capability has the advantage that an application can be written and debugged on the host side 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 containing the graphics manager global variables (`gspglobs.h`) is supplied for this purpose. The file is shown in the figure below, which details 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 ENVCURS envcurs;            /* cursor environment         */
extern ENVTEXT envtext;            /* text environment           */
extern MODEINFO *modeinfo;         /* operating mode info        */
extern MONITORINFO *monitorinfo;   /* monitor timing info        */
extern MODULE Module[32];          /* function module descr.     */
extern OFFSCREEN_AREA *offscreen;  /* pointer to current data    */
extern PAGE *page;                 /* pointer to current data    */
extern PALET palet[];              /* current palette in use     */
extern PATTERN pattern;            /* current pattern info.      */
extern char *setup;                /* current setup pointer      */
extern short sin_tbl[];            /* sine look-up table         */
extern long stack_size;            /* declared in link file      */
extern long start_of_dram;         /* declared in link file      */
extern FONT sysfont;               /* system font                */
extern PACKET *sys_free;           /* pointer to free packets    */
extern long *sys_memory;           /* pointer to heap packets    */
```

Where these variables reference a specific type of declaration, such as `PALET`, the include file `gsptypes.h` should also be included to define this type of declaration.

4.7.1 Register Usage Conventions

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

- Status register fields FE1 and FS1 must be restored. Fields FE0 and FS0 need not be restored.
- All A-file registers except A8 must be restored. A14 should not be used as a temporary variable by a user function. It must always contain a pointer into the C parameter stack, because an interrupt service routine (ISR) may interrupt a user function, and that ISR may call a C function using the C stack.
- In general, all B-file registers must be restored. However, certain B-file registers may be altered by attribute control functions that update parameters such as COLOR0 and COLOR1.
- In general, I/O registers CONTROL, DPYCTL, CONVDP, and INTENB should be restored before returning to the calling routine. However, some I/O register bits may be altered by attribute control functions that update parameters such as the plane mask, pixel processing operation, or transparency flag. These register bits typically are not changed by graphics output functions.

Upon entry to a downloaded extension, certain registers are in a known state and contain well-defined parameters. These assumptions cannot be made of interrupt service routines, since 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.

4.7.2 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 registers:

- 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.
- The four host interface registers (HSTADRL, HSTADRH, HSTDATA, and HSTCTL).

4.8 Example Programs

The TIGAPGMS directory that is shipped with TIGA contains several example functions. To gain the maximum benefit from the following sections of this guide, they should be read in conjunction with a hard copy of the listings of the source code of those examples.

4.8.1 Stars Example

The TIGA release disk with the example programs contains a stars directory that is an example of the use of C-packet and direct-mode extensions of TIGA. This demonstration program may be familiar because it has been ported to many different graphics environments. It basically consists of moving through a three-dimensional galaxy in which stars grow larger as they are approached and then disappear off the edge of the screen. As they do so, new stars are created in the distance. This scenario is performed in four ways. First, using host calls to TIGA extended primitives to perform the drawing of the stars. Second,where the host calls to a custom TMS340 C routine using the C-packet communication mechanism. Third, where the host calls a custom direct-mode C routine. Finally, where the direct-mode routine has been re-written using TMS340 assembly code. The stars program prints out the elapsed time to call these different functions, and the time saving is evident. It should be noted at the outset that this example, though demonstrating the capabilities of downloading TIGA extensions, is very artificial. The time savings in a real application is typically better than with this example, especially when the downloaded function performs something a little more substantial than drawing a few pixel-wide stars.

This example consists of the following source files:

`stars.c`	Main program (Microsoft-C)
`star.h`	Insert file containing type definitions and external references
`data.c`	Star shapes
`starscp.c`	C-packet extension to draw a star (TMS340-C)
`starsdm.c`	Direct-mode extension to draw a star (TMS340-C)
`starsasm.asm`	Direct-mode extension to draw a star (TMS340-assembly)
`starsgsp.asm`	TIGAEXT file describing extension routine names

The routine that forms the downloaded extension is one that draws a single star. The four versions of it are

`draw_star_host` in file `stars.c`

`draw_star_cp` in file `starscp.c`

`draw_star_dm` in file `starsdm.c`

`draw_star_asm` in file `starsasm.asm`

A comparison between the `draw_star_host` and `draw_star_cp` shows that besides the function name, the two are identical (apart also from the more important fact that one is compiled in Microsoft C and the other in TMS340 C). This underlines an important advantage of TIGA: that it is possible to take an existing application running under Microsoft C, move a function to the TMS340 side, and invoke it with the same parameterization as if it were locally resident and obtain an immediate speed improvement, as can be seen from running this program. Further speed improvement can be accomplished with just a little more work.

A comparison between the `draw_star_cp` and `draw_star_dm` functions shows that after the first four lines of the direct-mode version, the functions are again identical. The only difference between them highlights the fundamental difference between C-packet and direct-mode functions. Direct-mode functions receive parameters, just as the host downloads them, as sequential items in a communication buffer. The direct-mode function receives a single parameter, which is a pointer to the data area of the communication buffer where the data has to be fetched. In the C-packet case, the functions parameters are sent down in packets describing the size and type of the data being sent. Then a C-packet interpreter parses these packets and pushes parameters onto the stack where the C-packet extension expects to find them. This enables the C-packet routine to be called just as if it were local to the host program, but it incurs the additional time overhead of sending more information in the packet than the data itself.The direct-mode extension eliminates this overhead but puts a very slight extra burden on the extension to fetch its own data. Because the transition from C-packet to direct mode is very simple, it is expected that most applications will use C-packet to start and then move to direct mode for those time-critical functions that need to be optimized.

The final function `draw_star_asm` also uses direct mode, but the function no longer uses the TIGA **bitblt** function to draw the star. Instead, it re-codes the whole function in TMS340 assembly language. This function requires the most effort form the application programmer to produce. Not every extension should use this approach, but there is a well-defined route that allows an easy progression from host alone, through the simpler approaches

of C-packet and direct mode, to the custom assembly function. The custom assembly function allows a programmer to develop applications quickly, optimizing time-critical functions to the limit.

4.8.1.1 Generating the Downloadable Extension File

The extension consists of the files containing the three downloaded subroutines, a data file containing the star shapes, and a special file containing the TIGAEXT section (`starsgsp`). The latter declares the list of downloaded functions to be installed in a specific order so that they can be referenced later. The order in which the functions appear in the TIGAEXT section define the command numbers used when the functions get invoked thus: `draw_star_cp` has a command number of 0, `draw_star_dm` has a command number of 1, and `draw_star_asm` has a command number of 2. All these files are linked together using the TMS340 linker (in the make file) with the `-cr` and `-r` options.This produces the relocatable load module `starsgsp.rlm`. Note that in building the rlm file the linker produces the message **>> warning: entry point symbol _c_int00 undefined**. This can be ignored.

4.8.1.2 Installing the Downloadable Extension File

The initialization routine of `stars.c` performs the installation of the rlm file by calling **install_rlm**. If the call to **install_rlm** returns a negative result, an error occured; if it returns a positive number or zero, it is the module number of the installed group of functions. Every installed RLM receives its own module id. The first id is 0, the second id is1, etc. Because this application was invoked with **set_videomode** style of `INIT`, which initializes the heap, and as a by-product of this, deletes all extended primitives that were installed, the application can be assured that the id of the first set of installed extensions is 0. Thus, the module identifier (`mod_star`) can be a constant 0 in the program.

The expected approach for the common mode of operation is that an application flushes out all extended primitives and downloads a single RLM file containing all its extensions. This approach has some minor speed improvements over the more general approach where the module number is not known until runtime and the command number needs to be stored in a variable.

Note that no directory is specified in the filename of the downloadable extension. This is not a problem for a development environment because the current directory is the one searched first and the one where the extension is stored. In a production mode where different TIGA applications and drivers are stored in different directories, the user should set up a TIGA library directory that is pointed to by the `-l` field of the TIGA environment variable.

4.8.1.3 Invoking the Downloadable Extensions

The three `update_cp`, `_dm` and `_asm` functions are the ones that actually need to invoke the extensions. The extensions are invoked through the use of TIGA communication entry points; `cp_cmd` for the C-packet call and `dm_cmd` for the two direct-mode calls. To make the invocations more readable, these calls are #defined to function calls that look like regular host functions.

All the communication entry points take as a first parameter the command number of the TMS340 function to be invoked. The entry point consists of a function command number indicating the order in which the function appears in the TIGAEXT section, ORed with the module number (which from the previous section is known to be 0 and so can be ignored). Following that are the commands parameters:

1) **C-Packet** The number of `PACKETS, 4`, followed by 4 `WORD` packets with parameters of the actual parameter data. The `WORD` macros build a packet containing the data size and type for the C-packet handler to interpret.

2) **Direct Mode** The number of 16-bit words, 4, followed by four 16-bit words pushed onto the stack.

4.8.2 Curves Example Program

The curves example program draws a series of graphs of mathematical functions. It is similar to the stars example described above. The major difference is that it installs the extensions as separate modules. Thus, rather than assuming the module id is 0, the module id that is returned from the 3 calls to **install_rlm,** is stored in the global variables `module_draw_cp, _dm, _fp`.

The downloadable extensions are all passed an array of (x,y) coordinates which they draw. The points are produced by a `generate_curve` function, which involves a series of calls to a runtime support math function involving floating point arithmetic. The final list of points are, for the C-packet and direct-mode calls scaled, to screen coordinates.

The `_fp` case is a bit different. It passes a list of floating point values which are scaled to the screen by the extension function. It illustrates how floating point values can be passed through TIGA. Currently TIGA does not support the passing of floating point parameters directly. The reason is not due to TIGA but to the fact that TMS340 floating point numbers are not in IEEE format (and require conversion to and from IEEE format). The floating point extension contains the source code of IEEE format conversion routines, which can be used for this. The TMS340 floating point format will be available in IEEE format in the near future, and direct floating point support will then be put into TIGA.

4.8.2.1 Speed Optimization of Parallel Processing

The timing of the extensions is done in two ways: First, time is taken directly following the functions being invoked (without synchronization). This gives a much shorter time than the second set of timings, which are taken after a call to the **synchronize** function. In the first case the time measured is that taken by the host. Second is the time taken by both the host and the TMS340. The TMS340 is a coprocessor and can offload much of the graphical processing from the host and do it much faster. However, the time saved by an application also depends on utilizing both processes in parallel. If the application is written so that the host is simply waiting for the TMS340 to complete, then little or no time may be saved.

When the application can perform an operation, say the calculation of the next set of graphical drawing coordinates, while the TMS340 is drawing, is whent the best performance improvements are achieved. This is important when choosing the communication entry points to use. Entry points that return values, such as **cp_ret, cp_alt, dm_ret, dm_ptrx**, etc. all cause the host to wait until the TMS340 is finished. If a downloadable extension, which takes a long time to execute, is to return status information, it is perhaps bet-

ter to split the function into two. One to do the drawing, the other to return the status. That way the host calling function can invoke the first function without waiting, then go on to perform some calculations that are not dependent upon the status, then call the status function some time later. This utilizes both processors more efficiently.

4.8.2.2 *Invoking Downloadable Extensions*

The invocation of the three downloadable extensions brings out some further points that were not covered by the stars example. Referring to the #defines for the communication entry points:

- **C Packet:** This illustrates the passing of pointers in C-packet mode. Notice that the third parameter `c` is also used to determine the size of the second parameter `b`. This is a very typical case. Unless a pointer is pointing to a fixed size structure, a parameter is needed to tell the calling function how big the array being passed really is. This parameter can be used to tell the communication driver how much to send. The value may require scaling, as in this case, c refers to the number of vertices being passed, but as each vertex is made up of two 16-bit coordinates (x, y), the number of WORDS to be sent is c*2.

- **Direct Mode:**The direct-mode entry point **dm_pcmd** allows the transfer of combinations of immediate data and pointers. The parameters are:

  - `l` number of immediate words being sent
  - `a` immediate word
  - `l` number of pointers being sent
  - `c*4` number of BYTES to be sent in the pointer
  - `b` (far) pointer to the data

  Note that the `c` parameter is not sent explicitly as an immediate word. This is because since it is used as a count for the pointer data `b`, it appears in the communication buffer multiplied by 4. Because the downloaded extension can recreate it by a simple shift there is no point in sending it down twice. Notice too that the size is sent in `BYTES` not `WORDS` as it is for C-packet. What ends up in the communication buffer is best seen by consulting the routine `draw_curve_dm`.

- **Floating Point:** Although the floating point uses a DOUBLE_PTR which looks as a TIGA macro, it is defined in the curves program. TIGA treats doubles (which are the only floating point parameters passed to routines, floats are always promoted to doubles) as an array of 4 unsigned 16-bit words. Consult the function `draw_curve_fp` to see how the conversion of floating point occurs. Although this example does not show it, the `ALTDOUBLE_PTR` can return floating point values from TIGA

extensions. These floating point values require reconverting back into IEEE format using `gsp2ieee`.

4.8.3 ADI Driver Example

The ADI directory contains the source of an example driver for ADI release 3.1 which works with AutoCAD release 9. There is no discussion on how to write an ADI driver, since this is fully covered in the ADI Driver Development Kit that can be obtained from AutoDESK Corp. The only details given here are regarding certain features of TIGA utilized in this driver.

4.8.3.1 Installing an ALM

The requirement for an ALM has been fully discussed in Section 4.1.2 . The main program (in file `adi.c`) makes a call to **create_alm** to create the ALM from the RLM that is shipped with the driver. Then a trial call to **install_alm** is made, to see if there will be any problems in installing the ALM (for example, not enough heap) later. Toward the end of the main program, a call is made to the initialize function in the `adiasm.asm` file, to turn the program into a terminate-and-stay-resident task. Note that previously a call is made to **set_videomode(PREVIOUS,** to end the TIGA session and return the board to an IBM graphics mode, such as EGA.

Later, when AutoCAD is invoked and a drawing is edited, AutoCAD makes a call to the `pinit` function (in `adi.c`). The `pinit` function calls **set_videomode(TIGA**, to start the TIGA environment and then calls **install_alm** to install the ALM.

4.8.3.2 Linking the Extended Primitives with the User RLM

There is no call to **install_primitives** in the ADI driver; although, some of the extended primitives are used in the driver. These primitives are linked in with the ADI driver primitives and are loaded simultaneously. Thus, in the `adiext.asm` file, references are made to TIGA extended functions such as **draw_line, bitblt**, etc. These primitives are supplied in TIGA both as an RLM and in the form of a library that can be linked. This library is referenced when the ADI RLM is created (see `adiext.cmd`).

The advantage of linking the extended primitives with a user load module is that only those functions that are needed by the application are included, freeing up valuable space in TMS340 memory. Also, the time to load the functions is reduced. However, since the extended primitives are being loaded into a user module, their command numbers need to be modified. This is why their definitions appear in a header file `adiext.h` and why the TIGA extended primitive definitions appear in a separate include file `extend.h` rather than in `tiga.h`. This enables their command numbers to be changed without the need to edit the standard TIGA include files.

4.9 Installing Interrupts

Interrupt service routines contained within a dynamic load module must be written as a function called with no arguments; that is, the last instruction should be a RETS 0 instead of a RETI. This is because the TIGA graphics manager provides a general interrupt handler that invokes the interrupt routines only if they are enabled. This handler performs the actual RETI instruction to return from the interrupt.

In addition, the handler also provides for chaining of multiple interrupt service routines on a single interrupt level. This is vital for the TMS340 processor, which often has more than one display interrupt active. For instance, the graphics manager provides three interrupts to control a hardware emulated cursor, page flipping, and wait-scan, all using the display interrupt.

The interrupt service routines must be 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 Section 4.2.2 on page 4-5. 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. The interrupt priority can be retrieved for each ISR declared in the TIGAISR section, after a successful installation, by performing a call to **get_isr_priorities**. This routine returns an interrupt priority for each ISR in their order of declaration in the section. Each interrupt is also installed in a disabled state and must be explicitly enabled by the programmer.

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.

Note that it is possible for a downloaded extension to be executed from the host and, in turn, set the traps to its own server 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 primitives such as the cursor functions. If equivalent support is not given to these functions, as provided by the global interrupt handler, certain TIGA primitives may not execute correctly.

Certain TMS340 boards provide external connection to the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ TMS340 processor pins. In such cases, interrupt service routines can be written for them using the techniques outlined here. However, such techniques are clearly not portable across all TMS340 processor boards.

4.9.1 Clock Example of Using Interrupts

This example displays a real-time analog clock on the TMS340 screen, which is updated by the use of the timer interrupt function installed in the display interrupt.

The timer functions are trivial, as can be seen in the `timer.asm` file; they simply increase a count. There is an additional function `get_time`, which returns the value of the count to the host. Notice that the TIGAEXT section is included with the timer. Because this is an assembly language program, there is no need to keep TIGAEXT separate. There is an additional TIGAISR section, that is similar to TIGAEXT but holds an interrupt level in addition.

The installation of the interrupt service routine (ISR) is exactly the same as for a regular extension, except that directly after the call to **install_rlm** is a call to **get_isr_priorities** (see main program of `clock.c`). These calls return the priority value for each of the interrupt service routines installed. Note that this means that an array big enough to hold all these priorities must be declared prior to invoking **get_isr_priorities** to hold the values that will be returned. In this example only one ISR is installed, so a single short variable will suffice. The priority is used in the call to **set_interrupt** to enable the interrupt. It is required, since TIGA allows any number of ISRs on a given interrupt level; thus the priority is the mechanism for identifying individual ISRs.

Following the call to **get_isr_priorities** is a call to **set_interrupt**. This takes two parameters to identify the ISR (an interrupt level and a priority) and two parameters, which may be set; an enable/disable flag and a display line (the latter is valid only for display interrupts and is ignored by interrupts at other levels).

After the interrupt is enabled, no direct reference is made to it. The function **get_time** is used to return the value of the count and thus determine the elapsed time. The **get_time** function in this example is not #defined; there is no absolute requirement to do this, but it is also clear that the code is less readable because of it.

4.9.2 Ball Example Using Interrupts

This example demonstrates many of the same features of the previous example with one major exception. The interrupt service routine performs some graphics operation (in the form of drawing a ball on the screen). Because the graphics operation uses implied operands in the B-file and I/O registers that cannot be guaranteed to be correct (since the interrupted routine could be using the OFFSET or DPTCH B-file registers as temporaries), the interrupt service routine has to set up these values. Because this involves over-writing their current values, they must first be saved somewhere. In this example they are saved in a global structure, by the routine setup_gsp_env. In an actual application, the registers could be pushed onto the stack using an MMTM instruction, if this function were recoded in TMS340 assembly. The graphics registers are then initialized using the values from the global structures such as CONFIG. After the ISR has completed, the restore_gsp_env function is called to restore the register values prior to returning to the interrupted function.

4.10 The TIGA Linking Loader

The TIGA linking loader TIGALNK is the mechanism by which extensibility is made possible. It is a full TMS340 linker that provides the capability of resolving references to TIGA graphics manager (GM) functions. TIGALNK is a full COFF loader which provides the capability of relocating object code 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.

The linking loader is invoked by several TIGA primitives for installing extensions into TIGA, and for performing various other tasks. Applications and device drivers written for TIGA should restrict themselves to the TIGA primitives and never invoke the linking loader directly, as the linker is subject to change in future revisions of TIGA, while the procedural interfaces will remain the same. A list of linking loader flags with their procedural equivalents is given in the list below:

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**

† TIGALNK can install an ALM. This is not done by the **install_alm** function, but by a function in the communication driver.

Below is a detailed description of the TIGALNK options. Note that 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 (this mode is used in the procedural interface). In verbose mode operation, the linker provides messages during every internal operation.

4.10.1 /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.

4.10.2 /cs – Create External Symbol Table

This option reads the symbolic information from the specified COFF file and places it in TIGA340.SYM, or if the optional command line argument was specified, in the *Symbol filename* supplied.

4.10.3 /ec – Error Check

This command line option can be used to check the integrity of an RLM prior to installing it. The TIGA graphics manager does not have to be active in order for this option to work, but if it is, the largest amount of available heap that can be used to load RLMs is also displayed.

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, 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 if the graphics manager is active, the largest available block of TMS340 heap is also displayed.

If the module contains any unresolved references that would not be resolved at load time, these are printed out. This allows the user to resolved symbol references before actually attempting to download and install the file.

Note:
Only symbols contained in the TIGA external symbol file (TIGA340.SYM) 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 primitives. If the module being checked contains references to other modules, such as the TIGA extended primitives, then these must be loaded prior to performing the check.

Example:

```
TIGALNK /LX - load and execute TIGAGM.OUT
TIGALNK /LR extprims - load TIGA extended primitives
                                                    (EXTPRIMS.RLM)
TIGALNK /EC user - check integrity of user.rlm
```

4.10.4 /fs – Flush External Symbol Table

This option flushes all but the symbols related to TIGAGM.OUT from the external symbol table, TIGA340.SYM. As the symbols for each installed module are deleted, a call to the TIGA graphics manager is also made to delete the module from TMS340 memory.

4.10.5 /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 such that functions contained in the module can 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.

4.10.6 /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 TIGA340.SYM, the external symbol table, so that they can be referenced by modules loaded afterwards.

4.10.7 /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 file (TIGA340.SYM) is created, and the symbols contained in TIGAGM.OUT are written to it. Once complete, a call to the TIGA communication driver function handshake is performed to initialize communications between the host and the TMS340.

Appendix A

TIGA Data Structures

This appendix contains the data structures used in TIGA. They are defined in the include file `typedefs.h`.

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-6
A.5	FONTINFO Structure	A-7
A.6	MODEINFO Structure	A-11
A.7	MONITORINFO Structure	A-13
A.8	OFFSCREEN Structure	A-14
A.9	PAGE Structure	A-15
A.10	PALET Structure	A-16
A.11	PATTERN Structure	A-17

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`. This file also has all type definitions in uppercase. 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

This structure contains the configuration information. Part of this structure is the MODEINFO structure defined in Section A.6, which describes the board configuration. If alternate configurations are available, they can be set using **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 revision number, assigned by Texas Instruments.

`comm_buff_size` Size, in bytes, of the communications buffer; application needs to ensure that the data sent does not overflow this buffer, for commands that do not check the size of the downloaded data.

`sys_flags` Bits 0 — 7 indicate FPUs (Floating Point Units) are present to be compatible with the TMS34020 coprocessor ID codes. Bits 8 —15 are reserved.

`device_rev` This function invokes the TMS340's REV instruction and returns its result here.

`num_modes` Number of extended modes, for boards that allow the switching between different display setups.

`current_mode` Mode number corresponding to the current operating mode.

`program_mem_start` Start address of program memory.

`program_mem_end` End address of program memory.

`display_mem_start` Start address of display memory.

`display_mem_end`	End address of display memory.
`stack_size`	Default stack size; can be modified using **gsp_minit**.
`share_mem_size`	Size (in bytes) of shared memory that is available for the application to use.
`share_host_addr`	If `share_size` is nonzero, it is the start address in host memory of the shared memory; otherwise it is undefined.
`share_gsp_addr`	If `share_size` is nonzero, it is the start address in TMS340 memory of the shared memory; otherwise, it is undefined.

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;
   PTR data;
}CURSOR;
```

This structure consists of the following fields:

`hot_x` — Offset x-coordinate added to the top left corner of the cursor shape to define the pixel specified by the **set_curs_xy.**

`hot_y` — Offset y-coordinate added to the top left corner of the cursor shape to define the pixel specified by the **set_curs_xy.**

`width` — Width of the cursor shape in pixels.

`height` — Height of the cursor shape in pixels.

`pitch` — Linear difference in the addresses of successive rows of the cursor shape (in bits).

`color` — Foreground color index with which the cursor is drawn.

`mask_rop` — Pixel processing operation used when applying the mask data to the background. This is normally specified as AND.

`shape_rop` — Pixel processing operation used when drawing the shape of the cursor onto the screen. This is normally specified as OR or XOR.

`data` — Pointer to TMS340 memory containing two contiguous arrays of `width` by `height`.The first array is the mask data with 0s where the cursor is located and 1s elsewhere. The second array is the shape data, which has 1s where the cursor is located and 0s elsewhere.

A.4 ENVIRONMENT Structure

The ENVIRONMENT structure contains the TIGA drawing environment global variables.

```
typedef struct
{
  ulong xyorigin;
  ulong pensize;
  PTR patnaddr;
  PTR srcbm;
  PTR dstbm;
  unsigned long stylemask;
}ENVIRONMENT;
```

The ENVIRONMENT structure consists of the following fields:

`xyorigin`	Current drawing origin in y::x format set by **set_draw_origin**
`pensize`	Current pen size arranged in y::x format, set by **set_pensize**
`patnaddr`	TMS340 address of current pattern, set by **set_patn**
`srcbm`	TMS340 address of current source bitmap structure, set by **set_srcbm**
`dstbm`	TMS340 address of current destination bitmap structure, set by **set_dstbm**
`stylemask`	Current line style mask used by **styled_line** function

A.5 FONTINFO Structure

The text rendering capabilities included as part of the TIGA extended primitives are very rich, providing the application writer with the ability to display everything from simple, fixed cell type text, such as that used by dumb terminals and EGA, VGA graphics adapters, to the desktop publishing type (wysiwyg) text, where the height and width of characters, along with the style and size can vary.

The fonts used for text rendering are a collection of characters having a unique combination of height, width, style, and other attributes. The format of these fonts are unique to TIGA and are described in the following paragraphs:

The characters within a font have an associated two-dimensional bitmap that defines the shape of the character. When the text is rendered, On bits (1s) within a character bitmap are expanded to pixels in the active foreground color, as set by the **set_fcolor** function. Off bits (0s) are expanded pixels in the background color. The format of a font is defined by the following data structure:

```
typedef struct
{
   short magic;       /* TIGA Identifier                    */
   long  length;      /* length of font in bytes            */
   char  facename[32];
   short first;       /* ASCII code of first character      */
   short last;        /* ASCII code of last character       */
   short maxwide;     /* maximum character width            */
   short maxkern;     /* maximum character kerning amount   */
   short charwide;    /* char. width (0 if proportional)    */
   short avgwide;     /* average width of characters        */
   short charhigh;    /* character height                   */
   short ascent;      /* ascent (how far above base line)   */
   short descent;     /* descent (how far below base line)  */
   short leading;     /* leading (row bott.to next row top) */
   PTR   fontptr;     /* address of font in GSP memory      */
   short id;          /* id of font (set at install time)   */
}FONTINFO;
```

The following is a description of the FONTINFO structure parameters. Parameters 8, and 10 through 13 are shown in Figure A–1.

1) `magic`

   This field is an identifier for the data structure. It consists of three parts:
   bits 00 — 01: data structure sub type
   bits 02 — 07: data structure type
   bits 08 —15: TIGA identifier

   For the bitmap fonts described here, the `magic` identifier is filled in as follows:

bits 00 — 01: 0 (FONT subtype = bitmap)
bits 02 — 07: 1 (FONT)
bits 08 — 15: 0x80 (Indicates TIGA font format1.x)

For this particular font data structure, the magic number value is 0x8040. In the future, TIGA may support outline or stroke fonts, in which case the font subtype would change.

2) `length`

The length of the entire font in bytes. This is useful when allocating memory for a font and for reading it from disk.

3) `facename`

A NULL terminated string of ASCII characters up to 32 long containing the name of the font. Example: TI Roman, TI Helvetica, etc.

4) `first`

ASCII code of the first character defined in the font. For example, if first was 0x20, the ASCII code for a space character, then that is the lowest ASCII code for which a bitmap is defined in the font.

5) `last`

ASCII code of the last character defined in the font.

6) `maxwide`

The width of the widest character defined in the font.

7) `maxkern`

The maximum kern for any character within the font, expressed as a positive value. For example, if kerning was 3, then the maximum any character will back up to overlap the previous drawn character is 3.

8) `charwide`

The character width is the image width of the character, plus the space separating this character from the next. If the character width is zero, then the width of characters within the font varies. In that case an entry in the offset/width table specifies the width for each character.

9) `avgwide`

The average width of characters within the font. This is the cell width of all defined characters within the font (not considering any kerning or extra intercharacter spacing) divided by the number of characters defined. It is useful when selecting a font for a best fit at varying target resolutions.

10) `charhigh`

The character height is the sum of the ascent and decent. It is constant throughout any particular font but may vary between fonts.

11) `ascent`

The ascent is the number of vertical pixels from the base line to the top of the font cell.

12) `descent`

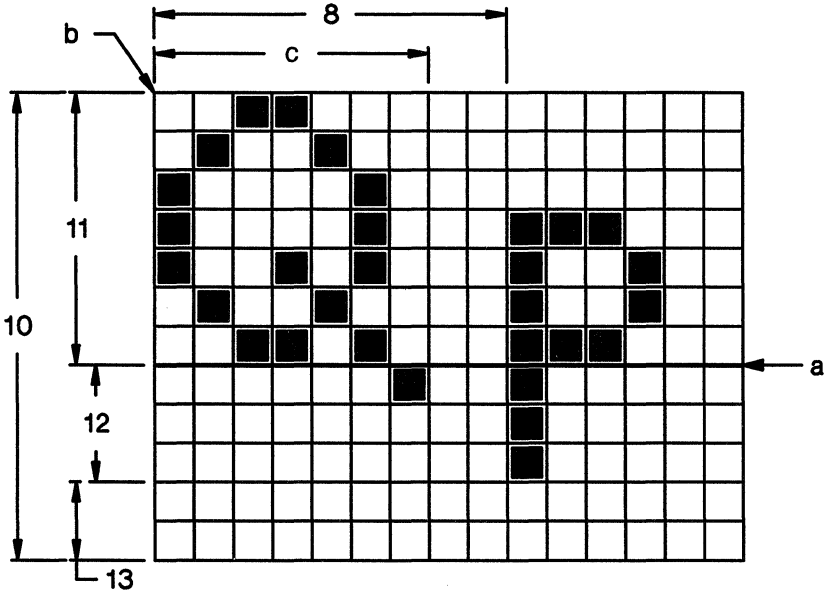
The descent is the number of vertical pixels from the base line to the bottom of the cell.

13) `leading`

This term comes from the fact that typesetters often used strips of lead to adjust spacing between rows of characters when building a plate to be printed with a printing press. For bitmap fonts, this value is the number of pixels recommended by the font designer that should be skipped between rows of characters, that is, if the leading is 3, then 3 pixels should be skipped between the descent line of a row of characters, and the ascent line of the row of characters drawn directly beneath.

Some of the fields in the font structure are illustrated in Figure A–1. The numbers refer to the numbered sections in the parameter description.

Figure A–1. Bitmap Font Format



In addition, Figure A–1 illustrates the following font characteristics:

a) Base Line

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

b) Character Origin

The character origin is that part of a character corresponding to a specified drawing location. This origin may vary, depending on the text alignment attribute used to draw text. The default text alignment is relative to the top left corner of the character cell. Alignment can also be set relative to the leftmost point on the character baseline by performing a call to **set_textattr**. Baseline origin is useful when a string of characters consists of different size or style fonts, in which case the baseline should remain constant throughout the text.

c) Image Width

The image width is the number of bits of the portion of the character pattern bitmap containing the actual character image. This width does not include any blank space to the left or right of the character when it is displayed and can vary within a font and between fonts.

A.6 MODEINFO Structure

This structure contains all the configuration information that can vary on a specific board. It is part of the configuration structure returned by **get_config** (which returns only the MODEINFO for the currently installed mode). The total possible modes can be inquired using **get_modeinfo**.

```
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;
}MODEINFO;
```

The MODEINFO structure consists of the following fields:

`disp_pitch` — Display pitch: linear difference between two scan lines in bits.

`disp_vres` — Vertical resolution in scan lines.

`disp_hres` — Horizontal resolution in pixels.

`screen_wide` — Contains the width of the monitor in centimeters. For systems where these dimensions are unknown, set to 1.

`screen_high` — Contains the height of the monitor in centimeters. For systems where these dimensions are unknown, set to 1.

`disp_psize` — Pixel size.

`pixel_mask` — Contains a mask of the bits used in a pixel. It will normally contain the value of *2 to the power* `disp_psize` *minus 1*, indicating that every bit of pixel data is pertinent. On some boards, the frame buffer may be arranged by 8 (`disp_psize = 8`) but with only 6 bits implemented. In that case, pixel mask would contain the value 63 (hexadecimal 3F).

`palet_gun_depth` — Number of bits per gun in palette.

`palet_size`	Number of entries in the palette.
`palet_inset`	For most systems, this field is set to 0. For TMS34070-based boards, which store the palette in the frame buffer, this is the offset from the start of the scan line to the first pixel data.
`num_pages`	Number of display pages in multi-buffered systems.
`num_offscrn_areas`	This is the number of offscreen memory blocks available. If nonzero, then it is used to allocate space for the offscreen array, which can be obtained from the TMS340 via a call to the **get_offscreen_memory** function.
`wksp_addr`	Starting linear address in memory of offscreen workspace area.
`wksp_pitch`	Pitch of offscreen workspace area. If `wksp_pitch = 0`, then no offscreen workspace is currently allocated.

A.7 MONITORINFO Structure

This structure is not of general interest to an application writer. It is used by the OEM porting TIGA to its board to specify the values of the video timing parameters for a particular mode. Note that this structure is board-specific. An OEM is free to add to this structure its own OEM-specific video timing information. This structure will invariably change for a TMS34020 version of TIGA.

```
typedef struct
{
  ushort hesync;
  ushort heblnk;
  ushort hsblnk;
  ushort htotal;
  ushort vesync;
  ushort veblnk;
  ushort vsblnk;
  ushort vtotal;
  ushort dpyctl;
  ushort screen_delay;
  ushort flags;
}MONITORINFO;
```

The MONITORINFO structure consists of the following fields:

`hesync`	value loaded into the HESYNC I/O register
`heblnk`	value loaded into the HEBLNK I/O register
`hsblnk`	value loaded into the HSBLNK I/O register
`htotal`	value loaded into the HTOTAL I/O register
`vesync`	value loaded into the VESYNC I/O register
`veblnk`	value loaded into the VEBLNK I/O register
`vsblnk`	value loaded into the VSBLNK I/O register
`vtotal`	value loaded into the VTOTAL I/O register
`dpyctl`	value loaded into the DPYCTL I/O register
`screen_delay`	Number of frames that the screen is blank when loading the video registers. This allows a monitor time to synchronize to the new timing before the screen is unblanked.
`flags`	Monitor desription flags. Current flags defined are 0 = color monitor, 1 = monochrome monitor.

A.8 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 an offscreen work area.

`xext` x extension of the offscreen area in the current screen pixel size.

`yext` y extension of the offscreen area in the current screen pixel size.

A.9 PAGE Structure

This structure is not of general interest to an application writer. It is used by the OEM porting TIGA to his board to specify the start addresses of the display page (the value loaded into the display start I/O Register) and drawing page (the value loaded into the offset B-file register). This structure is used to support multiple display pages used by the **page_flip** function. Note that this structure is board-specific and may change in future versions of TIGA.

```
typedef struct
{
  PTR BaseAddr
  ushort DpyStart
  short DummyPad;
}PAGE;
```

The PAGE structure consists of the following fields:

`BaseAddr` Base address of start of drawing page; this value is loaded into the OFFSET B-file register.

`DpyStart` Base address of start of display page; this value is loaded into the Display Start I/O register.

`DummyPad` 16 bits to pad structure to power of 2 size.

A.10 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 gun

`g` Value of the green gun

`b` Value of the blue gun

`i` Value of the intensity

A.11 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.

`height` Height of pattern in bits.

`depth` Depth (bits/pixel) of pattern.

`data` Pointer to pattern data in TMS340 memory.

Appendix B

Graphics Output Primitives

This appendix describes some of the assumptions made in the design of the TIGA graphics output primitives which are part of the extended primitives. It also describes the conventions adopted regarding the drawing, mapping, and filling with pixels to represent mathematical functions on a video screen. This appendix includes the following sections:

Section		Page
B.1	Categories of Graphics Output Primitives	B-2
B.2	Fill Patterns	B-4
B.3	Mapping Pixels to XY Coordinates	B-5
B.4	Area Filling Conventions	B-6
B.5	Vector Drawing Conventions	B-7
B.6	Drawing Pen	B-8
B.7	Color Selection	B-9

B.1 Categories of Graphics Output Primitives

The graphics functions draw several shapes in a variety of styles. Table B–1 describes the figure types and drawing styles. Table B–2 shows the shapes that can be drawn in a particular style. The column headers list the available styles and the row headers list the available shapes; a check mark indicates that a shape can be drawn with a particular style.

Table B–1. List of Function Types and Drawing Styles

Function Types	
Function Name	**Description**
line	A straight line.
oval	Ellipse in standard position (major and minor axes parallel to coordinate axes).
ovalarc	An arc of an ellipse in standard position, specified in terms of beginning and ending angles.
point	A single pixel or pen image drawn at the indicated XY coordinate pair.
polygon	A filled region defined by a collection of straight edges. Both convex polygons and arbitrarily complex polygons are supported.
polyline	A collection of straight lines. Figures made up of many lines can be drawn more efficiently by using the polyline commands than by repeated calls to the line functions.
piearc	Pie arc or wedge. Similar to ovalarc, but with addition of sides drawn from center of ellipse to arc endpoints to produce a closed figure.
rect	Rectangle with vertical and horizontal sides.
seed	Fill connected set of pixels beginning at specified seed point.

Drawing Styles	
Function Name	**Description**
draw	Draws figure outline one pixel wide using background color.
fill	Draws figure interior filled in solid background color.
frame	Draws frame in solid background color. Horizontal and vertical thicknesses of frame border are both specified.
patnframe	Draws frame, using pattern in the foreground and background colors. Horizontal and vertical thicknesses of frame border are both specified. The pattern is programmable.
patnpen	Draws figure outline using pen and pattern in the foreground and background colors. Pen size and pattern are programmable.
pen	Draws figure outline using pen in solid background color. Pen is rectangular with programmable height and width.
patnfill	Draws figure interior filled with pattern in the foreground and background colors. The pattern is programmable.

Table B–2. Checklist of Available Figure Types and Drawing Styles

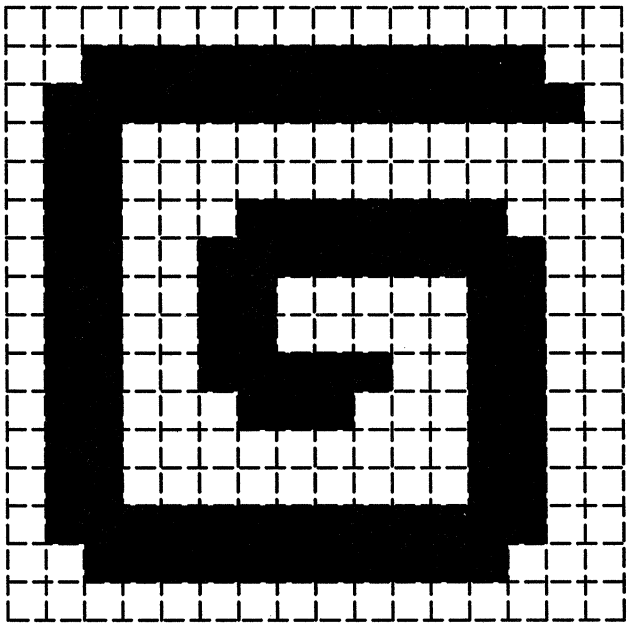
	Drawing Styles						
Figure Type	**draw**	**pen**	**patnpen**	**fill**	**patnfill**	**frame**	**patnframe**
line	√	√	√	N/A	N/A	N/A	N/A
oval	√			√	√	√	√
ovalarc	√	√	√				
piearc	√	√	√	√	√		
point	√	√	√	N/A	N/A	N/A	N/A
polygon	N/A	N/A	N/A	√	√	N/A	N/A
polyline	√	√	√	N/A	N/A	N/A	N/A
rectangle	√			√	√	√	√
seed	N/A	N/A	N/A	√	√	N/A	N/A

B.2 Fill Patterns

Graphics functions that include **patn** as part of their names draw with a pattern instead of a solid color. The pattern is currently limited to a 16 x 16 bitmap and is represented in memory as an array of 256 contiguous bits. The bits in a pattern are listed in left-to-right order within a row, and rows are listed in top-to-bottom order.

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

Figure B–1. A 16 x 16 Pattern



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

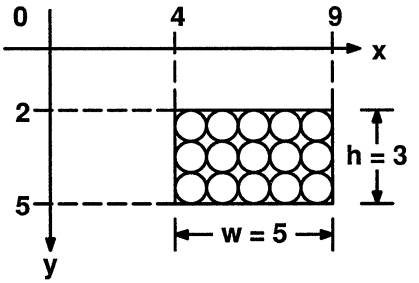
B.3 Mapping Pixels to XY Coordinates

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

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

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

Figure B–2. Rectangle Fill

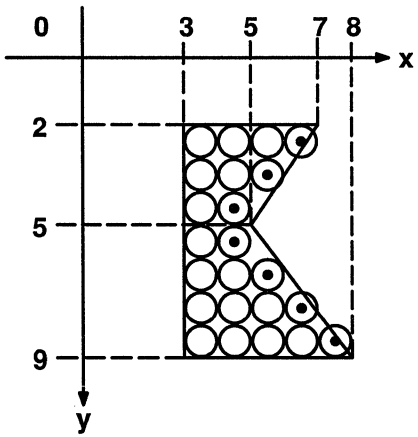


B.4 Area Filling Conventions

Figure B–3 shows a filled polygon, in which a **fill_polygon** function defines the fill area indicated by the straight edges in the figure. The rule for determining whether a pixel is selected as part of the fill area is as follows: if the center of the pixel falls within the mathematical boundary of the area, it is turned on to indicate that is part of the fill area. (If a pixel's center falls precisely on the boundary between two areas, by convention the pixel is considered to be part of the area immediately below and to the right of the pixel). Pixels whose centers lie outside the boundary are not considered part of the fill region. The same principles are applied to the filling of other shapes (ellipses and thick lines drawn with a rectangular drawing pen, for example).

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

Figure B–3. Polygon Fill



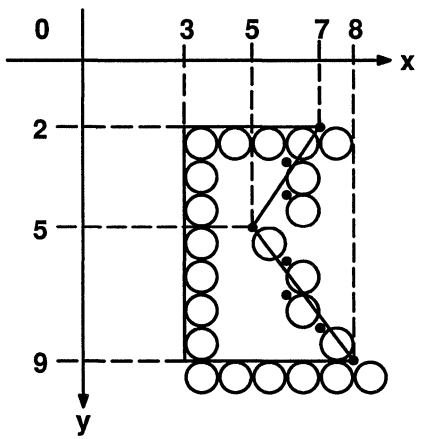
B.5 Vector Drawing Conventions

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

The vector drawing conventions associate the point identified by the integer coordinate pair (X,Y) with the pixel located to its lower right; that is, the pixel whose center is located at coordinates (X+1/2,Y+1/2). For example, the **draw_point** `(7,10)` command turns on the pixel at (7.5,10.5). As a second example, the polygon from Figure B–3 is shown again in Figure B–4 but is outlined rather than filled. (The **draw_polyline** function is used.) The points selected to represent the right side of the polygon are indicated as small black dots. The pixel to the lower right of each point is turned on to represent the edge of the polygon.

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

Figure B–4. Polygon Outline



B.6 Drawing Pen

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

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

The area under the drawing pen is filled according to the area-filling conventions described previously. When the width and height of the drawing pen both equal 1, a line or arc drawn by the pen is similar in appearance to that drawn by a function following the vector drawing conventions. However, the pen functions conform to the area-filling conventions, so a pen function can track more faithfully the perimeter of a filled area than a corresponding draw function.

For example, consider an ellipse defined by some width `w`, height `h`, and coordinates (`x`, `y`). If a **draw_oval**(`w`, `h`, `x`, `y`) function call outlines a filled ellipse drawn by a **fill_oval**(`w`, `h`, `x`, `y`) function, the **draw_oval** function may not in all instances select the same perimeter pixels as the filled ellipse. This can leave gaps between the filled area and the outline. In contrast, a **pen_oval**(`w`, `h`, `x`, `y`) function call follows the filled ellipse precisely, remaining flush to the ellipse at all points along the perimeter.

B.7 Color Selection

The TIGA standard enables applications to be ported from one TMS340 board to another. One of the most difficult parts of the porting process is ensuring that the colors chosen for the application are distinguishable (if not identical) when the application is run on another board. Palettes vary from board to board, sometimes considerably. This section describes the TIGA methodology concerning color selection.

The configuration structure returned in **get_config**, contains the `disp_psize` element, which the application can use to determine the number of colors that can be on the screen at any given time. The application must interrogate this value to determine if this number is sufficient, and double-up if necessary, painting different geometries with the same color.

Selecting the colors is done via the **set_palet**, or **set_palet_entry** functions for a RAM-based palette. Because the palette may be ROM-based (making it impossible to set the palette entries), the **function_implemented** function should be used on the **set_palet** functions prior to invoking them. If they are not implemented, the palette can be assumed to be ROM-based and a technique described later can be used to select colors. For RAM-based palets, each entry can be set via a call to **set_palet**, which takes as its parameters an 8-bit value of red, green, blue, and intensity.

For color monitors, the intensity field is ignored and the R-G-B values are used to load the palette entry. Because the palette may only use 4 or 6 bits, it takes the most significant portion of the 8-bit palette entry to set the color. The number of bits for each color gun is stored in the `palet_gun_depth` field of the CONFIG structure. Alternatively, the **get_palet** function will return the physical colors stored in each entry (as opposed to the logical colors requested by the **set_palet** function). Thus, colors can be chosen and specified directly with this approach. For monochrome monitors, only the intensity field is used, to specify the level of the grey scale for each entry. Again, the most significant bits are used when the palette entry size is less than 8 bits. Thus, for RAM-based palettes, the application should specify both a color and monochrome values for each color index used.

For ROM-based palettes, the **get_nearest_color** function can be used to inquire which color index to use. This function operates in reverse to the previous case where instead of setting an 8-bit red, green, blue color index with a desired value, the nearest one to the desired value is returned to the calling application. Again, an independent grey-scale value for each color index must also be requested for ROM-based monochrome monitors.

In summary, the application must test **function_implemented** on **set_palet** to determine whether the palette is ROM- or RAM-based. If it is

RAM-based, the application can select its palette directly and must do so in both R-G-B and intensity values for monochrome monitors. If the palette is ROM-based, the application must use **get_nearest_color** on each of its desired palette entries to set up the color indices, again specifying both color and monochrome values.

Finally, there is also a short cut: if the **init_palet** function is implemented (which is the case in RAM-based palettes with 4 bits-per-pixel or more), the palette values after initialization are those stated in the **init_palet** function. The palette values are declared symbolically in an insert file `tiga.h`, and if these values are acceptable, they can be used directly by an application.

Appendix C

TIGA Reserved Symbols

Section		Page
C.1	Reserved Functions	C-2
C.2	TIGA Core Primitive Symbols	C-3
C.3	TIGA Extended Primitive Symbols	C-5

C.1 Reserved Functions

TIGA currently reserves the following functions for internal use. Do not chose function names that conflict with these. Avoid calling functions from an application program, since 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

C.2 TIGA Core Primitive Symbols

TIGA currently uses the following symbols in its core primitives and for the TMS340 C environment. To guarantee succesful operation, do not use downloadable extensions that conflict with any of these symbols.

If the extension is also to work with the extended primitives, then Section C.3 should also be considered when selecting symbol names.

.bss
.data
.text
IsrCStk
IsrEntryTable
IsrSrv
_CoreFunc
_CursorISR
_DEFAULT_PALET
_DefaultCursor
_DiTable
_Module
_NextDiEntry
_PageFlipISR
_TrapVector
_WaitScanISR
_abort
_add_interrupt
_add_module
_atexit
_bottom_of_stack
_c_int00
_check_dpyint
_clear_frame_buffer
_clear_page
_clear_screen
_comm_info
_config
_cpacket
_cpw
_default_setup
_del_all_modules
_del_interrupt
_del_module
_delay
_dm_clear_frame_buffer
_dm_clear_page
_dm_clear_screen
_dm_cpw
_dm_get_nearest_color
_dm_get_palet
_dm_gsp2gsp
_dm_init_palet
_dm_lmo
_dm_peek_breg
_dm_poke_breg
_dm_rmo
_dm_set_bcolor
_dm_set_clip_rect
_dm_set_colors
_dm_set_curs_shape
_dm_set_curs_state
_dm_set_fcolor
_dm_set_palet_entry
_dm_set_pmask
_dm_set_ppop
_dm_set_windowing
_dm_set_wksp
_end_of_dram
_envcurs
_envtext
_env
_exit
_flush_extended
_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_transp
_get_vector
_get_windowing
_get_wksp
_getrev
_gsp2gsp
_gsp_calloc
_gsp_free
_gsp_malloc
_gsp_maxheap
_gsp_minit
_gsp_realloc
_illop
_init_cursor
_init_interrupts
_init_palet
_init_text
_init_trap_vectors
_init_video_regs
_lmo
_main
_modeinfo
_monitorinfo
_offscreen
_page_busy
_page_flip
_page
_palet
_palloc
_pattern
_peek_breg
_poke_breg
_put_vector
_release_buffer
_rmo
_set_bcolor
_set_clip_rect
_set_colors
_set_config
_set_curs_shape
_set_curs_state
_set_fcolor
_set_interrupt
_set_palet
_set_palet_entry
_set_pmask
_set_ppop
_set_windowing
_set_wksp
_setup
_stack_size
_start_of_dram
_strcpy
_sys_free
_sys_memory
_sysfont
_text_out
_transp_off
_transp_on
_video_enable
_wait_scan
cinit
edata
end
etext

C.3 TIGA Extended Primitive Symbols

TIGA currently uses the following symbols in its extended primitives. Downloadable extensions that work with the extended primitives should not use names that conflict with any of these symbols; this guarantee successful operation.

_arc_draw
_arc_fill
_arc_pen
_arc_quadrant
_arc_quad
_arc_slice
_bitblt
_c_int00
_config
_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_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_seed_fill
_dm_seed_patnfill
_dm_set_draw_origin
_dm_set_patn
_dm_set_pensize
_dm_zoom_rect
_draw_eliparc
_draw_line
_draw_ovalarc
_draw_oval
_draw_piearc
_draw_point
_draw_polyline
_draw_rect
_env
_envtext
_fill_convex
_fill_eliparc
_fill_oval
_fill_piearc
_fill_polygon
_fill_rect
_fill_shape
_frame_oval
_frame_rect
_get_env
_get_pixel
_get_textattr
_gsp_malloc
_gsp_realloc
_install_font
_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
_pattern
_pen_eliparc
_pen_line
_pen_ovalarc
_pen_piearc
_pen_point
_pen_polyline
_seed_fill
_seed_patnfill
_select_font
_set_draw_origin
_set_dstbm
_set_patn
_set_pensize
_set_srcbm
_set_textattr
_sin_tbl
_styled_line
_swap_bm
_sysfont
_text_width
_zoom_rect

Appendix D

Porting TIGA

The TIGA-340 Software Porting Kit (SPK) contains all source required to port TIGA to any TMS340-based graphics board. The SPK is originally shipped with a TI Software Development Board (SDB) compatible version of TIGA. This version is used as an example, and a port of TIGA to a different TMS340 board will involve modifications to the SDB port.

Before beginning the porting process, make sure you have the following software tools installed on your system:

- Microsoft macro assembler, version 5.0 or above
- Microsoft MAKE utility
- Microsoft linker
- TMS340 assembler/linker, version 3.0 or above
- TMS340 C compiler, version 3.0 or above

Porting TIGA consists basically of two tasks.The first is to modify the host-side TIGA code (the TIGA communication driver or CD for short). The second is to modify the TMS340-side code (the TIGA graphics manager or GM for short). Each of these tasks have well-defined, step-by-step procedures that make porting TIGA to a different TMS340-based board a relatively simple operation. Because the GM rebuilding process relies on a functional CD, you must first port the CD before porting the GM.

Porting TIGA entails some knowledge of TIGA's architecture; therefore, it may be helpful to refer to Section 1.3 for an overview of the components of TIGA.

Section **Page**

D.1 Porting the Communication Driver (CD) D-2
D.2 Porting the Graphics Manager D-14
D.3 Verifying Correct Operation D-21
D.4 Debugging Your Port D-22

D.1 Porting the Communication Driver

The TIGA communication driver (CD) is an MS-DOS Terminate-and-Stay-Resident (TSR) program that enables host communications with the TMS340-based board. When the TIGA-340 SPK is installed, it places all CD-specific files in the directory `\tiga\cd`. The files within this directory that may need modifications, along with a description of each, are listed below:

`oemdata.asm`	Contains information defining each mode of operation supported by the TMS340-based board
`oeminit.asm`	Contains board-specific initialization and inquiry functions
`setvideo.asm`	Contains routines to set/get video mode information
`sdbdefs.inc`	Contains hardware-specific equates

Porting the TIGA CD involves modifications to each of the files above. The following four sections describe in detail these modifications. Note that all references to file names are assumed to be files within the `\tiga\cd` directory unless otherwise noted.

D.1.1 Modifying the `sdbdefs.inc` File

The `sdbdefs.inc` file contains general information describing the hardware aspects of the target TMS340-based board TIGA is being ported to. Before making any modifications to this file, first copy it to a file that will be used to describe your target board. Make sure to copy it within the `\tiga\cd` directory and that its extension is `.inc`. For example purposes, assume the copied filename is `newdefs.inc` and use this filename throughout the communication driver porting guide.

Next, edit `newdefs.inc`; note that the file contains a number of equates defining constants that are used when the TIGA CD is rebuilt. Modify **only** the constants described below. Modifications to any constant not listed below will result in a non-functioning communication driver:

OEMMSG	This message is displayed when the CD is installed. It describes the board on which TIGA is running. Be sure to enclose the description within double quotes.
SDB	If your target board is not the TI Software Development Board (SDB), then set this constant to false (0).
MEMORY	This constant defines how the TMS340's host registers (HSTDATA, HSTCTRL, HSTADRH, and HSTADRL) are accessed from the host. If your target board's host registers are memory-mapped, set this constant to 1. Otherwise, they are I/O-mapped, in which case the constant should be set to 0.

SYSTEM If you want your TIGA CD to be compatible with Intel's 8086 and later microprocessors, set this constant to false (0). If set to TRUE (1), 80x86 instructions will be used in the CD. This results in faster execution but places a restriction on the host processor type.

Next, modify the host portion defining the host port locations. These equates define the host addresses (either memory or I/O) used to communicate with the TMS340. Note that these addresses are hard-coded. If your board can be configured to different host address memory locations, select one set of addresses for the initial port (refer to section D.1.19 for more information on boards with multi-host port addressing).

Note that the HSTSEG address is valid for memory-mapped host register boards only. Modify these constants to match your board's host register addressing.

The following values for the host register addresses are taken from the SDB (memory-mapped) port.

```
HSTSEG          Equ     0C000h ; Host port seg.(mem-mapped only)
HSTADRL         Equ     7E00h  ; Host address low
HSTADRH         Equ     7F00h  ; Host address high
HSTDATA         Equ     7000h  ; Host data
HSTCTRL         Equ     7D00h  ; Host control
```

The next two equates are used as timeout values in the CD. For most ports, these values should suffice. However, if the call to **gm_is_alive** fails consistently, you may have to increase the GM_TIMEOUT value (see Section D.1.18 for more information).

The next set of constants defines various monitors supported by the different operating modes of your board. Each monitor constant defines a bit flag in a 16-bit word. Therefore, TIGA supports up to 16 monitors per mode. The following monitor definitions were taken from the SDB port:

```
NEC     Equ     1       ; SDB port supports two monitors, the
SONY    Equ     2       ; NEC and the SONY Multisyncs
```

Here, two monitors are supported, the NEC Multisync and Sony Multisync. Note that the first monitor constant has a value of 01h, and the next has a value of 02h. If additional monitors were supported, their values would be 04h, 08h, 10h, etc., up to 80h.

Next, modify the number of palette entries supported on your board in its default power-up mode. The SDB uses the TMS34070 4-bit color palette and, therefore, has 16 palette entries.

Finally, one miscellaneous equate may need modification for your port.

STKSIZE – The host-side communication driver stack size may be modified to suit your needs. The SDB port allocates 0400h bytes of stack.

D.1.2 Modifying the `oemdata.asm` File

The `oemdata.asm` file contains descriptions of all the operating modes supported by your board. An operating mode is defined as a given resolution with appropriate monitor timing values that support this resolution. The operating modes of the SDB port are used as an example. Use this example as a guide to define the operating modes of your board.

The label Setup_Table defines the start of the operating mode specific data. The SDB port supports four operating modes: the first is a 640 x 480 x 4 resolution mode (1 display page) and the second is a 448 x 480 x 4 (2 display pages) mode. These two modes are duplicated for the NEC and Sony Multisync monitors. Each mode supported by your board must be defined by a label following the Setup_Entry label. These mode labels are used to actually define the mode-specific data.

```
;
; Define number of modes and mode/setup tables
;
Setup_Table   Equ  This Word      ; Start of setup table
Setup_Entry   Mode_640x480x4N     ; SDB mode 0 for NEC
Setup_Entry   Mode_448x480x4N     ; SDB mode 1 for NEC
Setup_Entry   Mode_640x480x4S     ; SDB mode 0 for Sony.
Setup_Entry   Mode_448x480x4S     ; SDB mode 1 for Sony.
```

D.1.3 Defining the Mode-Specific Information

Each board operating mode has its own set of data describing the following items:

- Monitor identification flags
- Mode-specific information
- Monitor timing information
- Display page information
- Offscreen memory information

For example, mode 0 of the SDB port is defined as follows:

```
;*******************************************************************************
;  SDB Mode 0: 640x480x4, 1 page for NEC Monitor
;*******************************************************************************
;
;  SETUP Structure
;
Mode_640x480x4N  Equ     This Byte              ; Mode 0 setup structure
   Setup_Struc     <NEC>
   Mode_Info       1000h,480,640,27,20,4,0Fh,4,PALET_ENTRIES,100h,1,2,0B00h,1000h
   Monitor_Info       01Bh,01Ch,0CCh,0CDh,001h,018h,01F8h,01FAh,0F010h,30,0
   Page_Info       00000100h,0FFFCh
   Off_screen      00000B00h,640/4,480
   Off_Screen      00000D80h,160,480
   Off_Screen      001E0000h,1024,32
   End_Setup
```

Each operating mode of your board has a similar block of information defined, one block for eack mode defined in the Setup_Table. Use the following instructions to modify each mode information block for your board.

D.1.4 Defining the Mode Label and Setup_Struc Structure

The mode setup structure starts with a label identifying the mode. This label is the same as the one added in the Setup_Table entries earlier.

Next, initialize the Setup_Struc macro with the valid monitors supported by this mode. These monitor constants were defined earlier in the `newdefs.inc` file.

D.1.5 Defining the Mode_Struc Structure

Next, modify the mode-specific information. The Mode_Info structure contains parameters describing this operating mode. The structure is defined in the file `struct.inc` as follows:

```
Mode_Struc             Struc; Mode information structure
Mode_Disp_Pitch        Dd? ; Display pitch (bits)
Mode_Disp_Vres         Dw? ; Vertical resolution (pixels)
Mode_Disp_Hres         Dw? ; Horizontal resolution (pixels)
Mode_Screen_Wide       Dw? ; Screen width (centimeters)
Mode_Screen_High       Dw? ; Screen height (centimeters)
Mode_Disp_Psize        Dw? ; Display pixel size
Mode_Pixel_Mask        Dd? ; Pixel mask
Mode_Palet_Gun_Depth   Dw? ; Palette gun depth (bits)
Mode_Palet_Size        Dd? ; Palette size
Mode_Palet_Inset       Dw? ; (For TMS34070 palette only)
Mode_Num_Pages         Dw? ; Number of screen pages
Mode_Num_Offscrn       Dw? ; Number of off-screen areas
Mode_Wksp_Address      Dd? ; Temporary Workspace Address
Mode_Wksp_Pitch        Dd? ; Temporary Workspace Pitch
Mode_Struc             Ends ; End of Mode_Struc structure
```

The Mode_Num_Pages field describes the total number of display pages supported for this mode. Multiple display pages are used in TIGA to support animation via the **page_flip** function. Section D.1.7 provides additional information.

The Mode_Num_Offscrn field describes the total number of x-y offscreen memory blocks available for use by an application. Section D.1.8 describes these memory blocks in greater detail.

The fields Mode_Wksp_Address and Mode_Wksp_Pitch describe the off-screen workspace. This workspace is a 1-bit-per-pixel bitmap with the same horizontal and vertical dimensions as the visible screen. It is used by the **fill_polygon** and **patnfill_polygon** functions as a working buffer. If enough offscreen memory is available to support this workspace, then this offscreen memory block should be the first Off_Screen structure defined (see Section D.1.8) and the Mode_Wksp_Address and Mode_Wksp_Pitch fields should be initialized to point to this block.

D.1.6 Defining the Monitor_Info Structure

Next, modify the Monitor_Info specific data for this mode. This structure is defined in the `struct.inc` header file as follows:

```
Monitor_Struc          Struc; Monitor information structure
Monitor_Hesync         Dw? ; End horizontal sync signal
Monitor_Heblnk         Dw? ; End horizontal blank signal
Monitor_Hsblnk         Dw? ; Start horizontal blank signal
Monitor_Htotal         Dw? ; Horizontal total (Characters)
Monitor_Vesync         Dw? ; End vertical sync signal
Monitor_Veblnk         Dw? ; End vertical blank signal
Monitor_Vsblnk         Dw? ; Start vertical blank signal
Monitor_Vtotal         Dw? ; Vertical total (Scanlines)
Monitor_Dpyctl         Dw? ; Display control register
Monitor_Screen_Delay   Dw? ; Screen delay (Frames)
Monitor_Flags          Dw? ; Monitor type flags (Bit0
                           ; :0=color, 1=mono) Bits 1-15
                           : reserved
Monitor_Struc          Ends ; End of Monitor_Struc structure
```

The structure element, Monitor_Screen_Delay, specifies the delay (in frames) that the screen will be blanked when the video registers are loaded. This allows the monitor time to synchronize to the new timing values.

The structure element Monitor_Flags specifies whether the monitor is color (when the LSB is a 0), or monochrome (when the LSB is 1). Bits 1 —15 are reserved. The palette routines use this flag to choose between the intensity level or the R, G, B values specified in the palette structure.

D.1.7 Defining the Page_Info Structure

The next structure, Page_Info, contains information defining the available display pages for this particular mode. Each mode must have at least 1 display page defined. For each display page, a corresponding Page_Info structure must be defined. The actual number of display pages is defined in the Mode_Num_Pages field of the Mode_Struc structure. The Page_Info structure is defined in the file `struct.inc` as follows:

```
Page_Struc          Struc       ; Page information structure
Page_Base_Addr      Dd      ?   ; Page base address
                                ; (linear bit address)
Page_DpyStart       Dw      ?   ; Page start offset
Page_Pad            Dw      ?   ; Page dummy pad bytes
Page_Struc          Ends        ; End of Page_Struc structure
```

Each page is defined by two elements:

1) The base address (loaded into the B-file register OFFSET) when this page is being written to

2) The display start (loaded into the DPYSTRT I/O register) when this page is being displayed

Using Mode 1 (448 x 480 x 4, 2 pages) of the SDB port as an example, two display pages are supported. These page definitions are as follows:

```
Page_Info     00000100h,0FFFCh ; Page 0
Page_Info     00000900h,0FFF4h ; Page 1
```

The TIGA core function, **page_flip**, enables the selection of the current display and drawing page. For example, **page_flip(0,1)** selects page 0 as the display page and page 1 as the drawing page. Therefore, B-file register OFFSET would be loaded with 0900h (the base address for page 1) and the DPYSTRT IO register with 0FFFCh (the display start for page 0).

Note:

Even though the Page_Info structure contains 16 bits of pad, this value should not be entered as part of the Page_Info information.

Be sure and note the maximum number of display pages defined in any operating mode of your board. This value is required when porting the TIGA graphics manager.

D.1.8 Defining the Off_Screen Structure

TIGA enables an application to use offscreen memory as a bitblt storage or temporary workspace though the **get_offscreen_memory** function. This function returns information describing the available offscreen memory areas that are defined in the OFF_SCREEN structure.

The OFF_SCREEN structure is defined in the file `struct.inc` as follows:

```
Screen_Struc      Struc  ; Screen information structure
Screen_Address    Dd ?   ; Start (linear) off-screen memory
Screen_X_Ext      Dw ?   ; X extent of memory (pixels)
Screen_Y_Ext      Dw ?   ; Y extent of memory (pixels)
Screen_Struc      Ends   ; End of Screen_Struc structure
```

The actual number of offscreen areas for a particular mode is defined in the Mode_Num_Offscrn field of the Mode_Struc structure. For each offscreen area, a corresponding Off_Screen structure is defined. If your board does not contain any offscreen areas, then no off-screen structures need be defined.

Using Mode 0 of the SDB port as an example, 3 offscreen memory areas are available and are defined as follows:

```
Off_Screen 00000B00h,640/4,480 ; Alloc.to offscrn wksp 0
Off_Screen 00000D80h,160,480   ; Offscreen area 1
Off_Screen 001E0000h,1024,32   ; Offscreen area 2
```

Note that the first Off_Screen block defined is intended to be used for the offscreen workspace. The Mode_Wksp_Pitch and Mode_Wksp_Addr fields of the Mode_Info structure for Mode0 are initialized to point to this block.

Be sure and note the maximum number of offscreen areas defined in any operating mode of your board. This value will be required later when porting the TIGA graphics manager.

D.1.9 Defining OEM-Specific Data

If you have any other mode-specific data, it should be added to the operating mode data using the OEM_Data structure. This structure, defined in the `macro.inc` file, follows the Off_Screen structure data. To use the OEM_Data structure, modify the number of parameters expected by the OEM_Data macro in the `macro.inc` file. Then, supply the OEM-specific data using the OEM_Data macro. A corresponding change is required in the graphics manager portion of TIGA to support this new data.

Note:

In the SDB port, no OEM_Data is defined, but an example of its usage is shown.

D.1.10 Completing Modifications to `oemdata.asm`

Repeat the above instructions to define all operating modes for your particular board.

After the last mode information block, global data variables used by the CD are declared. The Previous_Mode variable may require changing. This variable is used to store the TMS340 board emulation mode (that is, EGA, VGA) prior to loading TIGA. Because the SDB does not support emulations, the Previous_Mode is set to TIGA. However, if your board does support emula-

tions, then initialize this variable to the emulation mode in which the board powers up. If the emulation is configurable by DIP switches, then an appropriate function called within the **oem_init** function (see Section D.1.11) should be written to initialize this variable. Valid constants for emulation modes are defined in the file `\tiga\include\tiga.h` file.

Finally, the DRAM_Start and DRAM_End symbols need to be initialized to the largest block of DRAM on the target board.

DRAM_End is used to store the high-water mark in memory where the system stack resides. The address should be double-word aligned and must not be higher than $0FFFFDFE0_h$, since memory above this address is reserved in the TMS340 memory map. This address should also be equal to the bottom_of_stack value in the link control file of the graphics manager.

D.1.11 Modifying the `oeminit.asm` File

The `oeminit.asm` file contains functions used to initialize a specific TIGA-compatible TMS340 target board. The functions in `oeminit.asm` shipped with the TIGA Software Porting Kit perform specific initializations for the SDB and therefore require modifications for your particular board.

The `oeminit.asm` file contains the following four board-specific initialization functions:

OEM_Init	Initializes the board.
OEM_Sense	Returns ID of current monitor in use.
Monitor_Init	Initializes the TIGA mode table with all valid modes for current monitor in use.
Video_Enable	Switches video from EGA/VGA to TIGA.

D.1.12 Modifying the OEM_Init Function

The OEM_Init function performs all initializations necessary to put the target TMS340-based board in a state where the TIGA graphics manager can be loaded to it. The TIGA communications driver calls the OEM_Init function when it is initially loaded.

Using the SDB port as an example, the OEM_Init function first halts the TMS340, flushes the cache, and sets the INCR and INCW bits in the TMS340's HSTCTL register. It then switches on the SDB's shadow RAM, clears the interrupt enable I/O register so that the board TMS340 can be halted later, and clears the TMS340's CONTROL register.

Note the comment within the OEM_Init function regarding the setting of the CONTROL register. It is extremely important to initialize the CONTROL

register properly to support the type of DRAM refresh cycles used on the board.

Note:

Do not modify the code related to the high-water mark.

D.1.13 Modifying the OEM_Sense Function

The OEM_Sense function is used to return the ID of the monitor that is connected to the target TMS340-based board. The valid monitor IDs were previously defined in the `newdefs.inc` file.

In the SDB port, there is no way to sense the type of monitor using the SDB. However, on other boards, DIP switches or monitor sense lines are available for this purpose. This example function searches for a `-s` command line argument and sets `dx` to the Sony id if found. Otherwise, `dx` is set to NEC.

D.1.14 Modifying the Monitor_Init Function

The monitor_init function calls OEM_Sense to get the current monitor connected to the target TMS340-based board. It then steps through the list of all modes defined in the Setup_Struc structures (defined in `oemdata.asm`) and puts the indices of those modes that support the current monitor into space allocated in Mode_Table (defined in `oemdata.asm`). This, in essence, defines the total number of modes available for the current monitor in use.

This function requires no modifications except when more than 16 monitors are supported.

D.1.15 Modifying the Video_Enable Function

This function invokes a graphics manager function that does nothing on the SDB, because there is no EGA emulation implemented on the SDB. Video_Enable is shown here as an example of its implementation. If the video on your board can be switched directly from the host side, then modify this function to do so. Otherwise, as illustrated in the SDB port, call the Video_Enable graphics manager function to perform the video switch.

D.1.16 Modifying the `setvideo.asm` File

The two functions within `setvideo.asm`, **set_videomode** and **get_videomode**, require porting only for those TMS340-based boards which support other graphic modes (that is, EGA, VGA, etc.). Because the

SDB does not support any other graphics modes, comments are given within the `setvideo.asm` code to offer suggestions on alternative graphics mode support.

D.1.17 Miscellaneous Communication Driver Porting Issues

The following sections describe additional modifications that may be needed for porting the TIGA communications driver to your specific board.

D.1.18 Default Timeout for gm_is_alive Function

The file `error.asm` contains the function **gm_is_alive**. The purpose of this function is to check if the TIGA graphics manager (the part of TIGA which runs on the TMS340-side) is alive and running. It does this by installing its own error trap, calling one of TIGA's core primitives, and then waiting a certain period for the TIGA primitive to complete. If the function does not complete in the time allotted, the error handler is called and false (0) is returned, indicating a non-functioning graphics manager.

The delay time is defined by the constant GM_TIMEOUT in the file `newdefs.inc` and is set in the SDB port to 0.5 seconds. This timeout value may have to be lengthened on those boards with monitor screen delays longer than 30 frames (see Defining the Monitor_info Structure in Section D.1.6).

This potential problem is evident when an application calls a function requiring the use of TIGA's linking loader TIGALNK (that is, **create_alm**) immediately following a call to **set_videomode**(TIGA,INITIALIZE). Because the call to **set_videomode** reloads the video timing registers (using the delay defined by the monitor delay amount), and `TIGALNK` first checks if the TIGA graphics manager is alive (via **gm_is_alive**), there is a chance that this delay will cause **gm_is_alive** to fail. `TIGALNK` then returns an error message indicating that the ALM file could not be created (in this example).

D.1.19 Using Boards with Multi-Addressable Host Port Locations

TMS340-based boards with multi-addressable host port locations were mentioned in Section D.1.1. Because the host port addresses are hard-coded in the `newdefs.inc` file, only one of the address sets are supported. However, by making some simple modifications, programmable host port addressing is possible.

First, five 16-bit variables must be declared in the CD data section (in the file `data.asm`). These variables are

```
HstCtrlAdr    dw   ?    ; HSTCTRL host address
HstAdrlAdr    dw   ?    ; HSTADRL host address
```

```
HstAdrhAdr    dw   ?    ; HSTADRH host address
HstDataAdr    dw   ?    ; HSTDATA host address
HstSegAdr     dw   ?    ; Segment address (memory map only)
```

Next, these addresses must be initialized before any I/O is performed through the TMS340's host port.

The file `macro.inc` contains all the macros that perform I/O functions through the TMS340 host port. For example, Write_HSTDATA is a macro which writes 16 bits of data to the TMS340 HSTDATA register. For memory mapped boards, ax is assumed destroyed by these macros. For I/O-mapped boards, ax and dx are assumed destroyed. Currently, these macros assume hardcoded addresses for the host port locations. To modify the macros, use the following Write_HSTDATA example as a guideline:

```
Write_HSTDATA   Macro   Reg, Seg
  Local  SegReg
  Ifb <Seg>
SegReg  Equ     <es>
  Else
SegReg  Equ     <Seg>
  Endif
  push    bx    ; We must save bx (not assumed destroyed)
  mov bx,ds:HstDataAdr    ; Load bx with address of HSTDATA
  If  MEMORY
    Ifidni  <Reg>,<bx>    ; If reg passed == bx
      mov ax,Reg          ; then copy to ax
      mov SegReg:[bx],ax  ;  and send it
    Else
    mov SegReg:[bx],Reg   ; otherwise, send the reg
    Endif
  Else
    Set_IO  [bx]          ; Set io address
    Ifidni  <Reg>,<ax>    ; If reg passed == ax
      out dx,ax           ; then output ax
    Else
      mov ax,Reg          ; otherwise, move reg to ax
      out dx,ax           ; and output ax
    Endif
  Endif
  pop     bx              ; Restore bx
Endm
```

You must also search through each of the `.asm` files in the `\tiga\cd` directory and replace all occurrences of HSTSEG with the appropriate value in the HstSegAdr variable.

D.1.20 Rebuilding the Communication Driver

Rebuilding the TIGA communication driver is a simple operation. First, edit the file `makecd.bat`. This batch is designed to rebuild the communication driver for the SBD and must be modified for your board as follows:

Step 1: Change lines 3 and 4 to check for an id for your board. Also, change the label from SDB to a label for your board.

Step 2: Change the label on line 7 to your new label.

Step 3: Modify the description on line 10.

Step 4: Change line 15 to copy `newdefs.inc` into `defs.inc`.

After making the above modifications, enter **makecd** *yourid* from the MS-DOS command line from the `\tiga\cd` directory. Note that *your id* is the identification added to the `makecd.bat` batch file in step 1 above.This rebuilds the TIGA communication driver `tigacd.exe` and copies it into TIGA's main directory `\tiga`.

D.2 Porting the Graphics Manager

The TIGA Graphics Manager (GM) is the portion of TIGA that resides on the target graphics board (the TMS340 side). All files associated with the TIGA GM are installed in the \tiga\gm directory, with specific portions of the GM split into subdirectories under the \tiga\gm directory as follows:

Directory	Contents
\tiga\gm	TIGA graphics manager command executive
\tiga\gm\corprims	TIGA GM core primitives
\tiga\gm\extprims	TIGA GM extended primitives
\tiga\gm\sdb	TIGA GM board-specific functions

It is suggested that another directory be created under the \tiga\gm directory to contain board-specific functions for your particular board. For example, assume a directory called \tiga\gm\newgm exists and that all of the files from the \tiga\gm\sdb directory have been copied into the \tiga\gm\newgm directory.

The majority of the modifications necessary to port the TIGA GM are made to the board-specific functions in the \tiga\gm\newgm directory.

The following sections describe in detail these modifications. Note that all references to filenames are assumed to be files within the \tiga\gm\newgm directory unless otherwise noted.

The board-specific functions in the \tiga\gm\newgm directory are grouped into four main categories:

- Clearing video memory,
- Palette-specific functions,
- Configuration functions, and
- Miscellaneous functions.

D.2.1 Video Memory Initialization Functions

clearfrm.asm The **clear_frame_buffer** function uses the fastest possible method to clear the entire frame buffer to a specified color. In the SDB port, shift register transfer cycles are used. If this is not possible on the target TMS340 board, use a FILL (see the **clear_screen** function in the file clearscr.asm for information on how this is done).

clearpag.asm The **clear_page** function uses the fastest possible method to clear the entire current drawing page to a specified color. Because shift register transfers cannot be used to clear

only a portion of the entire frame buffer on the SDB, the SDB port simply calls the **clear_screen** function, which performs a FILL. The port of this function should use shift register transfers if possible.

`clearscr.asm` The **clear_screen** function uses the fastest possible method to clear the visible portion of the current drawing page to a specified color. Although this file is not grouped with the other board-specific, video memory initialization functions, it may be ported to utilize a faster screen clearing method. The SDB port uses the FILL instruction.

D.2.2 Palette-Specific Functions

The following functions provide the capability to utilize a color palette on the target board. The SDB board uses the TMS34070 color palette, and these functions are therefore written to use the TMS34070. They must be modified if your board has a color palette other than the TMS34070.

`getpalet.c` The function **get_palet** returns the values in the global palette array `palet` stored in TMS340 memory. Generally, this function does not require any porting.

`initpale.c` The function **init_palet** sets the palette to EGA default colors. It should replicate these colors through the entire palette. On the SDB, this is trivial since it has only16 entries. Where there are more entries, this will need to be done in a loop. If the palette is in ROM and no initialization is possible, this function should not be implemented and its entry should be cleared in the `\tiga\gm\primdefs.c` file to insure **function_implemented** returns false.

`setpalet.asm` The **set_palet** and **set_palet_entry** functions within `setpalet.asm` perform initialization of a TMS34070 palette which stores the palette information in the frame buffer. Note that the data stored in the global array `palet` (declared in `oem.c`) is the physical color. The LS 4 bits are masked because they are not used in the TMS34070. If the palette is in ROM and no initialization is possible, this function should not be implemented and its entry should be cleared in the `\tiga\gm\primdefs.c` file to ensure **function_implemented** returns false.

Note that the palet routines use the `monitor_flags` field of the `monitor_info` structure to determine if the monitor is color or monochrome. This field selects the use of either the `r,g,b` values or the `i` value to initialize the palet.

D.2.3 Configuration Functions

`config.c` The **get_config** function does not require porting. The **set_config** function may require porting if your board requires specific initializations for a particular mode. This initialization can be performed wherever convenient within the **set_config** function.

An example of an initialization that may be required for your board involves adding support for OEM mode data specified in the Setup structure. If OEM-specific data was added to the `oemdata.asm` file during the CD port (via the OEM_Data structure, see Section D.1.9), then the graphics manager portion of TIGA must be modified to support this data. Normally, the code to handle OEM-specific data is added to the **set_config** function.

`oem.h` This file contains constants and type definitions used to initialize and maintain mode information in the graphics manager. The constants VIDEO_MEMORY_START, VIDEO_MEMORY_END, and PALET_ENTRIES should be modified to match your board specifications. PALET_ENTRIES should contain the number of palette entries in your default board mode. The three shared memory constants, SHARED_MEM_SIZE, SHARED_HOST_ADDR, and SHARED_GSP_ADDR should be initialized to values other than 0 if the board supports shared memory. If it does not, (as in the SDB port), initialize to 0.

The DRAM_RM_RR constant defines the DRAM refresh mode and refresh rate. It is ORed with TIGA's default CONTROL value to initialize the TMS340 processor CONTROL register. The value of this constant in the SDB port, 0x08, specifies RAS-only refresh (RR field = 0) and refresh to occur every 64 local clock periods (RM field = 01).

The communication buffers, used by TIGA to buffer commands between the host and the TMS340, are declared statically in TMS340 memory. The `oem.h` file contains two constants that define the size and number of these buffers.

The NUM_COMM_BUFFERS constant defines the number of communication buffers. Each communication buffer contains the information for one command. Therefore, the more communication buffers defined, the more commands that can be queued. The recommended minimum is three communication buffers.

The COMM_BUFFER_SIZE constant defines the size (in bytes) of the data area in each communication buffer. The value assigned to this constant

must be at least 1K (1024) bytes and a multiple of 2. This minimum size is necessary to ensure that data less than 1024 bytes long can be sent to the TMS340 processor without fear of overflowing the communication buffer.

The MAX_PAGES and MAX_OFFSCREEN constants must be set to the maximum number of pages and offscreen areas, respectively, in **any** operating mode defined in the communication driver port (see Section D.1.7, which defines the PAGE_INFO Structure and Defining the OFF_SCREEN structure for more information). This is to insure that there is sufficient memory allocated to download these structures to the GM.

The HEADER and SETUP structures may require modification to support any OEM-specific data added to the CD port (via the OEM_Data structure, see Section D.1.9 for more information).

`oem.c` This file contains the initial configuration and setup structures. The data contained in these default structures is used to initialize the TIGA environment when the graphics manager is initially executed (that is, the GM banner message is displayed), and also to statically define an area in TMS340 memory where mode information from the host is downloaded. This configuration is overwritten almost immediately thereafter by the current mode information downloaded by the host. The data in Default Modeinfo, Default Monitor info, Default Page and Default Offscreen should be initialized with one mode from the `oemdata.asm` file in the communication driver.

In a future release of TIGA, the copying of the setup structure into a fixed length default structure may well be changed to use system heap, thereby enabling the setup structure to be dynamic in size. This initial configuration is also useful because it enables debug messages to be printed from the graphic manager's main loop during debug of the GM port.

D.2.4 Miscellaneous Functions

`initvide.c` The function **init_video_regs** initializes the video registers for the new mode. This function does not require porting unless the target board needs initialization of some other board-specific latches relating to video timing.

`videnbl.c` In the SDB port, the **video_enable** function does nothing. However, for a board that uses a separate frame buffer for alternate graphics modes (that is, EGA), the **video_enable** function switches the

back-end to display the frame buffer for the correct mode. It is called by the host side function **set_videomode**.

`sysfont.asm` Two system fonts are supplied with the TIGA port: `sys640.asm` and `sys1024.asm`. The former is designed for low resolution (640 by 480 and below), the latter for high resolution (1024 x 768 and above). To select a particular system font, copy it into the file `sysfont.asm`. This is the files that will actually be assembled and linked into the graphics manager when it is rebuilt.

`trapvect.asm` The functions within the `trapvect.asm` file perform I/O with the TMS340 processor interrupt trap vectors. This file requires modifications if the interrupt traps for your board are located in ROM or are not contiguous in memory from TMS340 address $0FFFFFC00_{16}$ (trap 31) to $0FFFFFFE0_{16}$ (trap 0).

`\tiga\gm\primdefs.c` This file defines all implemented core functions provided by theTIGA graphics manager. The addresses of functions that are not implemented in your specific board port should be cleared and their declarations removed to insure **function_implemented** will return false for these functions. Consult the **function_implemented** description in Chapter 3 for a list of functions that are likely not to be implemented on all boards.

`\tiga\gm\gmdefs.c` The only modification required in the `\tiga\gm\gmdefs.c` file is the pathname of the OEM-specific header include file (line 10). Modify the pathname of this file to include your OEM-specific definitions.

`\tiga\gm\tigagm.inc` The include file `\tiga\gm\tigagm.inc` contains a label named OEMMSG as an identifying string for your board. Modify the string to the name of the board being ported to. This message is displayed when the graphics manager is executed.

There are two possible modes of cursor operation in TIGA. The default mode (recommended method) is a display interrupt driven cursor. This entails the cursor being redrawn every frame, which is acceptable for most systems. In very high resolution displays, however, the overhead of a display

interrupt cursor may be unacceptable, in which case a hide/show cursor mechanism may be used. To obtain a hide/show cursor, modify the value of NON_DI_CURSOR from 0 to 1. The latter cursor flashes quite noticeably, whereas the display interrupt cursor is solid.

D.2.5 Rebuilding the Graphics Manager

Rebuilding the TIGA graphics manager consists of four parts:

1) Rebuilding the TIGA core primitives
2) Rebuilding the TIGA extended primitives
3) Rebuilding the board-specific functions
4) Rebuilding the TIGA command executive

The batch file `\tiga\gm\makegm.bat` does all of this automatically. However, since this batch file and all others associated with rebuilding the GM were designed for the SDB port, a few changes are required before rebuilding your GM port.

`\tiga\gm\makegm.bat`

This is the main batch file that rebuilds the TIGA GM. It is invoked from the DOS command line with one argument.

The following SDB port dependencies may have to be modified:

- The batch file argument id of SDB
- The board-specific directory (`\tiga\gm\sdb`) and make file name (`sdblib.mak`)
- The board-dependent library filename (`\tiga\gm\sdb\sdb.lib`)
- Comments

After building the GM, `makegm.bat` automatically attempts to create the new GM symbol file `\tiga\tiga340.sym` by first running `tiga.cd` and then `tigalnk /cs`. It is for this reason that the CD port should be completed prior to porting the GM.

`\tiga\gm\tigagm.cmd`

This is the link command file used by `gsplnk.exe` to build the TIGA graphics manager out file `tigagm.out`.

The following SDB port dependencies may have to be modified:

- The board-dependent library filename (\tiga\gm\sdb\sdb.lib)
- The values assigned to the labels:
  _start_of_dram
  _bottom_of_stack
  _stack_size
  _end_of_dram
- The values assigned to program:
  origin
  length
- Comments

Note:

The label _bottom_of_stack must be double-word (32-bit) aligned, (that is, the 5 LSBs must be zero) and must correspond to the address DRAM_End in the communication driver. See Section D.1.10.

`\tiga\gm\tigagm.mak`

This make file rebuilds the GM command executive and links all portions of the GM to form the out file `tigagm.out`.

The following SDB port dependency may have to be modified:

The board-dependent library filename `\tiga\gm\sdb\sdb.lib` located in the list of dependencies for `tigagm.out`.

`\tiga\gm\newgm\sdblib.mak`

This make file rebuilds the board-dependent library.

The following SDB port dependencies may have to be modified:

- The filename of the make file itself
- The filename of the board-dependent library (`sdb.lib`)

After modifying the above files, change your current directory to `\tiga\gm` and enter: **makegm** *yourid*, where *yourid* is the board identifier you added to the `makegm.bat` batch file. The TIGA graphics manager is then built and the output file `tigagm.out` copied into the TIGA system directory `\tiga`.

D.3 Verifying Correct Operation

Included with the TIGA SPK is a comprehensive test suite designed to test different aspects of a TIGA port. This test suite can be run from the `\tigapgms\tests` directory by entering **tigatest** from the DOS command line.

It is suggested that the tests be run in the order displayed on the menu and that problems be fixed as they are encountered.

After verifying correct operation of the test suite, try some of the other test programs supplied in the `tigapgms` directory.

D.4 Debugging Your Port

The TIGA communication driver can be easily debugged using Microsoft's CodeView(R) or comparable debugger. The TIGA graphics manager can be debugged by using your board's TMS340 debugger. The following are some suggestions for debugging the TIGA GM:

The TIGA graphics manager initializes all TMS340 trap vectors upon startup. This can cause havoc with your debugger, which may initialize trap vectors (that is, for single-step capability) before loading and executing the TIGA GM. To overcome this problem, modify the **init_trap_vectors** function in the file `\tiga\gm\sdb\trapvect.asm` so that trap vectors used by your debugger are not overwritten.

After displaying its startup message, the GM then waits for handshaking to occur with the host. If necessary, a host TIGA application should be written that performs the handshake with the TIGA GM as follows:

```
#include <tiga.h>
main()
{
   set_videomode(TIGA,NO_INIT);
   handshake();
}
```

After performing the handshake, the GM command executive waits for a TIGA command from the host.

Appendix E

Debugger Support for TIGA

TIGA is the definitive interface standard for applications software written to run on the TMS340 architecture, but it gives no guidelines to developers of software with special hardware accessing requirements, such as debuggers.

A set of routines has been included in TIGA to meet the often unique needs of debugger developers. This appendix contains the initial TIGA debugger routines developed.

Section **Page**

E.1 TIGA Debugger Routines E-2
E.2 Compatibility Functions E-12

E.1 TIGA Debugger Routines

A separate document describing the use of the debugger functions will be published in the future. The debugger routines development will be based on the following criteria and on any user feedback received:

1) Portable to any TIGA environment, which potentially includes all TMS34010- and TMS34020-based PC graphics displays.
2) Transparent to share the TMS34010's host interface registers with an application being debugged that uses the host interface for communication between host and TMS340 resident software.
3) Able to support the symbolic debug of RLMs (Relocatable Load Modules), if running in an environment where the TIGA graphics manager is active.

The following is a list of the routines in TIGA that provide debugger support:

- **get_vector**: Get contents of GSP trap vector
- **set_vector**: Set contents of GSP trap vector
- **set_xstate**: Set GSP execution state
- **get_xstate**: Get GSP execution state
- **get_memseg**: Get high/low bounds of GSP memory segment
- **set_memseg**: Set high/low bounds of GSP memory segment
- **set_msg**: Send a message to the GSP
- **get_msg**: Receive a message from the GSP
- **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 Section 3.3 because their usefulness is not restricted to debuggers.

Syntax

```
void get_memseg( addrlo, addhi );
   unsigned long *addrlo, *addrhi;
```

Type

Core

Description

The **get_memseg** function is not for general use. It is provided for use by 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 be used to 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 Core

Description The **get_msg** function is not for general use. It is provided for use by 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 Core

Description The **get_xstate** function is not for general use. It is provided for use by debuggers and other such tools which 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
- Bits 5—15 — Reserved for future use

Example

```
#include <tiga.h>
main()
{
  if (cd_is_alive())
  {
    if (get_xstate() & 1)
      printf("GSP is halted\n");
    else
      printf("GSP is running\n");
   }
}
```

Syntax `unsigned short rstr_commstate();`

Type Core

Description The **rstr_commstate** function is not for general use. It is provided for use by 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 unable to save the state, nonzero if it is successful.

Syntax `unsigned short save_commstate();`

Type Core

Description The **save_commstate** function is not for general use. It is provided for use by debuggers and other such tools that have special hardware accessing requirements. This function saves the state of TMS340 communications. The function returns zero if unable to save the state, nonzero if it is successful.

Syntax

```
void set_memseg(addrlo, addhi);
   unsigned long addrlo, addrhi;
```

Type Core

Description The **set_memseg** function is not for general use. It is provided for use by 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 after using some of the memory returned by **get_memseg** to reflect the new memory size.

Syntax

```
void set_msg( msg );
   unsigned short msg;
```

Type

Core

Description

The **set_msg** function is not for general use. It is provided for use by 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 Core

Description The **set_xstate** function is not for general use. It is provided for use by debuggers and other such tools which 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 NMI
- Bit 3 — 1 to flush cache, 0 to stop cache flush
- Bit 4 — 1 to disable cache, 0 to enable cache
- Bits 1 —15 — Reserved for future use, must be set to 0s

Example

```
#include <tiga.h>
main()
{
  if (cd_is_alive())
  {
    set_xstate(1);  /* halt the GSP                    */
    set_xstate(0);  /* run the GSP                     */
  }
}
```

Syntax `void oem_init()`

Type Core

Description This function halts the TMS340 and performs any board-specific initialization prior to loading a COFF file.

E.2 Compatibility Functions

It is recommended that the compatibility functions **not** be used by an application programmer. Their functions can be performed by the entry points in the previous section and by the communication functions described in Chapter 3. These functions talk directly to TMS34010 hardware, which is not present on the TMS34020, and 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, in 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 Core

Description This function returns the contents of the host address register of the TMS34010.

Syntax `unsigned short read_hstadrh();`

Type Core

Description This function returns the contents of the host address high register of the TMS34010.

Syntax `unsigned short read_hstadrl();`

Type Core

Description This function returns the contents of the host address low register of the TMS34010.

Syntax `unsigned short read_hstctl();`

Type Core

Description This function returns the contents of the host control register of the TMS34010.

Syntax `unsigned short read_hstdata();`

Type Core

Description This function returns the contents of the host data register of the TMS34010.

Syntax

```
void write_hstaddr(value)
    unsigned long value;
```

Type

Core

Description

This function writes the 32-bit value supplied into the host address register of the TMS34010.

Syntax

```
void write_hstadrh(value)
    unsigned short value;
```

Type Core

Description This function writes the 16-bit value supplied into the host address high register of the TMS34010.

Syntax

```
void write_hstadrl(value)
   unsigned short value;
```

Type Core

Description This function writes the 16-bit value supplied into the host address low register of the TMS34010.

Syntax

```
void write_hstctl(value)
   unsigned short value;
```

Type Core

Description This function writes the 16-bit value supplied into the host control register of the TMS34010. Note that in order to function properly, TIGA expects the values of the INCR, INCW, and LBL bits in host control to be set in a particular manner. If these bits are modified, they **must** be restored prior to invoking another TIGA function or the TIGA environment may be corrupted.

Syntax

```
void write_hstdata(value)
    unsigned short value;
```

Type Core

Description This function writes the 16-bit value supplied into the host data register of the TMS34010.

Appendix F

Glossary

A

ADI™: *Autodesk Device Interface*, an interface specification used for developing customized drivers for peripheral devices for Autodesk products.

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 potential of appiying 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 when specifying 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-340 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 that 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 primitives: A group of TIGA functions that can **always** be invoked by an application after TIGA has been installed, as opposed to the extended primitives, which 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, this refers to a graphics cursor, which is an icon on the screen. The cursor is generally under mouse control and is used as a pointing device in a graphics application.

D

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. It consists of a TIGA driver for the Texas Instruments software development board (SDB), the TIGA application interface (AI), and example programs.

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).

E

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 primitives: A portion of the TIGA interface functions that can be invoked ***only*** by a TIGA application after they have been explicitly installed.They consist of mostly drawing primitives.

extensibility: A key feature of TIGA 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.

F

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.

G

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 primitives. Some of these are integral TIGA primitives 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.

H

heap: An area of memory that a program can allocate dynamically.

I

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 it refers to a company that writes a software product to interface directly with TIGA. ISVs include Microsoft, Autodesk, etc.

L

LIM ™: *LIM expanded memory.* This system was developed by Lotus, Intel, and Microsoft, to define a hardware and software interface for 80x86 processors running under MS-DOS. LIM provides access to bank-switched random access memory.

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.

M

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 esentially the same operating system, this manual uses the term MS-DOS to refer to both systems.

N

NMI: *Non-Maskable 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:** SomeTIGA graphics output primitives use a pen 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 using the current background color, and 1s are drawn using the current foreground color.

pen or **drawing pen:** SomeTIGA graphics output primitives use a pen to draw an outline. The drawing pen has application-selectable width and height. 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: ***Pixel Block 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 multi-bit operations.

plane: (Also bit plane or color plane). A plane is 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 the 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: 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. The SDB contains a TMS34010 graphics processor. The two TIGA kits (DDK and SPK) produced by Texas Instruments use the SDB as their target 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. This kit contains the DDK and tools such as the C compiler, assembler, and linker.

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 as well as the SDK.

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 primitives that are resident on the TMS340 board.

T

TIGA ™, TIGA-340 ™ : *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: This is the file name of the executable program containing the communication driver that you run to install TIGA on your system.

TIGALNK: See linking loader.

time-out: An application invokes a TIGA TMS340 function by placing a command number and appropriate parameters in one of several com-

mand buffers. After loading several commands, the command buffers may be full; the host has to 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: When a pixel with the attribute of transparency is written to the screen, it is effectively invisible, and 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.

W

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, 3-12, 3-22, 3-96, 4-2, 4-8, 4-48, 4-49
  installation, 4-8, 4-43
application interface, AI, 1-4, 2-12, 3-2
  DDK, 2-5
  SPK, 2-5
attributes, 3-5, 3-6, 3-9, 3-75, 3-76, 3-95, 3-167, 3-179, 3-180, 4-33, A-7

B

background color, 3-5, 3-55, 3-141, 3-143, 3-145, 4-33
bitblt, 3-8, 3-9, 3-14, 3-155, 3-166, 3-184, 4-36

C

C-packet, 4-10, 4-11, 4-13, 4-18, 4-36, 4-41
cc utility, 2-12
cd_is_alive, 3-3, 3-16, 3-86
cl, 2-17
clear functions, 3-4
clear_frame_buffer, 3-4, 3-17
clear_page, 3-4, 3-18, 3-19
clear_screen, 3-4, 3-17, 3-18, 3-19
cltiga batch file, 2-12
COFF loader, 3-86, 4-47
comm_buff_size, 3-57, A-3
command buffer, 1-5, 3-3, 3-8, 3-35, 3-40, 3-44, 3-57, 3-112, 3-118, 3-129, 3-135, 4-12, 4-13, 4-16, 4-18, 4-36
command number, 1-5, 3-103, 4-4, 4-10, 4-11, 4-13, 4-36, 4-43
communication buffer. *See* command buffer
communication driver, CD, 1-4, 2-5, 2-6, 2-10, 2-11, 3-3, 3-8, 3-16, 4-12, 4-13, 4-16, 4-47
communication functions, 1-4, 3-11
compatibility functions, E-12
CONFIG structure, 3-3, 3-6, 3-8, 3-10, 3-14, 3-35, 3-40, 3-44, 3-57, 3-64, 3-66, 3-69, 3-90, 3-108, 3-112, 3-118, 3-129, 3-135, 3-144, 4-13, 4-16
  See also MODEINFO structure
cop2gsp, 3-11, 3-20
coprocessor, 3-12, 3-20, 3-57, 3-81
core primitives, 1-6, 2-5, 2-8, 3-2, 4-10, 4-11, 4-32, C-3
cp_alt, 4-13, 4-14
cp_cmd, 4-13, 4-14
cp_ret, 4-13
cpw, 3-5, 3-21
create_alm, 3-12, 3-22, 3-159, 4-2, 4-8, 4-9, 4-32, 4-43, 4-47
create_esym, 3-12, 3-24, 4-32, 4-47
current_mode, 3-57, 3-108, A-3
cursor, 3-10, 3-59, 3-60, 3-90, 3-145, 3-146, 3-152, 3-153, 4-33, 4-44, A-5

D

debugger functions, E-2
delete_font, 3-9, 3-26
demonstrations and examples, 2-6
device_rev, 3-57, A-3

D

debugger functions, E-2
delete_font, 3-9, 3-26
demonstrations and examples, 2-6
device_rev, 3-57, A-3
direct mode, 4-10, 4-11, 4-12, 4-18, 4-32, 4-36, 4-41
disp_hres, 3-58, A-11
disp_pitch, 3-14, 3-58, A-11
disp_psize, 3-58, A-11, B-9
disp_vres, 3-58, A-11
display_mem_end, 3-57, A-4
display_mem_start, 3-57, A-3
dm_cmd, 4-18
dm_ipoly, 4-17, 4-28
dm_palt, 4-24
dm_pcmd, 4-25, 4-41
dm_pget, 4-23
dm_poly, 4-17, 4-26
dm_pret, 4-25
dm_psnd, 4-21
dm_pstr, 4-24
dm_ptrx, 4-24
dm_ret, 4-20
draw_line, 3-7, 3-27, B-2, B-7
draw_oval, 3-7, 3-29, B-2, B-7
draw_ovalarc, 3-7, 3-31, B-2, B-7
draw_piearc, 3-7, 3-33, B-2, B-7
draw_point, 3-7, 3-34, B-2, B-7
draw_polyline, 3-7, 3-8, 3-35, B-2, B-7
draw_rect, 3-7, 3-37, B-2, B-7
drawing origin, 3-6, 3-60, 3-61, 3-72, 3-142, 3-145, 3-153, 3-154, A-6, B-5
driver, 2-10
driver developer's kit, DDK, 1-2, 2-4
  subdirectories, 2-5
  system requirements, 2-2

E

ENVIRONMENT structure, 3-5, 3-61, 3-145, 4-33, A-6
environment variable, 2-11, 2-17, 3-22, 3-24, 3-49, 3-61, 3-96, 3-99, 3-101, 4-7, 4-36, 4-47
  autoexec modification, 2-9
extended primitives, 2-5, 2-8, 3-2, 3-3, 3-50, 3-99, 4-10, 4-11, 4-43, B-1, C-5
  DDK, 2-5
extensibility, 1-3, 1-4, 1-6, 3-12, 3-22, 3-24, 3-49, 3-50, 3-63, 3-90, 3-96, 3-99, 3-101, 3-159, 4-1

F

field_extract, 3-11, 3-16, 3-38, 4-32
field_insert, 3-11, 3-16, 3-39, 4-32
fill_convex, 3-7, 3-8, 3-40, B-2, B-6
fill_oval, 3-7, 3-42, B-2, B-6
fill_piearc, 3-7, 3-43, B-2, B-6
fill_polygon, 3-7, 3-8, 3-44, 3-80, 3-174, B-2, B-6
fill_rect, 3-7, 3-48, B-2, B-6
floating point, 2-6, 4-40, 4-41
floating point coprocessor. *See* coprocessor
flush_esym, 3-12, 3-49, 4-32
flush_extended, 3-12, 3-50, 4-32
font, 2-13, 3-9, 3-26, 3-62, 3-95, 3-97, 3-140, 3-145, 3-179, 3-180, A-7
FONTINFO structure. *See* font
foreground color, 3-5, 3-55, 3-143, 3-145, 3-158, 4-33
frame_oval, 3-7, 3-51, B-2
frame_rect, 3-7, 3-52, B-2
function_implemented, 3-3, 3-6, 3-20, 3-53, 3-69, 3-70, B-9

G

get_colors, 3-5, 3-55
get_config, 3-3, 3-35, 3-57, 3-64, 3-90, 3-112, 3-118, 3-129, 3-135, 3-144, 4-13, 4-16
get_curs_state, 3-10, 3-59
get_curs_xy, 3-10, 3-60
get_env, 3-5, 3-61, 3-175

get_fontinfo, 3-9, 3-62
get_isr_priorities, 3-12, 3-63, 3-96, 3-101, 3-159, 4-32, 4-44, 4-45
get_modeinfo, 3-3, 3-57, 3-64, 3-144, 4-32, A-11
get_nearest_color, 3-6, 3-65, B-9
get_offscreen_memory, 3-11, 3-58, 3-66, 3-174, A-12, A-14
get_palet, 3-6, 3-69, B-9
get_palet_entry, 3-6, 3-69, 3-70
get_pixel, 3-10, 3-72
get_pmask, 3-5, 3-73, 3-164
get_ppop, 3-5, 3-74, 3-164
get_textattr, 3-9, 3-75
get_transp, 3-5, 3-76
get_vector, 3-11, 3-77
get_videomode, 3-3, 3-78, 3-171, 4-32
get_windowing, 3-5, 3-79
get_wksp, 3-8, 3-44, 3-80
graphics manager, GM, 1-5, 2-5, 2-10, 3-3, 3-8, 3-16, 3-24, 3-86, 3-96, 3-101, 4-14, 4-32, 4-35, 4-44, 4-47, 4-49
graphics output functions, 3-7, B-2
graphics utility functions, 3-10
gsp_calloc, 3-11, 3-85
gsp_execute, 3-3, 3-16, 3-86, 3-106
gsp_free, 3-11, 3-87
gsp_malloc, 3-11, 3-66, 3-88, 3-162, 3-174
gsp_maxheap, 3-11, 3-89
gsp_minit, 3-11, 3-57, 3-90
gsp_realloc, 3-11, 3-91
gsp2cop, 3-11, 3-53, 3-81
gsp2gsp, 3-11, 3-82
gsp2host, 3-11, 3-16, 3-83, 4-32
gsp2hostxy, 3-11, 3-16, 3-84, 4-32

H

host2gsp, 3-11, 3-16, 3-92, 4-32
host2gspxy, 3-11, 3-16, 3-93, 4-32

I

include files, 2-5, 2-8, 2-9, 2-18, 4-11, 4-14, 4-32, 4-43, A-1
init_palet, 3-6, 3-53, 3-94, B-10
init_text, 3-9, 3-95
initialization, 2-10, 3-3, 3-90, 3-94, 3-95, 3-145, 4-46, 4-49
install_alm, 3-12, 3-63, 3-96, 3-159, 4-3, 4-9, 4-11, 4-32, 4-43, 4-47
install_font, 3-9, 3-95, 3-97
install_primitives, 3-3, 3-12, 3-99, 4-10, 4-32
install_rlm, 3-12, 3-24, 3-63, 3-101, 3-159, 4-3, 4-7, 4-11, 4-32, 4-38, 4-45, 4-47
install_usererror, 3-3, 3-4, 3-35, 3-41, 3-45, 3-103, 3-113, 3-119, 3-129, 3-135, 3-168, 3-171, 4-32
installation, 2-4, 4-7, 4-38, 4-44
interrupt, 2-6, 2-11, 3-12, 3-63, 3-79, 3-96, 3-101, 3-159, 3-173, 4-2, 4-4, 4-5, 4-32, 4-33, 4-35, 4-44, 4-45, 4-46

L

lib, 2-17
link, 2-17
linking loader, 1-5, 2-10, 3-22, 3-24, 3-49, 4-1, 4-2, 4-9, 4-47
lmo, 3-10, 3-105
loadcoff, 3-3, 3-16, 3-86, 3-106

M

make, 2-6, 2-12, 2-17, 4-38
math/graphics, 1-2, 2-12, 2-13
memory management, 3-2, 3-11, 4-2, A-3
mg2tiga utility, 2-13
MODEINFO structure, 3-3, 3-57, 3-64, 3-144, A-11
MONITORINFO structure, A-13

N

num_modes, 3-57, A-3
num_offscrn_areas, 3-58, 3-66, A-12
num_pages, 3-58, 3-108, A-12

O

offscreen, 3-4, 3-18, 3-19, 4-33, A-12, A-14

P

page, 3-4, 3-10, 3-18, 3-58, 3-107, 3-108, 3-144, 4-33, A-12
PAGE structure, A-15
page_busy, 3-10, 3-107
page_flip, 3-10, 3-108, A-15
PALET structure, A-16
palet_gun_depth, 3-6, 3-58, A-11, B-9
palet_inset, 3-58, A-12
palet_size, 3-58, 3-69, A-12
palette, 3-6, 3-18, 3-65, 3-69, 3-70, 3-94, 3-160, 3-161, A-11, A-16, B-9
patnfill_convex, 3-7, 3-8, 3-112, B-2, B-4, B-6
patnfill_oval, 3-7, 3-114, B-2, B-4, B-6
patnfill_piearc, 3-7, 3-115, B-2, B-4, B-6
patnfill_polygon, 3-7, 3-8, 3-118, B-2, B-4, B-6
patnfill_rect, 3-7, 3-120, B-2, B-4, B-6
patnframe_oval, 3-7, 3-121, B-2, B-4
patnframe_rect, 3-7, 3-122, B-2, B-4
patnpen_line, 3-7, 3-123, B-2, B-4, B-8
patnpen_ovalarc, 3-7, 3-126, B-2, B-4, B-8
patnpen_piearc, 3-7, 3-127, B-2, B-4, B-8
patnpen_point, 3-128, B-2, B-4, B-8
patnpen_polyline, 3-7, 3-8, 3-129, B-2, B-4, B-8
pattern, 3-5, 3-6, 3-7, 3-61, 3-112, 3-114, 3-115, 3-118, 3-120, 3-121, 3-122, 3-123, 3-126, 3-127, 3-128, 3-129, 3-139, 3-145, 3-162, 4-33, A-6, A-17, B-2, B-4
PATTERN structure. *See* pattern
peek_breg, 3-10, 3-130
pen, 3-6, 3-7, 3-61, 3-123, 3-126, 3-127, 3-128, 3-129, 3-131, 3-132, 3-133, 3-134, 3-135, 3-145, 3-163, A-6, B-2, B-8
pen_line, 3-7, 3-131, B-2, B-8
pen_ovalarc, 3-7, 3-132, B-2, B-8
pen_piearc, 3-7, 3-133, B-2, B-8
pen_point, 3-7, 3-134, B-2, B-8
pen_polyline, 3-7, 3-8, 3-135, B-2, B-8
pixel array function, 3-8
pixel mask, 3-58, A-11
pixel processing, 3-5, 3-74, 3-76, 3-145, 3-146, 3-165, A-5
plane mask, 3-5, 3-73, 3-145, 3-164
poke_breg, 3-10, 3-136
poly drawing functions, 3-8, 4-16, B-2, B-6, B-8
porting TIGA, D-1
program_mem_end, 3-57, A-3
program_mem_start, 3-57, A-3

R

register usage, 4-33
relocatable load module, RLM, 3-22, 3-101, 4-2, 4-5, 4-38, 4-48, 4-49
  installation, 4-7
rmo, 3-10, 3-137

S

screen_high, 3-58, A-11
screen_wide, 3-58, A-11
seed_fill, 3-7, 3-66, 3-138
seed_patnfill, 3-7, 3-139
select_font, 3-9, 3-140
set_bcolor, 3-5, 3-141
set_clip_rect, 3-5, 3-14, 3-142
set_colors, 3-5, 3-143
set_config, 3-3, 3-144, 4-32, A-3
set_curs_shape, 3-10, 3-90, 3-146, A-5
set_curs_state, 3-10, 3-152
set_curs_xy, 3-10, 3-146, 3-153, A-5
set_draw_origin, 3-5, 3-61, 3-154, A-6, B-5
set_dstbm, 3-8, 3-14, 3-61, 3-155, A-6
set_fcolor, 3-5, 3-158, A-7

set_interrupt, 3-12, 3-159, 4-44, 4-45
set_palet, 3-6, 3-53, 3-69, 3-160, B-9
set_palet_entry, 3-6, 3-53, 3-69, 3-161, B-9
set_patn, 3-5, 3-162, A-17
  See also pattern
set_pensize, 3-5, 3-123, 3-163, A-6
  See also pen
set_pmask, 3-5, 3-73, 3-164
set_ppop, 3-5, 3-74, 3-165
set_srcbm, 3-8, 3-14, 3-61, 3-166, 3-184, A-6
set_textattr, 3-9, 3-167, 3-179, A-10
set_timeout, 3-3, 3-4, 3-168, 4-32
set_transp, 3-5, 3-53, 3-169, 3-181, 3-182
set_vector, 3-11, 3-170
set_videomode, 2-10, 3-3, 3-16, 3-78, 3-145, 3-171, 4-32, 4-38, 4-43
set_windowing, 3-5, 3-79, 3-173
set_wksp, 3-8, 3-44, 3-66, 3-80, 3-90, 3-118, 3-174
share_gsp_addr, 3-57, A-4
share_host_addr, 3-57, A-4
share_mem_size, 3-57, A-4
software developer's kit, SDK, 1-2
software porting kit, SPK, 1-2, 2-4
  subdirectories, 2-5
  system requirements, 2-2
stack_size, 3-57, 3-90, A-4
styled_line, 3-7, 3-61, 3-175, A-6
swap_bm, 3-8, 3-177
symbol table, 3-12, 3-24, 3-49, 3-50, 4-2, 4-48, 4-49
synchronize, 3-3, 3-4, 3-178, 4-32, 4-40
syntax and programming examples, 2-18
sys_flags, 3-20, 3-57, 3-81, A-3
system requirements, 2-2

text, 2-13, 3-9, 3-26, 3-62, 3-75, 3-95, 3-97, 3-140, 3-167, 3-179, 3-180, 4-33, A-7
text_out, 3-9, 3-167, 3-179
text_width, 3-9, 3-167, 3-180
TIGAEXT section, 3-96, 3-101, 4-4, 4-5, 4-11, 4-13, 4-38, 4-39, 4-45, 4-47
TIGAISR section, 4-4, 4-5, 4-44, 4-45, 4-47
TIGALNK, 1-5, 2-10, 3-22, 3-24, 3-49, 4-1, 4-2, 4-9, 4-47
TIGAMODE utility, 2-6, 2-16, 3-3
transp_off, 3-5, 3-181
transp_on, 3-5, 3-182
transparency, 3-5, 3-53, 3-76, 3-145, 3-169, 3-181, 3-182, 4-33
trap vector, 3-11, 3-77, 3-170

utilities, 2-5, 2-12, 2-17, 3-3, 3-10

version_number, 3-57, A-3

wait_scan, 3-10, 3-183
windowing, 3-6, 3-79, 3-142, 3-145, 3-173
wksp_addr, 3-58, 3-80, 3-174, A-12
wksp_pitch, 3-58, 3-80, 3-174, A-12
workspace, 3-8, 3-44, 3-58, 3-66, 3-80, 3-118, 3-174

zoom_rect, 3-8, 3-66, 3-184

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 292-2640.

CALIFORNIA: Irvine (714) 660-1200;
Roseville (916) 786-9208;
San Diego (619) 278-9601;
Santa Clara (408) 980-9000;
Torrance (213) 217-7010;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116;
Ft. Lauderdale (305) 973-8502;
Tampa (813) 885-7411.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-2925.

INDIANA: Carmel (317) 573-6400;
Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Columbia (301) 964-2003.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1569;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600;
Pittsford (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0933;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Beaver Creek (513) 427-6200.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-7655;
Houston (713) 778-6592;
Richardson (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8105;
Santa Clara (408) 748-2220;

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (312) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kierulff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(508) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955;
Hall-Mark (205) 837-8700; Marshall (205) 881-9235;
Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750;
Hall-Mark (602) 437-1200; Marshall (602) 496-0290;
Schweber (602) 431-0030; Wyle (602) 866-2888.

CALIFORNIA: Los Angeles/Orange County:
Arrow/Kierulff (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 773-4500, (714) 669-4100;
Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 880-9686;
(714) 863-0200, (213) 320-8090; Wyle (818) 880-9000,
(714) 863-9953; Zeus (714) 921-9000; (818) 889-3838;
Sacramento: Hall-Mark (916) 624-9781;
Marshall (916) 635-9700; Schweber (916) 364-0222;
Wyle (916) 638-5282;
San Diego: Arrow/Kierulff (619) 565-4800;
Hall-Mark (619) 268-1201; Marshall (619) 578-9600;
Schweber (619) 450-0454; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow/Kierulff (408) 745-6600,
Hall-Mark (408) 432-0900; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444;
Hall-Mark (303) 790-1662; Marshall (303) 451-8383;
Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741;
Hall-Mark (203) 271-2844; Marshall (203) 265-3822;
Schweber (203) 264-4700.

FLORIDA: Ft. Lauderdale:
Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow/Kierulff (407) 323-0252;
Hall-Mark (407) 830-5855; Marshall (407) 767-8585;
Schweber (407) 331-7555; Zeus (407) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399; Schweber (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 449-8252;
Hall-Mark (404) 447-8000; Marshall (404) 923-5750;
Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500;
Hall-Mark (312) 860-3800; Marshall (312) 490-0155;
Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: Indianapolis: Arrow/Kierulff (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483;
Schweber (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230;
Schweber (319) 373-1417.

KANSAS: Kansas City: Arrow/Kierulff (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002;
Hall-Mark (301) 988-9800; Marshall (301) 235-9464;
Schweber (301) 840-5900; Zeus (301) 997-1118.

MASSACHUSETTS Arrow/Kierulff (508) 658-0900;
Hall-Mark (508) 667-0902; Marshall (508) 658-0810;
Schweber (617) 275-5100; Time (617) 532-6200;
Wyle (617) 273-7300; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 462-2290;
Hall-Mark (313) 462-1205; Marshall (313) 525-5850;
Newark (313) 967-0600; Schweber (313) 525-8100;
Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800;
Hall-Mark (612) 941-2600; Marshall (612) 559-2211;
Schweber (612) 941-5280.

MISSOURI: St. Louis: Arrow/Kierulff (314) 567-6888;
Hall-Mark (314) 291-5350; Marshall (314) 291-4650;
Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968;
Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900,
(609) 596-8000; GRS Electronics (609) 964-8560;
Hall-Mark (201) 575-4415, (201) 882-9773,
(609) 235-1900; Marshall (201) 882-0320,
(609) 234-9100; Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: Long Island:
Arrow/Kierulff (516) 231-1009; Hall-Mark (516) 737-0600;
Marshall (516) 273-2424; Schweber (516) 334-7474;
Zeus (914) 937-7400;
Rochester: Arrow/Kierulff (716) 427-0300;
Hall-Mark (716) 425-3300; Marshall (716) 235-7620;
Schweber (716) 424-2222;
Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990;
Hall-Mark (216) 349-4632; Marshall (216) 248-1788;
Schweber (216) 464-2970;
Columbus: Hall-Mark (614) 888-3313;
Dayton: Arrow/Kierulff (513) 435-5563;
Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537;
Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456;
Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000,
(215) 928-1800; GRS Electronics (215) 922-7037;
Marshall (412) 963-0441; Schweber (215) 441-0600,
(412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180;
Hall-Mark (512) 258-8848; Marshall (512) 837-1991;
Schweber (512) 339-0088; Wyle (512) 834-9957;
Dallas: Arrow/Kierulff (214) 380-6464;
Hall-Mark (214) 553-4300; Marshall (214) 233-5200;
Schweber (214) 661-5010; Wyle (214) 235-9953;
Zeus (214) 783-7010;
El Paso: Marshall (915) 593-0706;
Houston: Arrow/Kierulff (713) 530-4700;
Hall-Mark (713) 781-6100; Marshall (713) 895-9200;
Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913;
Hall-Mark (801) 972-1008; Marshall (801) 485-1551;
Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420;
Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150;
Hall-Mark (414) 797-7844; Marshall (414) 797-8400;
Schweber (414) 784-9020.

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 871-7500;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771; Marshall (416) 674-2161;
Vancouver: Arrow Canada (604) 291-2986;
Future (604) 294-1166.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007;**TUCSON:** 818 W. Miracle Mile, Suite 43, Tucson, AZ 85705, (602) 292-2640.

CALIFORNIA: Irvine: 17891 Cartwright Dr., Irvine, CA 92714, (714) 660-1200; **Roseville:** 1 Sierra Gate Plaza, Roseville, CA 95678, (916) 786-9208; **San Diego:** 4333 View Ridge Ave., Suite 100, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Blvd, Altamonte Springs, FL 32701, (305) 260-2116; **Ft. Lauderdale:** 2950 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Rd., Suite 390, Tampa, FL 33634, (813) 885-7411.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Carmel:** 550 Congressional Dr., Carmel, IN 46032, (317) 573-6400.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 201, Cedar Rapids, IA 52402, (319) 395-9550.

KANSAS: Overland Park: 7300 College Blvd., Lighton Plaza, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Dr., Columbia MD 21045, (301) 964-2003.

MASSACHUSETTS: Waltham: 950 Winter St., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1569. **Grand Rapids:** 3075 Orchard Vista Dr. S.E., Grand Rapids, MI 49506, (616) 957-4200.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: St. Louis: 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050.

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Melville:** 1895 Walt Whitman Rd., P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0933; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Beavercreek:** 4200 Colonel Glenn Hwy., Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Pkwy, Blue Bell, PA 19422, (215) 825-9500

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00918, (809) 753-8700.

TENNESSEE: Johnson City: Erwin Hwy, P.O. Drawer 1255, Johnson City, TN 37605 (615) 461-2192.

TEXAS: Austin: 12501 Research Blvd., Austin, TX 78759, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75081, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 250, Houston, TX 77074, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green St., Suite 200, Murray, UT 84123, (801) 266-8972.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 782-2899.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 336-1860.

ARGENTINA: Texas Instruments Argentina Viamonte 1119, 1053 Capital Federal, Buenos Aires, Argentina, 541/748-3699

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: 11, Avenue Jules Bondetlaan 11, 1140 Brussels, Belgium, (02) 242-3080.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, 2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Ahertajantie 3, P.O. Box 81, ESP00, Finland, (90) 0-461-422.

FRANCE: Texas Instruments France: Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay cedex (1) 30 70 1003.

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, 8050 Freising, 8161 + 80-4591; Kurfuerstendamm 195/196, 1000 Berlin 15, 30 + 882-7365; III, Hagen 43/Kibbelstrasse, .19, 4300 Essen, 201-24250; Kirchhorsterstrasse 2, 3000 Hannover 51, 511 + 648021; Maybachstrabe 11, 7302 Ostfildern 2-Nelingen, 711 + 34030.

TEXAS INSTRUMENTS

HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Ctr., 7 Canton Rd., Kowloon, Hong Kong, (852) 3-7351223.

IRELAND: Texas Instruments (Ireland) Limited: 7/8 Harcourt Street, Stillorgen, County Dublin, Eire, 1 781677.

ITALY: Texas Instruments Italia S.p.A. Divisione Semiconduttori: Viale Europa, 40, 20093 Cologne Monzese (Milano), (02) 253001; Via Castello della Magliana, 38, 00148 Roma, (06) 5222651; Via Amendola, 17, 40100 Bologna, (051) 554004.

JAPAN: Tokyo Marketing/Sales (Headquarters): Texas Instruments Japan Ltd., MS Shibaura Bldg., 9F, 4-13-23 Shibaura, Minato-ku, Tokyo 108, Japan, 03-769-8700. Texas Instruments Japan Ltd.: Nissho-Iwai Bldg. 5F, 30 Imabashi 3-chome, Higashi-ku, Osaka 541, Japan, 06-294-1881; Daini Toyota West Bldg. 7F, 10-27 Meieki 4-chome, Nakamura-ku, Nagoya 450, 052-583-8691; Daiichi Seimei Bldg. 6F, 3-10 Oyama-cho, Kanazawa 920, Ishikawa-ken, 0762-23-5471; Daiichi Olympic Tachikawa Bldg. 6F, 1-26-12 Akebono-cho, Tachikawa 190, Tokyo, 0425-27-6426; Matsumoto Showa Bldg. 6F, 2-11 Fukashi 1-chome, Matsumoto 390, Nagano-ken, 0263-33-1060; Yokohama Nishiguchi KN Bldg. 6F, 2-8-4 Kita-Saiwai-cho, Nishi-ku, Yokohama 220, 045-322-6741; Nihon Seimei Kyoto Yasaka Bldg. 5F, 843-2 Higashi Shiokohjidori, Nishinotoh-in Higashi-iru, Shiokouji, Shimogyo-ku, Kyoto 600, 075-341-7713; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki 873, Oita-ken, 09786-3-3211; Miho Plant, 2350 Kihara Miho-mura, Inashiki-gun 300-04, Ibaragi-ken, 0298-85-2541.

KOREA: Texas Instruments Korea Ltd., 28th Fl., Trade Tower, #159, Samsung-Dong, Kangnam-ku, Seoul, Korea 2 + 551-2810.

MEXICO: Texas Instruments de Mexico S.A.: Alfonso Reyes—115, Col. Hipodromo Condesa, Mexico, D.F., Mexico 06120, 525/525-3860.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 + 274681.

NETHERLANDS: Texas Instruments Holland B.V., 19 Hogehilweg, 1100 AZ Amsterdam—Zuidoost, Holland 20 + 5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 0585, Oslo 5, Norway, (2) 155090.

PEOPLES REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 Citic Bldg., 19 Jianguomenwai Dajje, Beijing, China, (861) 5002255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., Paseo de Roxas, Makati, Metro Manila, Philippines, 817-60-31.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific Division, 101 Thompson Rd. #23-01, United Square, Singapore 1130, 350-8100.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 28036, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): S-164-93, Stockholm, Sweden, 8 - 752-5800.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co., 9th Floor Bank Tower, 205 Tun Hwa N. Rd., Taipei, Taiwan, Republic of China, 2 + 713-9311.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 270111.

Printed in U.S.A.
2558570-9707

SPVU015A