COMPUTE!'s
Technical Reference Guide

ATARI ST

VOLUME THREE

TOS

Sheldon Leemon

Your passport to TOS, the input/output and fast graphics operating systems underlying GEM. Includes fully commented C and machine language programming examples. For the intermediate-to-advanced-level Atari ST programmer.

COMPUTE! Books

COMPUTE!'s

Technical Reference Guide ATARI ST VOLUME THREE

TOS

Sheldon Leemon

COMPUTE! Books
Greensboro, North Carolina
Radnor, Pennsylvania

Editor: Robert Bixby

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data
(Revised for volume 3)
Leemon, Sheldon.
  Compute!'s technical reference guide, Atari ST.
  Includes indexes.
  Contents: v. 1. VDI — v. 2. GEM, AES — v. 3. TOS.
  1. Atari ST computers. I. Title. II. Title:
Atari ST.
QA76.8.A824L44 1987 004.165 87-401655
ISBN 0-87455-149-8 (pbk. : v. 3)

Contents

Foreword

COMPUTE!'s Technical Reference Guide, Atari ST—Volume Three: TOS is the third information-packed ST book from noted Atari ST author Sheldon Leemon. Inside you'll find complete information on the Basic Input/Output System (BIOS), the GEM Disk Operating System (GEMDOS), eXtended Basic Input/Output System (XBIOS), and Low-level (line A) graphics, as well as an exhaustive reference section explaining each BIOS function, program examples in C and machine language, and a complete memory map.

You may recall with what fanfare the ST was introduced. It was widely called the "Jackintosh," a nickname combining the name of the *Macintosh* (they have similar user interfaces) with *Jack*, for Jack Trammiel, its creator. Despite its promise, early applications for the machine were simply games and programs ported over from lesser machines.

Over the years, however, programmers have come to recognize that the ST is not only lightning fast and highly sophisticated, but it is also programmable on several tiers of its operating system.

At the heart of the ST is the same famed Motorola 68000 microprocessor that drives the Macintosh, and the Amiga. Over the chip is BIOS, XBIOS, and TOS (collectively known as TOS, or the Trammiel Operating System). Above these are the AES and the VDI, the high-level interfaces that make up the GEM interface from Digitial Research. The AES and VDI were the subjects of the first two books in this series. Most programmers prefer to program through GEM. It's fast and friendly. With a minimum of fuss, it provides the maximum of features, such as reading the mouse position and providing menus and other services.

But faster still, and not lagging far behind in ease of use, is TOS. You can use it to provide your own creative uses of graphics, printer functions, and the disk operating system, just as GEM does, but with true 68000 speed and with no intermediary levels of operating systems.

If you're a serious ST programmer, you probably already own highlighted, dogeared copies of the first two books in this series. This book will complete your collection and lead you into hitherto unsuspected levels of programming. If you are only beginning to explore the world inside your ST, this book is an excellent starting point.

Chapter 1

The ST Operating System: An Overview

A computer's operating system is an organized collection of small built-in programs that enables the computer to communicate with external devices, such as the keyboard, display screen, and disk drive, and to perform fundamental tasks like loading and running an application program. While most people regard GEM as the ST's operating system, it's just a friendly user interface on top of a more conventional operating system to make the computer easier to program and to operate. While GEM provides higher level functions like support for drop-down menus, dialogs, and icons, it still must rely on a set of low-level operating system routines for tasks such as reading a file from the disk drive. It is this set of low-level routines that shall be referred to as TOS.

TOS Organization

The ST operating system is contained in a set of TOS ROM (Read-Only Memory) chips that contain a total of 192 kilobytes of program code and data. The name *TOS* may have at one time stood for *The Operating System,* but is now more commonly thought of as an acronym for *Tramiel Operating System,* named after the Tramiel family that now owns Atari. The TOS ROMs contain all of the ST's system software. This includes:

GEM	This software provides ST applications with a consistent user interface, featuring drop-down menus, multiple windows, icons, and dialog boxes. GEM is divided into the VDI (which provides low-level graphics calls), AES (which provides user services like menus), and the GEM Desktop pro-

	gram, (which provides the desktop metaphor for working with the disk filing system). The VDI and GEM AES were subjects of earlier books in this series by the same author. These books are also available from COMPUTE! Books.
BIOS	The Basic Input/Output System (BIOS) is a collection of low-level I/O routines that are not necessarily specific to the ST hardware. They include routines to communicate with character-oriented devices like the keyboard, screen, and printer, and to communicate with disk drives on the sector level. They also include routines to check which disk drives are available, if a disk has been removed from the drive, and so on.
XBIOS	XBIOS, the eXtended Basic Input/Output System is a set of hardware-specific I/O-related routines. There are routines for finding and changing the address of screen display memory, for setting the hardware color registers, for waiting for the vertical blanking interval, and for accessing the sound chip. There are also routines for communicating with the 68901 Multi-Function Peripheral (MFP) chip.
GEMDOS (or BDOS)	This is a set of functions used to implement the higher level disk filing system, which closely follows the model of MS-DOS. These routines allow the user to access the disk device on the file level, rather than directly reading specific physical sectors on the disk. They allow the user to perform functions like reading the disk directory, creating or deleting a subdirectory, deleting a file, renaming a file, and so on. The GEMDOS also contains miscellaneous routines for

	communicating with the character devices like the screen, keyboard, printer, and serial port.
Line A routines	Line A routines are the low-level graphics routines the GEM VDI calls for basic graphics functions. These functions include setting and reading individual pixels, drawing lines and filled polygons, and moving *software sprites,* like the mouse pointer. Since the ST screen is bitmapped, the line A routines are also used for drawing all text characters on the screen. Using the line A routines for graphics and text provides greater compatibility than accessing the ST graphics hardware directly, because such programs will continue to function correctly even if the ST graphics hardware changes. For example, programs that use the line A routines can take advantage of the blitter chip used by later ST models, while programs that write to screen memory directly cannot.
Exception Handlers	Many of the Operating System routines are invoked by interrupts and trap instructions, which in 68000 parlance are referred to as *exceptions.* For example, the BIOS routines are called via the TRAP #13 instruction, GEMDOS routines via the TRAP #1 instruction, and GEM AES and VDI routines are called via the TRAP #2 instruction. In addition to the handlers that route these calls, there are a number of lower-level interrupt handlers, such as the vertical blank interrupt handler, which are of interest to programmers.
Startup Code	The startup code is a short piece of program code that is called when the

computer is first turned on, or the reset button is pushed. It checks for ROM cartridges, configures the I/O ports and the screen, tests memory size, sets the exception handler vectors, executes the programs in the AUTO folder, and jumps to the GEM Desktop program.

Since all of the GEM code is included in the ROMs, it will be considered as part of TOS. For purposes of this book, however, TOS will be considered to be everything in the ROMs except GEM.

Calling TOS from Machine Language

The various TOS routines are called via the exception vectors. When programming in machine language, the general procedure is to push the function number and other function parameters on the stack, issue a TRAP instruction, and then remove the parameters from the stack. The specific TRAP instruction depends on the type of TOS routine you're calling. The BIOS routines are called with a TRAP #13 instruction, the XBIOS with a TRAP #14, and the GEMDOS routines with TRAP #1. A call to the BIOS routine Bconstat( ) would look like this:

```
move.w   #2,-(sp)   * push device number for console device
move.w   #1,-(sp)   * push function number for Bconstat
trap     #13        * call BIOS
addq.l   #4,sp      * pull the parameters off the stack
```

Note that if you plan to use GEM calls in your machine language application, you'll need to do some preparatory work at the beginning of the program. When GEMDOS starts an application program (but not a desk accessory), it allocates all of the system memory to that program. Therefore, if a program uses the system memory-management calls, or any of the GEM AES calls that themselves allocate memory, or runs another program using the Pexec( ) function, at startup time it must deallocate all of the memory it isn't actually using. This is done using the GEMDOS Mshrink( ) function. Complete details and some sample code can be found in Chapter 5 in the section dealing with

Mshrink( ). For now, it is sufficient to know that this step is necessary for programs using GEM or memory-management calls, and not for programs that only use TOS function calls.

Calling the TOS Routines from C

It's much simpler to call the TOS routines from C than from machine language, since most C compilers for the ST include library routines for the BIOS, XBIOS, and GEMDOS calls. These library routines make calling TOS routines exactly like calling any other kind of C routines. For example, the library call bios( ) issues the TRAP #13 command after the parameters have been pushed on the stack. To call the BIOS routine Bconin( ) to get a character from the console device (device 2), you need only use the statement:

```
bios(1,2);
```

Many C compilers include a header file called OSBIND.H. This header file contains C macro definitions for the various BIOS, XBIOS, and GEMDOS commands. For example, the macro Bconstat(a) is defined as follows:

```
#define Bconstat(a) bios(1,a)
```

Therefore, if you've #included the OSBIND.H file in your program, you could replace the bios(1,2); statement with:

```
Bconstat(2);
```

Since this is more readable than the bios( ) call, the macro format will be used wherever possible. Just remember that in order for the compiler and linker to understand these macros, you must use the #include directive to add the OSBIND.H file first.

C programmers usually don't have to worry about releasing extra memory with the Mshrink( ) command, or setting the program stack, since this work is done for them by the compiler's own startup code. This code is found in the GEMSTART.O or APPSTART.O module linked in by *Alcyon* C programmers, and in the INIT.O module of the SYSLIB library of *Megamax C*. You should note, however, that in some extreme cases, you may have to recompile the startup module to give back more or less memory than the default module. Again, more details about the Mshrink( ) function are provided in Chapter 5.

About the Examples

Because it's easy to make GEM calls from C, and because the language produces programs that are relatively small in size and quick in execution for a high-level language, it has become the language of choice for software development on the ST. For this reason, most of the examples in this book will be written in C. On occasion, however, machine language examples will be included as well, to show how the C examples may be translated to that environment. The macro names for the C functions will be used here as they appear in the official Digital Research GEM header files, since they have been adopted by the manufacturers of other C compilers as well.

The C programs in this book are designed to work specifically with the *Alcyon* C compiler, the compiler officially supported by Atari, and with *Megamax C*, which also provides a very complete development environment. For these compilers, the int data type refers to a 16-bit word of data. Some other compilers, such as the *Lattice C* compiler, use a 32-bit integer as the default data type. When compiling the programs in this book with such compilers, substitute *short* for each reference to *int*, and keep in mind that the default size for function returns and constants may be 32 bits instead of 16.

For the sake of simplicity, the portability macros such as WORD were not used. These macros use the C preprocessor to define a 16-bit data type that will be valid for any compiler. The reader is free to use the macros if they are seen as more convenient.

The machine language examples were all created with the assembler included in the Atari development package, but they should be so generic as to assemble unchanged with almost any good 68000 assembler.

Chapter 2

BIOS

The lowest-level ST Input/Output routines are in the section of the operation system known as the BIOS (Basic Input/Output System). The BIOS contains three basic types of I/O routines.

The first group of I/O routines contains routines for communication with character-oriented I/O devices like the printer, the screen, the serial port, and MIDI port. The second group contains the basic functions used to communicate with the disk drive at the hardware level. These allow you determine how many drives are connected, whether a disk has been changed in a drive, and where to find the BIOS parameter block for a drive, which gives information about the drive configuration. They also let you read or write to the disk at the sector level, which is a lower level of organization than the normal disk filing system. Finally, the BIOS contains some miscellaneous routines that perform various system functions, such as reading or setting the exception vectors and returning information about the memory management system and the precision level of the system clock.

The ST BIOS routines can be called from user mode, and are reentrant to three levels. They use registers A0–A2 and D0–D2 as scratch registers, which means if you're programming in machine language and using these registers to store important information, you must save their contents before making a BIOS call and restore them after the BIOS call. Each of the BIOS routines has a command number associated with it. It may also be associated with command parameters that specify more precisely what the function should do.

For example, the BIOS function to output a character to a device is command number 3. It requires two command parameters: One tells the function which character to print and the other specifies the output device to use.

To call a BIOS function from machine language, you must push the command parameters onto the stack, followed by the command number, and execute a TRAP #13 statement. The TRAP #13 instruction puts the program into supervisor mode and begins executing the instructions found at the address stored in exception vector 45, whose address is 180 ($B4). This exception vector contains the address of the BIOS handler, which reads the command number on the top of the stack and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register D0. When a BIOS function call is completed, the calling program is responsible for adjusting the stack to remove the command parameters and command number. You should note that the BIOS changes the command number and return address on the stack.

The following program fragment demonstrates sending the character X to the console device using BIOS command number 3:

```
move.w   #'X',-(sp)   * push character value on stack
move.w   #2,-(sp)     * push console device number on stack
move.w   #3,-(sp)     * push BIOS command number on stack
trap     #13          * call BIOS handler
addq.l   #6,sp        * pop parameters (6 bytes) off stack
```

Calling the BIOS routines from C is much simpler. Most C compilers come with a library routine called bios( ), which stacks the parameters and executes the TRAP #13 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement

```
bios(3,2,'X');
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OSBIND.H which defines macros for all of the BIOS functions. For example, the macro definition for BIOS command 3 is

```
#define Bconout(a,b)   bios(3,a,b)
```

Therefore, after you #include OSBINDS.H in your program, call your sample function like this:

```
Bconout(2,'X');
```

This is a more readable than the other version. For this reason, the macros will be used in the discussions of BIOS routines and sample programs. To use BIOS functions in your C programs, you must #include OSBIND.H if you use the macros, and you must link your program with the library that contains the bios( ) function.

Character Device I/O

This group of functions enables communication with certain I/O devices at the character level. These character devices are sequential in nature, which means they transfer information as a stream of characters sent one after the other, and the receiver has no control over the order in which the information is sent. Devices such as the printer, the serial port, and the keyboard differ in this regard from storage devices like disk drives, which allow random access to information stored at a particular location within named files. The BIOS I/O functions only allow you to send or receive a single character at a time. There are, however, XBIOS functions that allow you to send a string of characters to the MIDI or Intelligent Keyboard device with a single call.

The five character devices on the ST are shown in Table 2-1.

The first three of these devices can be found on most computers, while the last two are specific to the ST. Three of the character devices can both send and receive information, but the printer and intelligent keyboard devices can only receive output (at least through the BIOS routines).

Do not confuse the intelligent keyboard device with the console device. The console device consists of two physically separate devices: the display screen and the keyboard. This device receives ASCII characters from the keyboard and dis-

Table 2-1. The Five Character Devices

Device Number	Device Name	Description
0	PRN:	Parallel (Centronics) printer (output only)
1	AUX:	RS-232 serial device (modem)
2	CON:	The console device (keyboard and screen)
3		MIDI (Musical Instrument Digital Interface)
4		IKBD, the intelligent keyboard device (output only)

plays them on the monitor. The intelligent keyboard device, on the other hand, permits communication with the ST keyboard's own 8-bit microprocessor. This keyboard processor controls the keyboard, mouse, joysticks, and time-of-day clock. Since this device is complex and has a large set of commands, it is treated separately in Appendix I.

The most basic input function is to wait for a single character to be transferred from one of the devices. The BIOS call that implements this function is called Bconin( ). If a character is available from the input device when you call this function, it will receive that character and return immediately. If there is no character available at the time you call the function, it won't return until the device has sent a character. The C language macro defined for this function in the OSBIND.H file uses the following syntax:

```
int devnum;
long char;
  char = Bconin(devnum);
```

where *devnum* is the number of the device from which to receive a character. Only numbers 1–3 are valid, since the printer can't be used for input and the intelligent keyboard doesn't return any information through its character device. The character received from the device is returned in the low byte of the variable *char*. Note that the Bconin( ) function returns an entire longword, instead of a single character. Only the least significant byte of this longword is used for information received from the MIDI or serial device. The console device, however, uses both words of the longword.

The ASCII code for the character that was received is returned in the least significant byte of the low word. The least significant byte of the high word contains a special key code that indicates the physical key that was struck. This allows the program to differentiate between the number 1 on the top row of the keyboard and the 1 on the numeric keypad. Together, the ASCII value and the key code are known as the scan code. This scan code is more commonly expressed as a single word, with the keycode in the high byte and the ASCII code in the low byte. Appendix J contains a complete list of scan codes expressed in this format. To convert from the longword *char* to the word *scancode*, use the following C statement:

```
scancode = (int)((char>>8) | char);
```

This shifts the keycode into the high byte of the low word, and discards the high word.

Although the scan code tells what key was pressed and if it was pressed in combination with one of the shift keys, it still can't give you complete information about the Shift, Alternate, and Control keys. For example, if you press the Shift, Alternate, and *A* keys together, the scan code you receive is the same one you would get from pressing Alternate and *A*. The Control-cursor up combination yields the same value as the cursor up key alone. To get complete information about the status of the shift keys, you must use another BIOS routine called Kbshift. The C language macro for this function takes the following form:

```
int shiftcode, mode;
  shiftcode = Kbshift(mode);
```

where *mode* is a flag that indicates whether you wish to read or to set the shift key status code. A nonnegative number in *mode* sets the status code to the value indicated after reading the current code value. A negative number in *mode* causes the function to return the shift key status code in *shiftcode*. Each of the eight least significant bits in *shiftcode* represents a particular shift key. If a bit is set to 1, it indicates that the corresponding key was pressed at the time the function call was made. Otherwise, the key was not pressed. The bit assignments for the *shiftcode* flag are shown in Table 2-2.

Since *shiftcode* is a bit flag, any or all of shift keys can be detected at once. For example, a *shiftcode* value of 14 (8 + 4 + 2) would mean that the Alternate, Control, and left Shift keys were all pressed at the same time.

Table 2-2. Bit Assignments for the shiftcode Flag

Bit Number	Bit Value	Shift Key
0	1	Right shift key
1	2	Left shift key
2	4	Control key
3	8	Alternate key
4	16	Caps lock on
5	32	Alternate-Clr/Home key combination (Keyboard equivalent for right mouse button)
6	64	Alternate-Insert key combination (Keyboard equivalent for left mouse button)
7	128	Reserved (currently zero)

There's one problem with relying on Kbshift( ) to supply the information that Bconin( ) omits: The console device saves incoming keystrokes in a memory buffer, which means that it's possible that a character received via Bconin( ) may actually be the result of a keypress that took place some time ago. Therefore, testing the shift keys at the time the character is read may not tell you what their status was at the time the user entered the character.

If your program is reading the console device frequently, this will probably not present a real problem since a large number of program statements can execute in the time it takes the user to remove his fingers from the keyboard. If there is a significant time delay between reads, such that Kbshift may not reflect the actual shift key status, you may wish to change the system variable *conterm,* a byte value which is stored at location 1156 ($484). If you set bit 3 of this variable to 1, the BIOS Bconin( ) function will return the keyboard shift status code in the most significant byte of the *scancode*. This eliminates calling Kbshift separately. Since *conterm* is in protected memory, shift into supervisor mode before changing this value.

Shifting between supervisor and user modes will be discussed in Chapters 2 and 5.

Having the keyboard shift codes in the high byte of the *scancode* seems so handy that you may wonder why it isn't the default state of affairs. The reason is compatibility. The keycodes the ST uses are based on those used by the IBM PC, so the format in which scan codes are returned is also based on the PC format. Programs that rely on the fact that the ST console keyboard system mirrors that of the PC may not handle the scan codes correctly if there is unexpected data in the high byte. Therefore, if you set the *conterm* bit in your program, remember to set it back when your program ends, so other programs will not receive unexpected data in the high byte.

One of the problems with using Bconin( ) is that if no character is available from the device; the function waits until one is available. This leaves your program stuck until input is received. If your program doesn't receive any input, it remains stuck forever, forcing the user to turn off the computer to regain control. To prevent this situation, the BIOS includes a function that lets you determine whether a character is waiting to be received. This function is called Bconstat, and its C macro uses this syntax:

```
int devnum;
long status;
  status = Bconstat(devnum);
```

where *devnum* is the number of the input device whose status you wish to check. Since only devices 1–3 provide input, status checks should be limited to those devices. The value returned in *status* is a 0 if there are no characters waiting, and $FFFF (−1) if there is at least one character ready to be received. Thus, by calling Bconstat it's possible to determine whether Bconin will return immediately. If the call to Bconstat shows there are no characters ready, your program may omit the call to Bconin, go on to do something else, and then check the input device again later.

The BIOS output functions are very similar to the input functions. To output a character to one of the devices, use the function Bconout( ), whose C macro takes the following form:

```
int devnum, char;
  Bconout(devnum,char);
```

where *devnum* is the number of the device (0–4) to which the character is sent. The variable *char* contains the ASCII value of the character to send in its low byte. Note that like Bconin, Bconout doesn't return until the character is actually sent. To avoid sending a character to a device that isn't ready to receive it, and thus hanging up your program, first test the status of the output device with the Bcostat function. The C macro for this function takes the following form:

```
int devnum;
long status;
  status = Bcostat(devnum);
```

where *devnum* is the number (0–4) of the device to query. Bcostat returns a 0 in *status* if the device is not ready to accept a character, and $FFFF (−1) if it is ready. It always makes sense to check the output device to see if it's ready to receive characters, particularly before you send the first one.

Unlike GEM graphics text functions, which output any character for which there is image data, the console device screen emulates a DEC VT-52 display terminal and treats the ASCII characters from 0 to 31 as nonprinting control characters. This means, for example, that it interprets the ASCII

Table 2-3. VT-52 Codes to Which Bconout( ) Responds

Code	Action
Esc A	Cursor Up
Esc B	Cursor Down
Esc C	Cursor Right
Esc D	Cursor Left
Esc E	Clear Screen and Home Cursor
Esc H	Home Cursor
Esc I	Cursor Up (scrolls screen down if at top line)
Esc J	Clear to End of Screen
Esc K	Clear to End of Line
Esc L	Insert Line
Esc M	Delete Line
Esc Y (row+32) (column+32)	Position Cursor at Row, Col (starts with 0)
Esc b (register)	Select Foreground (Character) Color
Esc c (register)	Select Background Color
Esc d	Clear to Beginning of Screen
Esc e	Cursor On
Esc f	Cursor Off
Esc j	Save Cursor Position
Esc k	Move Cursor to Saved Position
Esc l	Clear line
Esc o	Clear from Beginning of Line
Esc p	Reverse Video On
Esc q	Reverse Video Off
Esc v	Line Wrap On
Esc w	Line Wrap Off

character 13 as a carriage return, an instruction to move the cursor to the beginning of the line, rather than as a character that should be printed. There are a number of VT-52 *escape codes* to which the console device responds. These escape sequences are strings of characters beginning with the ASCII character 27 (Esc), followed by one or more text characters. The VT-52 codes to which Bconout( ) responds are shown in Table 2-3.

In addition to the Escape codes, the ST terminal emulation also responds to the follow ASCII control codes:

ASCII Control Code	Action
07	Bell
08	Backspace
09	Tab
10–12	Line feed
13	Carriage Return

Though the console device will not print the nonprinting ASCII characters (those whose values are below 32), it will print the ST's extended ASCII set (characters whose value is above 128). These include a number of Greek and Hebrew characters as well as some math symbols. For a complete table of the system characters, see Appendix G. For more detailed information on the VT-52 Escape sequences, see Appendix E.

The ST buffers character device I/O. This means that incoming information from a character device and outgoing information to a character device is first stored in a reserved memory area before being read by your program or sent to the external device. This is done so that if the device is sending information faster than your program is reading it, or if your program is sending information faster than the device can read it, none of the information will be lost. Though the default buffers are sufficient for most purposes, they may not be large enough to prevent data loss when transferring a lot of information at very high speeds through the MIDI or serial ports. In those cases, you may need to substitute your own, larger buffer areas by using the XBIOS Iorec( ) routine (see Chapter 3).

The following C language sample program BCHARDEV.C demonstrates the use of some of the BIOS character device routines. It uses the Bconstat call to monitor the console device (keyboard) input status and prints a dot every so often if no key is struck, using Bconout. When a key is pressed, the program reads it with Bconin and Kbshift and prints the ASCII character, the ASCII code, and the shift and key scan codes. The program ends when the unshifted *q* is struck.

Program 2-1. BCHARDEV.C

```
/************************************************/
/*                                              */
/*  BCHARDEV.C                                  */
/*                                              */
/*  Demonstrates some of the BIOS               */
/*  character device functions                  */
/*                                              */
/************************************************/

#include <osbind.h>    /* For BIOS macro definitions */
#define CON   2        /* alias for Console device number */

main()
{
   long ch;
   int sh, count = 4999;

   while((char)ch != 'q') /* End program when 'q' is struck */
   {
   while(!Bconstat(CON))    /* until then, wait for key */
     {
     if ( (count++) == 5000)  /* print a dot every so often */
       {
       Bconout(CON,'.');
       count=0;
       }
     }                     /* When key is struck, */
   ch = Bconin(CON);       /* get the key value */
   sh = Kbshift(-1);       /* and shift status code. */
            /* Print the ASCII character and value, */
   printf ("  \n%c%6x",(char)ch,(char)ch[illegible]);
            /* shift status and key scan codes */
   printf("  %4x%8x\n",sh, (int)((ch>>8)|ch)  );
   }
}

/******** end of BCHARDEV.C *****/
```

Machine language programmers should refer to the XSCREEN.X program in Chapter 4 for examples on using the BIOS functions Bconin( ) and Bconout( ) in machine language.

Disk Device I/O

The ST BIOS contains four disk I/O routines, three of which merely return information about the drives. These routines are included mainly for use by other, higher-level operating system routines and may not be of much use to the average programmer. The first function, Drvmap( ), can be used to determine which drives are available. The syntax for this call, using the C macro, is

long drives;
  drives = Drvmap( );

where *drives* is a bitflag that indicates which drives are connected. Each bit of the *drives* variable corresponds to a different drive. Bit 0 is assigned to drive A, bit 1 to drive B, and so on up to bit 15, which corresponds to drive P (the current version of the ST operating system only recognizes 16 drives). If the bit that corresponds to a drive is set to 1, that drive is connected, otherwise, it is unavailable. The value returned by Drvmap is the same one stored in the system variable *_drvbits*, at memory location 1220 ($4C4). Note that if even one floppy is connected, both bits 0 and 1 are always set to 1. If drive A: is connected, the system will also use it as a logical drive B:, if no physical drive B: is present.

Once you've found which drives are connected, you can find more information about the layout of a particular disk by reading the BIOS Parameter Block in its boot sector. Getbpb( ) is the function used to find the address of the Parameter Block and it's called like this:

```
int drivenum;
long blockaddr;
  blockaddr = Getbpb(drivenum);
```

where *drivenum* is the number of the drive whose BIOS Parameter Block you wish to find (0 = drive A:, 1 = drive B:, and so on). The starting address of the BIOS Parameter Block is returned in *blockaddr*. The data structure pointed to by *blockaddr* consists of nine words of data. The structure elements are as shown in Table 2-4.

Table 2-4. Parameter Block Structure Elements

Element Number	Name	Description
0	recsiz	Number of bytes per sector (must be 512 under current GEMDOS)
1	clsiz	Number of sectors per cluster (must be two under current GEMDOS)
2	clsizb	Number of bytes per cluster (must be 1024 under current GEMDOS)
3	rdlen	Root directory length (in sectors)
4	fsiz	File Allocation Table (FAT) size (in sectors)
5	fatrec	Sector number of the start of second FAT
6	datrec	Sector number of the first data cluster
7	numcl	Number of data clusters on the disk
8	bflags	Bit flags*

* Currently only bit 0 is used. When set, it indicates 16-bit FAT entries instead of the usual 12-bit entries.

This information tells you how much storage space is on the disk and how it's allocated. GEMDOS performs a Getbpb( ) operation when it first accesses a drive, or when it accesses a drive after a media change. For more information on disk organization and the file system, see Chapter 6.

Perhaps the most important of the BIOS disk functions allows you to read or write disk sectors. This function is used mainly by the disk operating system. Since it operates at a lower level than the filing system, you probably won't use it unless you're writing a disk sector editor, or something equally exotic. The macro for this function is called Rwabs (for Read Write ABSolute), and it's called as follows:

```
int mode, sectors, start, drivenum;
long buffer, status;
  status = Rwabs(mode, buffer, sectors, start, drivenum);
```

Mode is a flag that indicates whether you wish to read or write sectors to the disk. The valid mode numbers are

Mode Number	Description
0	Read sectors
1	Write sectors
2	Read sectors without affecting media change status
3	Write sectors without affecting media change status

Buffer is a pointer to the memory area where the transferred data is stored. The size of this area depends on the number of sectors to be transferred, which is stored in the variable named *sectors*. Allocate 512 bytes for each sector.

Since the data transfer will proceed very slowly if the buffer area starts at an odd address, you should ensure that it starts at an even address by declaring the buffer variable as an array of words (ints). The start variable is used to indicate the starting sector from which to read or to which to write. The *drivenum* parameter specifies the disk drive to use (0 = drive A:, 1 = drive B:, and so on). When the function call concludes, the status variable will be 0 if the operation was successful. A negative number indicates an error has occurred. See Appendix D for the list of GEMDOS error messages.

The last of the BIOS disk functions is used by the disk operating system to determine whether a disk has been changed. Its name is Mediach( ), and its C macro call uses the following syntax:

```
int drivenum;
long status;
  status = Mediach(drivenum);
```

where once again the *drivenum* parameter specifies the disk drive to check (0 = drive A:, 1 = drive B:, and so on). The status returned by this call can be one of three values. A zero value means the media definitely has not changed, while a value of 2 means that it definitely has not changed. A status value of 1 means that the media might have changed, but the BIOS can't give a more definite answer until a read operation is performed.

System Functions

The last three BIOS functions are miscellaneous system calls. The first, Getmpb( ) (Get Memory Parameter Block), is used by GEMDOS to initialize the memory management system. The format for this call is

```
long mpbptr;
  Getmpb(mpbptr);
```

where *mpbptr* is a pointer to the starting address of a Memory Parameter Block. The definition for this data structure is as follows:

```
struct mpb
 {
 struct md *mp_mfl;      /* ptr to memory free list */
 struct md *mp_mal;      /* memory allocated list /*
 struct md *mp_rover;    /* roving pointer */
 }
```

As you can see, this structure consists of three pointers to other data structures, called *memory descriptor structures.* These contain information about blocks of memory, primarily the starting address of the memory block and its size:

```
struct      md
{
struct md   *m_link;    /* pointer to next MD [NULL]
long        m_start;    /* start addr of mem block */
long        m_length;   /* size of mem block (bytes) */
struct pd   *m_own;     /* ptr to MD owner's process-
                           descriptor [NULL] */
};
```

The Getmpb( ) function fills the designated Memory Parameter Block with initial system values. At the beginning, the memory allocated list pointer is 0, since no memory blocks have been allocated by programs yet. The memory free list pointer contains the address of a memory descriptor that specifies the entire Transient Program Area from system variables membot to memtop. For more information on these system variables, see Appendix K, memory locations 1074 ($432) and 1078 ($436).

The next system call reads or sets exception vectors. Exception vectors are a collection of addresses stored in low memory reserved for the use of the 68000 processor or the Operating System. Whenever an exception occurs, the processor goes into supervisor mode and program execution is diverted through the appropriate vector. For example, when a program tries to execute an illegal instruction, an exception occurs and program execution resumes at the address pointed to by exception vector 4, which starts at memory location 16.

The 68000 processor supports many kinds of exceptions, including interrupts, TRAP instructions, and several kinds of error conditions. One error of particular interest is called a *bus error*. On the ST, a bus error occurs when a program tries to access memory locations below 2048 ($800), or the hardware registers above $FF8000 from user mode. That's why this call is necessary: It allows you to change the exception vectors located in protected memory without switching into supervisor mode. The format for this call is

```
int vecnum;
long vecaddr, oldaddr;
  oldaddr = Setexec(vecnum, vecaddr);
```

where *vecnum* is the vector number to read or change, and *vecaddr* is the new address to be stored in that vector. If *vecaddr* contains a −1 (0xFFFF), it indicates that you want only to read the current address stored in the vector, not change it. In either case, the address stored in the vector before the call was made is returned in the variable *oldaddr*. Your program should always save this vector and restore it before terminating.

For more information on the exception vectors, see the memory map in Appendix K, locations 0–1036 ($0–$40C).

The final BIOS system call pertains to the system timer interrupt. This is a system interrupt routine that is called periodically to update the GEMDOS date and time. The Tickcal( ) routine returns the number of milliseconds between timer ticks. For the ST, this value is 20 milliseconds, since the timer interrupt updates the system clock at the rate of fifty times per second (even though the interrupt is actually called 200 times per second). The format for Tickcal( ) is

```
long ticklen;
  ticklen = Tickcal( );
```

where *ticklen* is the length of time that passes between timer ticks, in milliseconds. This call is unnecessary because the number of milliseconds since the last timer interrupt is passed on the stack when the timer interrupt handler is called. This value is also stored in the system variable table at location 1090 ($442). For more information on using the timer interrupt vector, see the entry for address 1024 ($400) in Appendix K.

Chapter 3

XBIOS Device and System Functions

Functions known as the XBIOS (eXtended Basic input/Output System) are at the next level up from the BIOS. Where the BIOS contains a small number of very low level I/O routines used mainly by other system routines, the XBIOS contains a larger number of functions that are more specific to the ST environment, and of greater interest to the applications programmer. These functions deal with the character devices, the disk device, the screen display, the sound chips, the MFP (Multi-Function Peripheral adapter) chip, and miscellaneous system functions. This chapter covers the XBIOS device and system functions. The next chapter will cover the sound and graphics routines, which are of particular interest to ST programmers.

Like the BIOS functions, the XBIOS routines can be called from user mode. They use registers A0–A2 and D0–D2 as scratch registers, meaning that if you are programming in machine language and your program uses these registers, you must save their contents before making an XBIOS call, and restore them after the XBIOS call terminates. Each of the XBIOS routines is associated with a command number and, optionally, command parameters that specify more precisely what it should do. For example, the XBIOS function to set one of the hardware color registers has a command number of 7. It requires two command parameters: One tells the function which register to set and the other specifies the new color value (from 0 to 0x777).

To call an XBIOS function from machine language, you must push the command parameters onto the stack, followed by the command number, and execute a TRAP #14 statement. The TRAP #14 instruction puts the program into supervisor mode and begins executing the instructions found at the address stored in exception vector 46, whose address is 184 ($B8). This exception vector contains the address of the

XBIOS handler, which reads the command number on the top of the stack, and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register d0. When an XBIOS function call is completed, the calling program has the responsibility to adjust the stack in order to remove the command parameters and command number.

The following program fragment demonstrates how to change the value of color register 0 (the background color) to yellow ($770) using BIOS command number 7:

```
move.w   #$770,-(sp)   * push color value on stack
move.w   #0,-(sp)      * push color register number on stack
move.w   #7,-(sp)      * push XBIOS command number on
                       * stack
trap     #14           * call XBIOS handler
addq.l   #6,sp         * pop parameters (6 bytes) off stack
```

Calling the XBIOS routines from C is much simpler. Most C compilers come with a library routine called xbios( ) that stacks the parameters and executes the TRAP #14 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement:

```
xbios(7,0,0x770);
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OBSIND.H that defines macros for all of the XBIOS functions. For example, the macro definition for XBIOS command 7 is:

```
#define Setcolor (a,b)   xbios(7,a,b)
```

Therefore, after you #include OBSIND.H in your program, call your sample function like this:

```
Setcolor(0,0x777);
```

As this format is more readable than the other, the macros will be used in the discussion of XBIOS routines and sample programs. Just remember that in order to use XBIOS functions in your C programs, #include OSBIND.H if you use the macros, and link your program with the compiler library that contains the xbios( ) function.

Character Device Configuration Functions

The XBIOS character functions are more device-specific than those in the BIOS. One group of functions enables you to set the configuration of specific devices. For example, there are two functions that affect the performance of the console device. They allow you to

- Control the screen display's cursor
- Configure the keyboard portion of the console device

First, the function that allows you to control the screen display's cursor:

```
int rate, mode, newrate;
  rate = Cursconf(mode,newrate);
```

where *mode* is a flag that indicates which cursor function you wish to change. Possible choices are:

Mode Number	Cursor Setting
0	Turn cursor off
1	Turn cursor on
2	Turn cursor blink on
3	Turn cursor blink off
4	Change rate of cursor blink to *newrate*
5	Read cursor blink rate

The first four settings should be self-explanatory. They're used to either show or hide the cursor and to change it from a blinking block to a solid block. In these modes, the value of *newrate* is unimportant, and this parameter need not be passed.

When the cursor is blinking, you can also change the rate at which it blinks by the value you pass in *newrate*. This is a number from 0–255, that determines the time for each complete blink cycle (cursor on once, cursor off once) according to the following formula:

Duration of a single blink (in seconds) = 2 * rate / cycles

Cycles is a value that depends on the monitor and the video system used. Its value is 70 for the monochrome monitor, 60 for the U.S. color monitor, and 50 for the European color monitor. The default rate setting is 30. For the monochrome monitor, this makes the cursor blink every (2 * 30 / 70) seconds, or every .86 seconds. For the U.S. color monitor, the default setting works out to (2 * 30 / 60), or a blink every second. Settings lower than 30 make the cursor blink faster. At a setting of 1, it's just a blur. Higher settings make it blink more slowly. A setting of 0 is not the fastest setting, but the slowest (it represents 256). When you used mode number 5, the current blink-rate value is returned in the low byte of the *rate* variable. None of the other modes return any useful values in *rate.*

There is also a function that allows you to configure the keyboard portion of the console device. The ST console keyboard has a *typeamatic* feature that repeats the input if you hold down one of the keys for a moment. The delay before a key starts repeating, and the rate at which it repeats can be set with a call to the Kbrate( ) function in the following format:

```
int oldvals, delay, rate;
  oldvals = Kbrate(delay, rate);
```

where *delay* is the amount of time you must initially hold down a key before it starts to repeat, and *rate* is the time that elapses between each repetition. These times are measured in ticks of the system clock (200 milliseconds). Although each is a 16-bit value, only the low byte is used. As with the cursor speed setting, a rate of 0 represents the maximum delay between repeats, not the minimum. The default values set by the system are a delay of 17 and a repeat of 3. This means that you must hold down a key for 0.34 seconds before it starts to repeat; from then on the character will repeat every 0.12 seconds as long as you hold down the key.

A value of −1 for either *delay* or *rate* means that you wish that setting to remain as it is. For example, the call Kbrate(30, −1) changes the delay to 30, but leaves the repeat rate as it is. Note, however, that in the current (preblitter) version of the Operating System, a −1 in delay causes the rate to stay the same also, so you must place a nonnegative value in delay in order to change the rate.

Another character device configuration function allows you to set the printer configuration. This is a code number that contains information about the type of printer used. The user usually sets this code from the Install Printer desk accessory, and application programs can then read it and determine what type of printer is connected. ST Operating System programs (like the screen dump and GEM Desktop file printing routines) also use these settings when determining what kind of output to send to the printer. Note that the built-in screen print feature does not support some possible printer types (like Epson color printers, or color daisywheels). The format for the Setprt( ) call is:

```
int code, newcode;
  code = Setprt(newcode);
```

where *newcode* is a 16-bit flag used to describe various attributes of the printer. The meaning of each flag bit is shown in the following table:

Table 3-1. newcode Flag Bits

Bit Number	Description	Meaning of Value
0	Print type	0 = Dot-matrix
		1 = Daisywheel
1	Color type	0 = Monochrome
		1 = Color print
2	Control code type	0 = Atari
		1 = Epson
3	Print quality	0 = Draft
		1 = Final quality
4	Printer port	0 = Parallel
		1 = RS-232 serial
5	Paper type	0 = Continuous
		1 = Single Sheet
7		Reserved for future use
8		Reserved for future use
9		Reserved for future use
10		Reserved for future use
11		Reserved for future use
12		Reserved for future use
13		Reserved for future use
14		Reserved for future use
15		Must be 0

The old value of the printer configuration code is returned in the *code* variable. By setting *newcode* to −1 (0xFFFF), it's possible to read the current code value without changing it.

The next character device function allows you to configure the RS-232 serial port.

```
int speed, handshake, ucr, rsr, tsr, scr;
  Rsconf(speed, handshake, ucr, rsr, trs, scr);
```

The *speed* parameter controls the communications speed, which is sometimes called the baud rate. The ST XBIOS supports 16 standard rates of communication, the most common of which are 300, 1200, 2400, and 9600 bps (bits per second). The rates represented by the various values of *speed* are:

Speed Value	Communication speed (in bits per second)
0	19200
1	9600
2	4800
3	3600
4	2400
5	2000
6	1800
7	1200
8	600
9	300
10	200
11	150
12	134
13	110
14	75
15	50

The *handshake* parameter is used to indicate which method of flow control or *handshaking* is used. Flow control is used to ensure that the sender isn't sending characters faster than the receiver can handle them. When the receiver starts to fall behind, it tells the sender to stop sending characters until it can catch up. When it catches up, the receiver tells the sender to start sending again. One method of handshaking is known as XON/XOFF, after the ASCII characters used. Using this protocol, the receiver sends the XOFF character (Ctrl-S, ASCII 19) when it wants the other side to stop, and XON (Ctrl-Q, ASCII 17) when it wants it to start again. The other major handshaking protocol is known as RTS/CTS. This method involves the receiver and sender using the RS-232 hardware lines Ready To Send (RTS) and Clear To Send (CTS) to indicate when they are ready to send and receive characters, respectively. The meaning of the various *handshake* parameters are as follows:

Handshake Values	Handshake Method
0	No handshaking
1	XON/XOFF
2	RTS/CTS (not implemented in preblitter ROMs)

Note that a setting of 3 turns on both XON/XOFF and RTS/CTS, which is meaningless. Also note that in the first (preblitter) TOS ROMs, the RTS/CTS handshake method was not supported.

The other input parameters for the Rsconf( ) function, *ucr, rsr, trs,* and *scr* are used to set various 8-bit registers on the 68901 Multi-Function Peripheral interface chip (MFP). The first, *ucr,* sets the USART (Universal Synchronous/Asynchronous Receiver/Transmitter) control register. This controls various communications parameters such as parity, stop bits, and data bits per word. The function of each of the register bits are as follows:

Table 3-2. ucr Bits

Bit	Function
0	Not used
1	Parity type 0 = Odd 1 = Even
2	Parity enable 0 = Off 2 = On
3–4	Async start and stop bits Bit value 4 3 Number of Start and Stop Bits 0 0 No start or stop bits (synchronous) 0 1 One start bit, one stop bit 1 0 One start bit, 1½ stop bits 1 1 One start bit, two stop bits
5–6	Data bits per word Bit value 6 5 Number of Data Bits 0 0 Eight bits 0 1 Seven bits 1 0 Six bits 1 1 Five bits

Table 3-2. ucr Bits (continued)

Bit	Function
7	Clock 0 = Use clock directly for transfer frequency (synchronous transfer) 1 = Divide clock frequency by 16

The other input parameters control the Receive Status Register (*rsr*), Transmit Status Register (*tsr*) and Synchronous Character Register (*scr*). These need rarely be set by the user. A value of −1 in any of the input parameters will retain the previous value for that parameter. For more complete information on the MFP chip, see Appendix F.

The final character device configuration function concerns the three input devices, the serial port, the MIDI port, and the console keyboard. This function returns a pointer to a data structure known as the I/O buffer record. The buffer record contains a number of items of information about the input buffer used by the device. The function, Iorec( ), uses the following syntax:

```
int dev;
long bufrec;
  bufrec = Iorec(dev);
```

where *dev* specifies the devices whose buffer record will be fetched. Possible device number include:

Device Number	Input Device
0	RS-232 serial port
1	Console (keyboard)
2	MIDI port

The address of the device's buffer record is returned in the variable *bufrec*. The buffer record contains 14 bytes of data, laid out as follows:

Table 3-3. bufrec Byte Values

Byte Number	Element Name	Contents
0–3	ibuf	Address of the input buffer
4–5	ibufsize	Size of the input buffer (in bytes)
6–7	ibufhd	Index to head (next write position)
8–9	ibuftl	Index to tail (next read position)
10–11	ibuflow	*Low water* mark *
12–13	ibufhi	*High water* mark *

* Explained below.

An output buffer record immediately follows the input buffer record for the RS-232 serial device only.

Each input device has an input buffer where incoming characters are stored until retrieved by a call to Bconin( ). As characters are stored, the ST Operating System increments the index to the buffer head, which is an offset from the beginning of the buffer that shows where the next character will be stored. As characters are retrieved, the Operating System increments the index to the buffer tail, which is an offset from the beginning of the buffer that shows where the next character will be read.

If the head and tail of the buffer are the same, the buffer is empty. There are also offsets from the buffer tail which are known as the *high water mark* and the *low water mark*. These are used by devices that support handshaking. When the buffer head is a certain number of characters ahead of the buffer tail (the high water mark), the device signals the sender that it can't receive any more data. When the buffer head drops back to within a certain number of characters of the tail (the low water mark), the device signals the sender to resume transmission.

The ST Operating System sets up a default buffer for each input device. The size of these buffers are 256 bytes each for RS-232 input and output, and 128 bytes each for keyboard and MIDI input. Under normal circumstances, these buffer sizes are quite sufficient. When a continuous stream of bytes is coming in faster than calls to Bconin( ) can read them, however, it may be possible to push the head index past the tail, overflowing the buffer, and causing incoming data to be lost. In circumstances where data is coming in at a very rapid rate, the user may wish to replace the default buffer with a larger one of his own. Do this by declaring a block of variable storage and setting the buffer address pointer to that address. The programmer should save the address of the default system buffer and restore that buffer before the program terminates.

How?

MIDI and IKBD I/O

Two of the ST's character output devices, the MIDI port and Intelligent Keyboard controller (IKBD), don't come as standard equipment on most personal computers. These devices process commands that always consist of more than one byte of data. Therefore, the XBIOS contains functions for sending an entire string of characters to either device. The Midiws( ) function sends a string of characters out the MIDI port, and Ikbdws( ) writes a string of characters to the IKBD. These two functions are called like this:

```
int bytes;
long buffer;
  Midiws(bytes, buffer);
```

```
int bytes;
long buffer;
  Ikbdws(bytes, buffer);
```

where *bytes* is a value one less than the length of the character string in bytes, and *buffer* is the address of the memory buffer that contains the string. More information on the command set used by the intelligent keyboard device can be found in Appendix I. The protocol used by MIDI devices is too complex to include here. Complete details are contained in the official MIDI specification, a 71-page book published by the International MIDI Association. This book may be obtained directly from the IMA, 5316 West 57th Street, Los Angeles, CA 90056 (818)505-8964. Its price is currently $35.

Because initializing the mouse packet mode requires a number of commands to be sent to the IKBD, the XBIOS provides a function that lets you send all of them at once. This function is called Initmous( ), and it's called like this:

```
int mode;
long params, vector;
  Initmous(mode, params, vector);
```

where *mode* specifies the type of mouse information packets that the IKBD is to send. Possible values for mode include:

Mode Number	Mouse Mode
0	Mouse disabled
1	Mouse enabled in relative mode
2	Mouse enabled in absolute mode
3	Unused
4	Mouse enabled in keycode mode

The *params* variable points to a data block that contains the following parameters:

Table 3-4. Data Block Pointed to by params

Byte Offset	Label	Description
0	topmode	Specifies origin of *y* position 0 = *y* origin (0 point) at bottom 1 = *y* origin at top
1	buttons	The parameter for the IKBD set mouse buttons command
2	xparam	In relative mode, *x* threshold In absolute mode, *x* scale In keycode mode, *x* delta
3	yparam	In relative mode, *y* threshold In absolute mode, *y* scale In keycode mode, *y* delta
4*	xmax	Maximum *x* position of mouse
6*	ymax	Maximum *y* position of mouse
8*	xinitial	Initial *x* position of mouse
10*	yinitial	Initial *y* position of mouse

* Used only in mouse absolute mode.

The *vector* variable is used to pass the address of new mouse packet interrupt handler. If you're changing the mouse mode, chances are that the current handler will not work.

Both the MIDI and IKBD devices can send input to the ST, as well as receive output. Receiving input is a little more complex than sending output, however. Input from these devices is handled by a number of system interrupt routines which store the characters received in buffers. Information on the input buffers is provided in the section on the Iorec( ) — p76 function, above. The XBIOS also provides a function called Kbdvbase( ), which returns pointers to several of the actual interrupt routines that are used to handle the input functions. The syntax for this function is:

```
long vecbase;
  vecbase = Kbdvbase( );
```

where *vecbase* contains the address of a vector table. The structure of this vector table is as follows:

Table 3-5. Vector Table Structure Pointed to by vecbase

Byte Offset	Vector Name	Routine Called
0	midivec	MIDI input routine
4	vkbderr	IKBD ACIA overrun error routine
8	vmiderr	MIDI ACIA overrun error routine

Table 3-5. Vector Table Structure Pointed to by vecbase (continued)

Byte Offset	Vector Name	Routine Called
12	statvec	IKBD status packet handler
16	mousevec	IKBD mouse packet handler
20	clockvec	IKBD clock packet handler
24	joyvec	IKBD joystick packet handler
28	midisys	System MIDI ACIA handler
32	ikbdsys	System IKBD ACIA handler

The MIDI port and Intelligent Keyboard Controller (IKBD) hardware are connected to the ST system through an ACIA (Asynchronous Communications Interface Adapter) chip. When a byte has been input from either the MIDI port or the IKBD, the ACIA chip causes an interrupt to occur on the 68901 MFP chip. The interrupt handler then determines whether the interrupt was caused by the MIDI ACIA chip or the IKBD ACIA chip. If it was the MIDI chip, the midisys routine is called. This determines whether the interrupt occurred because a byte of data was received, or because of an overrun error. If a byte of data was the cause, the midivec routine is called to store the data (which is contained in the low eight bits of D0) in the MIDI buffer. If an error occurred, the vmiderr routine is called to handle the error.

If the source of the interrupt was the IKBD ACIA, the ikbdsys routine is called. It checks to see if the interrupt was caused by data received or by an error. If it was an error, vkbderr is called to handle it. If a data byte was received, the byte is checked to see if it was a keycode or an IKBD packet header. If the former is true, the keypress is handled. If the byte was a packet header, however, execution is directed through statvec, mousevec, clockvec, or joyvec, depending on what kind of packet is waiting. Of these four, the mousevec and clockvec vectors are used by the system, and should generally be left alone (particularly if you want system support for the mouse and clock functions to continue).

The statvec and joyvec vectors are not used by the system, however, and you may want to install you own handlers for these functions. For example, in order to use joysticks with your program, you must send a command to the IKBD to begin sending joystick information packets, and install your own joystick packet handler to process these packets. If you do install your own handler, remember at the point that it is entered, the address of the packet buffer will be on the stack, and in register A0.

It should not spend more than one millisecond handling the interrupt (most of the time, it will just move the packet information to your own buffer), and should end with an RTS instruction. Your routine should begin by saving all registers that you will use, and restore those registers before ending. Remember also that if you replace one of the vectors used by the system (mousevec, clockvec, or ikbdsys), you must either duplicate its actions in your own handler, or lose system-level functions (like mouse or keyboard support). Whenever you replace one of these vectors, always save their original contents, so that you can restore them before your program ends.

For more information on the various Intelligent Keyboard Controller packet types, see Appendix I.

One common type of I/O carried on with the IKBD is getting and setting the keyboard's time and date clock. The XBIOS provides functions that make it easy to get or set the clock. These functions are Settime( ) and Gettime( ):

```
long datetime;
  Settime(datetime);

long datetime;
  datetime = Gettime();
```

where *datetime* is a 32-bit value that specifies the date and time in DOS format. The bit groupings for *datetime* are:

Table 3-6. datetime Bit Groupings

Bit Number	Description	Range
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23
16–20	Day	1–31
21–24	Month	1–12
25–31	Year	0–119*

* Year value is added to 1980 to arrive at current year.

For example, if Gettime( ) returns a value of $0F976723, you could find the date and time by breaking the number down into its binary equivalent:

0000 1111 1001 0111 0110 0111 0010 0011

and grouping the bits as required:

```
0000111      1100      10111   01100   111001   00011
   7          12         23      12       49       3
  1987     December     23d          12:49:06 p.m.
```

This gives a year of 7 (1987), a month of 12 (December), a day of 23, an hour of 12 (noon), a minute value of 49, and a seconds value of 3. Thus, the date shown is December 23, 1987, and the time is 12:49:06 p.m. (The seconds value is the quotient of seconds divided by two.)
XGETTIME.C (Program 3-1) shows how to use Gettime( ) function to find the date and time.

Program 3-1. XGETTIME.C

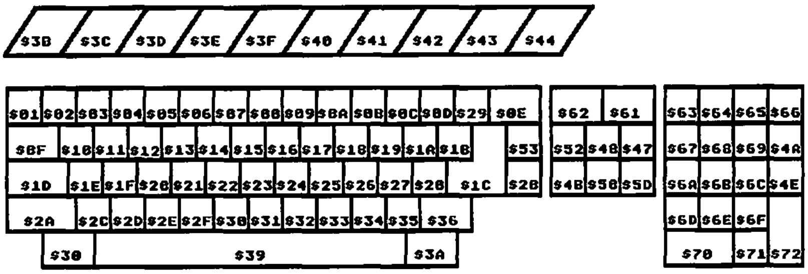
```
/**********************************************/
/*                                            */
/*   XGETTIME.C -- Demonstrates reading       */
/*   the IKBD clock/calendar and              */
/*   interpreting the results.                */
/*                                            */
/**********************************************/

#include <osbind.h>    /* For macro definitions */

main()
{
   unsigned long datetime, xbios();
   unsigned int date, time;
   unsigned int second, minute, hour;
   unsigned int day, month, year;

   datetime = Gettime();          /* get time and date longword */
   time = (int)datetime;          /* time is low word */
   date = (int)(datetime>>16);    /* date is high word */

   second = (time & 0x1F) *2;     /* secs/2  in 1st 5 bits */
   minute = (time>>5) & 0x3F;     /* minutes in next 6 */
   hour = time >> 11;             /* hours in last 5 */
   day = date & 0x1F;             /* day in 1st 5 bits */
   month = (date >> 5) & 0xF;     /* month in next 4 */
   year = (date >> 9) + 80;       /* year-1980 in last 6 */

   printf ("The date is %d/%d/%d and the time is %d:%d:%d \n",
           month,day,year,hour,minute,second);
}

/********** end of XGETTIME.C ********/
```

Note that though this function returns the date and time in the same format as the comparable GEMDOS functions, it does not use the same clock. Gettime( ) uses the hardware clock found in the IKBD device (or the Real-Time-Clock on the Mega STs), while the GEMDOS functions use a software clock maintained by GEMDOS. On 520 and 1040 STs, it is quite possible that these two clocks will not be set to the same time. The new (blitter) ROMs, however, set the GEMDOS clock from the hardware clock at the termination of every process.

Keyboard Vector Tables

The console device uses three sets of tables to tell it what ASCII character to return when it receives a certain key code from the IKBD device. One table has the ASCII values for unshifted keys, one has the ASCII values for keys pressed while holding down either Shift key, and one has the ASCII values for keys pressed while the CapsLock is in effect. Each table is 128 bytes long. Since there are only 104 keys on the ST keyboard, a number of the values in the tables are not used. The tables are arranged by key scan code. Since no key has a scan code value of 0, the first entry in the table is 0. Next comes the Esc key, which has a value of 1, followed by the 1 key which has a value of 2. The scan code for each key can be found in Appendix J, which shows all of the extended keyboard codes. The identifier is the first byte of the two-byte keycode value. For easier reference, a map of scan code values is given in Figure 3-1.

Figure 3-1. Map of Scan Code Values


$3B $3C $3D $3E $3F $40 $41 $42 $43 $44
$01 $02 $03 $04 $05 $06 $07 $08 $09 $0A $0B $0C $0D $29 $0E
$62 $61
$63 $64 $65 $66
$0F $10 $11 $12 $13 $14 $15 $16 $17 $18 $19 $1A $1B $53
$52 $48 $47
$67 $68 $69 $4A
$1D $1E $1F $20 $21 $22 $23 $24 $25 $26 $27 $28 $1C $2B
$4B $50 $4D
$6A $6B $6C $4E
$2A $2C $2D $2E $2F $30 $31 $32 $33 $34 $35 $36
$6D $6E $6F
$38 $39 $3A
$70 $71 $72

The TOS ROMs contain the default tables ordinarily used to map keys to their ASCII values. It is possible, however, to substitute a RAM table for one or more of these key maps. This allows you to change your keyboard layout to an alternate configuration, such as that used for Dvorak keyboards. An alternate key map may also be used to allow easy access to certain foreign characters or math symbols. The function with which you may alter the console keyboard tables is called Keytbl( ), and takes the following format:

char unshift[128], shift[128], capslock[128];
long vectable;
  vectable = Keytbl(unshift, shift, capslock);

where *unshift, shift,* and *capslock* are pointers to your own 128-byte tables. A value of −1 ($FFFFFFFF) in any of these pointers will signal the function that you wish to leave that table as it is. This function returns the pointer *vectable,* which contains the address of a vector table. This vector table contains pointers to each of the three keyboard tables. Its format is

Byte Number	**Contents**
0–3	Address of the unshifted table
4–7	Address of shifted table
8–11	Address of CapsLock table

The brief sample program XKEYTBL.C (Program 3-2), written in C, shows how to use the shift table for CapsLock as well. This causes the punctuation marks located on the number keys on top of the keyboard to be printed when CapsLock is set, but still allows you to print numbers with the numeric keypad.

Program 3-2. XKEYTBL.C

```
/**********************************************/
/*                                            */
/*  XKEYTBL.C -- demonstrates use of the      */
/*  XBIOS routine to change the keyboard      */
/*  mapping tables, so that CapsLock          */
/*  keys have their Shifted value.            */
/*                                            */
/**********************************************/

#include <osbind.h>   /* For XBIOS macro definitions */

main()
{
   struct keytab      /* Keytbl() returns a pointer */
        {              /* to this kind of structure   */
        char *unshift;
        char *shift;
        char *capslock;
        };

   struct keytab *vt;  /* Pointer to the vector table returned */

   vt = (struct keytab *)Keytbl(-1L,-1L,-1L); /* get vector table */
   Keytbl(-1L,-1L,vt->shift);   /* put value of shift in CapsLock */

}

/******** end of XKEYTBL.C *****/
```

Since the location of the default tables may change from version to version of the TOS ROMs, the XBIOS includes the Bioskeys( ) function, which allows you restore the default tables wherever they may be located. The syntax for this function is:

```
Bioskeys( );
```

Software developers should be aware that Atari has written a program called DEADKEYS.PRG, that allows the addition of foreign characters to the keyboard without remapping the current key assignments. This program adds itself to the beginning of the BIOS trap handler and checks to see if certain accent keys are struck. If they are, a flag is set so if the next key pressed is a vowel, the vowel is printed with the accent mark over it. The DEADKEYS.PRG program is available from Atari to registered software developers.

Screen Printing

TOS provides a screen print function. Whenever you press the Alternate-Help key combination, or choose the Print Screen menu item from the Options menu of the Desktop, the ST sends commands to the printer that cause it to print a graphic representation of the screen display, providing it's the right type of printer (Atari- or Epson-compatible) and it's properly installed (see Setprt( ) function, above). This same function can be performed under software control, by the XBIOS routine Scrdmp( ), which is called from C by the program statement

```
Scrdmp( );
```

As stated above, the default screen print routine only supports Atari and Epson-compatible dot-matrix printers. The ST screen print function can, however, be made to work with other printers by installing specialized printer drivers. To accommodate this, the Scrdump( ) routine is vectored through location 1282 ($502). This means that when Scrdump( ) is called, or the Alt-Help keys are pressed, program execution is directed to the routine whose address is found at that location. To install a printer driver for another printer, therefore, simply load the new screen print program as a Terminate-and-Stay-Resident program (see Ptermres( ), Chapter 5) and store its address in location $502. Since that location is in protected memory, first switch to Supervisor mode (see the Supexec( ) function below). If you want to keep the new driver installed after the program ends, use GEMDOS function 49 ($31) to keep the program code resident when it terminates.

The screen print vector at $502 can be diverted for other purposes in addition to installing new printer drivers. Some *snapshot* programs, for example, use this vector to install a routine that saves the screen picture to a disk file when the Alt-Help keys are pressed, rather than sending it to a printer. It's also possible to install a short routine that tests for shift-keys when Alt-Help is pressed, This allows additional *hot-key* programs to be installed, rather than just replacing the screen print function.

The default screen print code calls another XBIOS function to do the actual printing. This function can be used to print all of the screen or only a part of it. Its name is Prtblk( ), and it's called like this:

```
long prtable;
  Prtblk(prtable);
```

where *prtable* contains the address of a 30-byte parameter table that determines how the screen block is printed. The composition of this table is as follows:

Table 3-7. Structure of Table Pointed to by prtable

Byte Number	Element Name	Description
0–3	blkprt	Starting address of screen RAM
4–5	offset	Offset from start address (in bits, 0–7)
6–7	width	Screen width (in bytes)
8–9	height	Screen height
10–11	left	Left margin for screen dump
12–13	right	Right margin for screen dump
14–15	scrres	Screen resolution
		0 = Low
		1 = Medium
		2 = High
16–17	dstres	Printer resolution
		0 = Draft (960 dpi)
		1 = Final (1280 dpi)
18–21	colpal	Starting address of the color palette

Table 3-7. Structure of Table Pointed to by prtable (continued)

Byte Number	Element Name	Description
22–23	type	Printer type 0 = Atari monochrome dot-matrix 1 = Atari monochrome daisywheel 2 = Atari color dot-matrix 4 = Epson monochrome dot-matrix
24–25	port	Printer port 0 = Parallel 1 = RS232 serial
26–29	masks	Starting address of half-tone mask table (if 0, use default ROM table)

The prtblk( ) routine uses a number of RAM vectors. Before it sends a character to the printer, it jumps through a vector to a subroutine that returns the status of the printer. A return of −1 means the printer is ready, while a return of 0 means that it's still busy. When the printer is ready, prtblk( ) pushes the character to be printed onto the stack and jumps through another vector to the subroutine that actually outputs the character to the printer. The four RAM vectors, two for serial printers and two for parallel printers, are:

Table 3-8. RAM Vectors for Printers

Address	Vector Name	Description
1286 ($506)	prv_lsto	Pointer to lstostat( ), the PRN: device output status function
1290 ($50a)	prv_lst	Pointer to lstout( ), the PRN: device output routine
1294 ($50e)	prv_auxo	Pointer to autostat( ), the AUX: device status function
1298 (512)	prv_aux	Pointer to auxout( ), the AUX: device output routine

These vectors can be used to divert screen prints to other devices such as a laser printer connected to the DMA port. Note, however, that since prtblk( ) only supports the default printers, changing these vectors will not have any effect if a custom driver is installed at the vector at $502.

Floppy Disk Functions

The XBIOS includes a few functions that deal specifically with floppy disks as opposed to the more general BIOS disk routines. Among other functions, they are used to format and initialize a floppy disk. This operation requires several steps. The first is to format all the tracks on the disk, using the XBIOS routine Flopfmt( ), which formats and verifies a single track. The syntax for this call is:

```
long buffer, skewtabl, magic;
int status, devnum, spt, tracknum,
sidenum, intrlev, magic, initial;
  status = Flopfmt(buffer, skewtabl, devnum, spt, tracknum,
           sidenum, intrlev, magic, initial)
```

As you can see, you must supply quite a few parameters for this call. *Buffer* is a pointer to a memory buffer used to hold the data for the track image. For the normal layout (nine sectors per track), an 8K buffer is recommended. This buffer must start at an even address (a word boundary). The next parameter, *skewtabl,* is ignored in the first (preblitter) version of the TOS ROMs, which always write each sector sequentially on the track. The new blitter ROMs, however, allow sectors to be skewed within a track. While the first track has its sectors in the regular order:

Track 1: 1, 2, 3, 4, 5, 6, 7, 8, 9

the second track may have its sectors in a slightly different order:

Track 2: 8, 9, 1, 2, 3, 4, 5, 6, 7

Skewing the tracks this way makes sequential tracks read much faster. To use a skewed track format, place a −1 in the *intrlev* variable and have *skewtabl* point to a skew table that contains a 16-bit sector number for each sector, in the order in which sectors are to appear on successive tracks. If *intrlev* is set to something other than −1, the *skewtable* variable is ignored, but it must still be passed as a place holder.

The next parameter to pass is *devnum,,* the drive specifier. This value is 0 for drive A:, and 1 for drive B:. Next comes *spt,* which is short for *sectors per track.* The normal

Atari format calls for nine sectors per track. The Atari drives may reliably read and write ten sectors per track, however, and many users prefer format programs that use this value to expand each floppy disk's storage from 360K (720K double sided) to 400K (800K double sided).

The *tracknum* variable is used to specify the track number to be formatted. The normal Atari format uses track 0–79, though it's technically possible to use track 80 and sometimes even track 81. The *sidenum* variable is used for the side of the disk to format. Single-sided drives only use side 0, while double-sided drives use both 0 and 1.

The next parameter, *intrlev,* is a sector interleave format. In the old TOS ROMs, this is set to 1, but the new TOS format routines use a −1 to indicate sector skewing, as explained above. The *magic* parameter must be set to the number $87654321 for the format to work. Finally, *initial* is a 16-bit value to which all of the data bytes in a sector are initially set. Atari advises against the use of $0000 and recommends an initial value of $E5E5. In any case, the high nibble of each byte in this parameter must not equal $F.

On return from Flopfmt( ), the status parameter holds a status code. If there were no errors in formatting the track, its value will be 0. Any other code represents an error number, which indicates that the format operation failed. If the format fails due to sectors that could not be verified, a list of the bad sectors is returned in the buffer. This list consists of a string of 16-bit numbers, each representing a sector number (tracks start with sector 1) and terminated with a 0. This list is not necessarily in consecutive sequence. When a format operation fails, the format program may try to format the track again. If the format still fails after a couple of retries, the format program should note the bad sectors, and mark them as used in the File Allocation Table so the file system will not try to use these sectors. The exception to this is the first two tracks, which are used for the File Allocation Table and directory sectors. If any of the sectors in these tracks are bad, the media is unusable and formatting should be terminated.

Next, fill the sectors in the first two tracks with zeros. This initializes the File Allocation Table and directory. To write one or more sectors to a disk track, use the Flopwr( ) function:

```
int status devnum, secnum, tracknum, sidenum, numsecs;
long buf, resvd;
  status = Flopwr(buf, resvd, devnum, secnum tracknum,
           sidenum, numsecs);
```

where *buf* contains the address of a buffer that contains the data for one or more sequential sectors in a track. *Resvd* is a longword reserved for future use, which is currently ignored, but must be present. *Devnum* refers to the drive (0 = drive A:, 1 = drive B:). *Secnum* is the sector number at which to begin writing (ordinarily sectors are numbered 1–9). *Tracknum* is the number of the disk track, and *sidenum* the side of the disk to write to (0 or 1). *Numsecs* is the number of sequential sectors of data to write.

An error code is returned in *status*. If the status is 0, the operation was successful. Any other return represents a system error. For information on system errors codes, see Appendix D.

The final step in formatting a disk is to create a boot sector. This is a specially formatted block of information that is stored on the first sector on the disk (side 0, track 0, sector 1). It gives information about the disk storage format to the file system. You can create the block of information to be written to the boot sector using the Protobt( ) routine, which has the following syntax:

```
int disktype, execflag;
long buffer, serialnum;
  Protobt(buffer, serialnum, disktype, execflag);
```

where *buffer* contains the address of a 512-byte memory buffer where the boot block information will be created. *Serialnum* is a unique identifier code the file system uses to tell whether disks have been changed in a particular drive. Since each disk should have its own unique 24-bit number, pass a random number here. If you don't want to bother to generate a random number, the system will do it for you if you just pass a number larger than 24 bits ($1000000 or greater). The next parameter, *disktype,* is a code word that specifies the storage capacity and format of the disk. Possible values for disktype are:

Disktype Value	Disk Format
0	40 tracks, single sided (180K)
1	40 tracks, double sided (360K)
2	80 tracks, single sided (360K)
3	80 tracks, double sided (720K)
	Formats 2 and 3 are normally used for ST 3½-inch disks. Some 5¼-inch drives for the ST are formatted as type 1.

The last parameter, *execflag*, is used to indicate whether the disk is used to execute some boot code at startup time. This code consists of up to 480 bytes of machine language instructions, starting at byte 30 ($1E) of the boot sector. Most of the time, you'll place a 0 in this variable to show that the boot sector is not executable. If you place machine language instructions that you want executed at power up, indicate this by placing a one in *execflag*.

Protobt( ) can be used not only to create a new boot sector from scratch, but to modify an existing one as well. To use it this way, you could read an existing boot sector into the buffer with Floprd( ) (see below) and then call Protobt( ) with one or more parameters set to −1. If *serialnum, disktype,* or *execflag* are set to −1, Protobt( ) will leave that value as it currently exists is in the buffer and only change the other specified values.

Once Protobt( ) has been used to create the boot sector, all that remains is to write it to the first side 0, track 0, sector one of the disk. The XBIOS call Flopwr( ) should be used for this purpose and not the similar BIOS call Rwabs( ). For more information on the contents of a boot sector, see Appendix H.

The two remaining XBIOS disk functions are used to verify a floppy disk sector and to read one or more sectors from a floppy disk. Both are almost identical to Flopwr( ) in format. The verify function is called Flopver( ), and it's called as follows:

```
int status devnum, secnum, tracknum, sidenum, numsecs;
long buf, resvd;
  status = Floprd(buf, resvd, devnum, secnum, tracknum,
           sidenum, numsecs);
```

The function used to read sectors is called Floprd( ), and it's called this way:

int status devnum, secnum, tracknum, sidenum, numsecs;
long buf, resvd;
status = Floprd(buf, resvd, devnum, secnum, tracknum, sidenum, numsecs);

All of the parameters have the same meaning as in Flopwr( ), above. *Buf* contains the address for the memory buffer where the data read from the sectors will be stored. *Resvd* is a longword reserved for future use, which is currently ignored, but must be present. *Devnum* refers to the drive (0 = A, 1 = B). *Secnum* is the sector number at which to begin reading. *Tracknum* is the number of the disk track, and *sidenum* is the side of the disk from which to read (0 or 1). *Numsecs* is the number of sequential sectors of data to read. In the case of Floprd( ), the data will be read from the sectors. Flopver( ) not only reads the data, but compares the data that was read to the data still on the disk. Tracks are automatically verified as part of the Flopfmt( ) function. Also, sectors written with the BIOS function Rwabs( ) are automatically verified if the system variable *fverify* (at location 1092, $444) is set to a nonzero value, which is the default condition.

An error code is returned in *status*. If the status is 0, the operation was successful. Any other return represents a system error. For information on system errors codes, see Appendix D.

XFORMAT.C (Program 3-3) gives an example of how to format a double-sided disk with the XBIOS disk functions.

Program 3-3. XFORMAT.C

```
/***********************************************/
/*                                             */
/*                                             */
/*    XFORMAT.C--Simple double-sided           */
/*    floppy format using XBIOS routines       */
/*                                             */
/*                                             */
/***********************************************/

#include <osbind.h>     /* For XBIOS macro definitions */

int buf[4096];    /* buffer for track formatting, etc. */

main()
{
    int status, sector, track, side;

/* prompt for disk and wait for key press */
```

```
    printf("Insert disk to format in drive A and press Return\n\n");
    Bconin(2);     /* wait for a key press */

/* format tracks 0-79 on both sides */

    for(track=0;track<80;track++)  /* for 80 tracks.. */
        {
        for(side=0;side<2;side++)  /* on 2 sides */
            {
            printf("\33A Track %d, Side %d\n",track,side);
            /* format a track */
            status = Flopfmt(buf,0L,0,9,track,side,1,0x87654321L,0xE5E5);
            if (status != 0)  /* if there's an error, quit */
                {
                printf("Error %d at track %d, side %d\n",status, track,side);
                exit(100);
                } /* end of if */
            } /* end of for side */
        } /* end of for track */

/* Fill first two tracks on both sides with zeros */

    for(track=0;track<4096;buf[track++]=0); /* fill buf w/0's */
    for(track=0;track<2;track++)  /* for 2 tracks... */
        {
        for(side=0;side<2;side++) /* two sides each */
            {
            /* write zeros to track */
            status = Flopwr(buf,0L,0,1,track,side,9);
            if (status !=0)  /* if there's an error, quit */
                {
                printf("Sector zero failed, error %d\n",status);
                exit(100);
                }
            }
        }

/* prototype a boot sector, and write to side 0, track 0, sector 1 */

    Protobt(buf,0X01000000L,3,0);
    status = Flopwr(buf,0L,0,1,0,0,1);
    if(status !=0)   /* if there's an error, quit */
        {
        printf("Boot sector write failed, error %d\n",status);
        exit(100);
        }

/* if no errors, the disk is formatted! */

    printf("Format successful\n");
}

/*********** End of XFORMAT.C *********/
```

Though this program will format a disk, it is much simpler than the typical formatting program. It only formats a double-sided disk in drive A, and it quits the format procedure at the first sign of an error. Typically, a format program will retry formatting a track at least a couple of times before giving up and will mark bad sectors as used in the File Allocation Table instead of giving up on the whole disk if a couple of sectors are bad.

Accessing the I/O Chips

The ST uses a number of different I/O chips to perform its various input/output chores. The XBIOS provides functions that can help control two of these chips. One of them is the Programmable Sound Generator chip which will be discussed more fully in Chapter 4. In addition to its sound functions, this chip provides two 8-bit I/O ports. These are accessed through register 14 (I/O port A) and register 15 (I/O port B) of the sound chip. I/O port B is used for the 8 data bits that are transmitted through the Centronics parallel port to the printer. Port A, however, uses each bit for a different control function. The bit assignments for this port are:

Bit Number	Function
0	Floppy disk side 0/side 1 select
1	Floppy drive A select
2	Floppy drive B select
3	RS-232 Ready To Send (RTS)
4	RS-232 Data Terminal Ready (DTR)
5	Centronics STROBE line
6	General purpose output, available for application use (line connected to monitor port)
7	Reserved

Since these ports are located on the PSG chip, it's possible to read and write to them via the Giaccess( ) function explained in Chapter 4. In the case of port A, however, it isn't a good idea to do so. Since each bit is significant, make sure you change only the bits relevant to what you're doing. This usually entails reading the port, changing the value that you've read, and writing it back to the port.

TOS frequently changes the port A settings via interrupts, however. It uses this port to check the floppy disk drives for media changes, for example. This means that the port setting might change between the time you read it and the time you write back the new value. If that happens, you will unintentionally change the bits TOS has altered. The only way around this is to make sure that no interrupts occur between the time you read the port and the time you store your new value. This is known as changing the value atomically because it insures that the read and write operations will never be split by an interrupt. The XBIOS provides two functions to change individual bits on I/O port A. These functions are called like this:

```
int bitnum;
  Offgibit(bitnum);

int bitnum;
  Ongibit(bitnum);
```

where *bitnum* is the number of the bit (0–7) to change. Offgibit changes the specified bit number to a 0, while Ongibit changes the bit to a one.

The 68901 Multi-Function Peripheral (MFP) Chip is another important I/O chip on the ST. It contains an 8-bit parallel I/O port, a Universal Synchronous/Asynchronous Receiver/Transmitter (USART) for serial I/O, and four general-purpose timers. The serial port, the timers, and each bit of the parallel I/O port are capable of generating an interrupt. The MFP chip interrupts have an Interrupt Priority Level (IPL) of 6, but they are not autovectored. This means that when an MFP interrupt occurs, the IPL 6 interrupt handler is not called. Instead, the MFP chip directs execution to one of its own interrupt handlers. The 16 MFP interrupts (in order of priority), and their functions on the ST, are

Table 3-9. The 16 MFP Interrupts

Bit Number	Interrupt Source	ST Function
0*	I/O Port Bit 0	Parallel port busy
1*	I/O Port Bit 1	RS-232 Data Carrier Detect (DCD)
2	I/O Port Bit 2	RS-232 Clear To Send (CTS)
3*	I/O Port Bit 3	Graphics blitter chip done
4*	Timer D	RS-232 baud rate generator
5	Timer C	System Clock (200 Hz)
6	I/O Port Bit 4	Keyboard and MIDI ACIA data request
7*	I/O Port Bit 5	Floppy drive/DMA port data request
8*	Timer B	Horizontal blank counter
9	USART	RS-232 transmit error
10	USART	RS-232 transmit buffer empty
11	USART	RS-232 receive error
12	USART	RS-232 receive buffer full
13*	Timer A	User-defined timer interrupt
14*	I/O Port Bit 6	RS-232 Ring Indicator (RI)
15*	I/O Port Bit 7	Monochrome monitor detect

* See text below.

The interrupts that have an asterisk next to their bit number are initially disabled, but the user may enable or disable any MFP interrupt with one of the following XBIOS calls:

```
int intnum;
  Jdisint(intnum);
int intnum;
  Jenabint(intnum);
```

where *intnum* is the number of the interrupt (0–15), to enable or disable. Jdisint( ) is used to disable the interrupt, and Jenabint( ) is used to enable it.

The MFP includes four timers. Associated with each of these timers is

- A control register
- A data register
- A counter
- An interrupt vector
- A hardware output line

In addition, Timers A and B are connected to a hardware input line. All four timers may operate in what is known as delay mode. In this mode, the counter is decremented at each clock pulse until it reaches 0. At that point, an interrupt occurs, the hardware output line is pulsed, and the counter is loaded with the contents of the data register. Then the countdown process starts again.

Timers A and B can also operate in event count mode. In this mode, the counter is decremented not by the clock, but by pulses on the timer input line which come from an external hardware device. Timers A and B can also operate in pulse length mode. In this mode, the counter is decremented by clock pulses, but it can be turned on and off by pulses from the hardware input line.

TOS itself uses three of the four timers. Timer B is used for the Horizontal Blank counter. Timer C is used for the 200 Hz system clock that updates the GEM AES timer as well as executing the commands in the Dosound( ) queue. Timer D supplies the timing signal for the MFP's USART, which controls the communications speed (Baud rate) for the serial port. Thus, only Timer A is left free for use by applications. The XBIOS provides a function that lets you set the timer registers and interrupt vector. This function is called Xbtimer( ), and is called like this:

```
int timernum, control, data;
long vector;
  Xbtimer(timernum, control, data, vector);
```

where *timernum* is number from 0–3 that represents an MFP timer (0 = A, 1 = B, 2 = C, 3 = D). *Control* represents the value to place in the timers control register. This is an 8-bit register that controls the timer mode. For Timers A and B, possible values for this register include:

Control Value	Timer Mode
0	Timer off
1	Delay mode, clock divided by 4
2	Delay mode, clock divided by 10
3	Delay mode, clock divided by 16
4	Delay mode, clock divided by 50
5	Delay mode, clock divided by 64
6	Delay mode, clock divided by 100
7	Delay mode, clock divided by 200
8	Event Count Mode
9	Pulse Length mode, clock divided by 4
10	Pulse Length mode, clock divided by 10
11	Pulse Length mode, clock divided by 16
12	Pulse Length mode, clock divided by 50
13	Pulse Length mode, clock divided by 64
14	Pulse Length mode, clock divided by 100
15	Pulse Length mode, clock divided by 200

Data represents the value stored in the timer data register. Finally, *vector* is the address for the interrupt handler routine associated with this timer. To install your own timer interrupt routine for Timer A, set this value to the address of your interrupt code.

Miscellaneous System Routines

The remaining XBIOS functions perform a number of miscellaneous functions. The first of these is used to execute a subroutine in the 68000's supervisor mode. Normally, ST programs operate in what's known as user mode. Some privileged operations, however, can only be performed while in supervisor mode. For example, the address space from 0–2048 ($0–$800), and the I/O space from 16,744,448 ($FF80000) up is *protected*. If you attempt to access these memory areas while in user mode, you'll cause a bus error. Your program will display two bombs and it will crash. Therefore, if you wish to access any of the system variables in low memory, or any of the I/O registers, do so while in supervisor mode. The XBIOS function that runs a subroutine in supervisor mode is called Supexec( ), and its syntax looks like this:

```
long sub;
  Supexec(sub);
```

where *sub* is the address of the subroutine to execute in supervisor mode. An example of this function can be seen in the XGIACCES.C program in Chapter 4, in which it is used to change one of the low memory variables.

Another handy XBIOS function returns a 24-bit pseudorandom number. This function is called Random( ), and its syntax is

```
long rndnum;
  rndnum = Random( )
```

where *rndnum* is a 24-bit pseudorandom number (bits 24–31 are 0). The algorithm used to generate the number is:

```
SEED = (SEED * 3141592621) + 1
```

The value returned is the new seed value, shifted eight bits to the right. Since the initial seed value is taken from the screen's vertical blank frame counter, the sequence should be different each time the machine is turned on.

The lowest-level drawing routines are known as the line A routines. These routines are discussed fully in Chapter 7. In the first version of ST computers, these line A routines used software *blit* routines to perform the bit block transfers necessary to move images around on the screen. With the Mega ST series, however, Atari introduced a hardware blitter chip to speed up screen drawing. The line A routines in the Mega machines use the blitter hardware rather than the software blit routines. To maintain software compatibility across the entire ST lines, programmers are urged to use the line A routines for low-level drawing rather than writing directly to the screen, so their programs can take advantage of the blitter hardware if present. In some cases, however, a program may need to know if the blitter chip is available, and even to specify whether blit operations are to be performed by software or hardware means. The XBIOS routine used to get and set the blitter configuration has the macro name of Blitmode( ). Since many versions of the OSBIND.H file do not have this macro defined, you may need to add the line:

```
#define Blitmode(a)    xbios(64,a)
```

to that file in order to use the Blitmode( ) macro for this function. Once defined, Blitmode( ) is called like this:

```
int status, value;
  status = Blitmode(value);
```

where *value* is used to set the blitter configuration. If *value* is −1 (0xFFFF), no new value is set, and the current blitter configuration is returned. If *flag* is not −1, the blitter configuration is set using the following bit values:

Bit Number	Function
0	Set blit mode
	0 = use software blit routines
	1 = use blitter hardware
1–14	Undefined, reserved
15	Must be 0

The blitter configuration (as it stood prior to the set operation) is returned as a word value in *status*, each bit of which may have a meaning. The bit values are:

Bit Number	Description
0	Current blit mode
	0 = using software blit routines
	1 = using blitter hardware
1	Blitter chip availability
	0 = no blitter chip is available
	1 = blitter chip is installed
2–14	Undefined, reserved
15	A 0 is always returned

TOS will not allow the user to set the blit mode to *hardware* on a system that doesn't contain a blitter chip.

Chapter 4

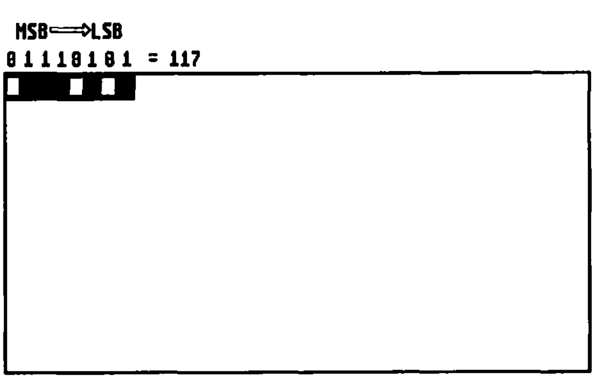
XBIOS Graphics and Sound Functions

Graphics

To understand the XBIOS graphics functions, you must first understand how the ST display is arranged in memory. The ST uses a bitmapped display. The bits in memory control the dots (picture elements or pixels) on the screen. On a bitmapped screen, everything, including text, is drawn using a number of dots.

The system used for the monochrome display on the ST is very simple. Each dot on the screen is represented by a single binary digit (bit) of memory. Screen memory is organized in such a way that the first byte represents the eight dots in the top left corner of the screen, and each succeeding byte represents the next eight dots to the right. Since each line contains 640 dots across, the first 80 bytes fill up the top line and the eighty-first byte is used to represent the first eight dots on the second line. There are 400 lines of 80 bytes each on the monochrome screen which means that 32,000 bytes of screen memory are used to represent the 256,000 dots on screen.

Figure 4-1. Monochrome Screen Memory



Each bit of screen memory can hold either the number 0 or 1. On a monochrome system, only one bit is needed to represent a screen dot or pixel (picture element), because each dot on the screen is either white (off) or black (on). But with a color ST system, things are somewhat different. In medium-resolution mode, any dot can be one of four colors. Two binary digits are used to yield four possible combinations:

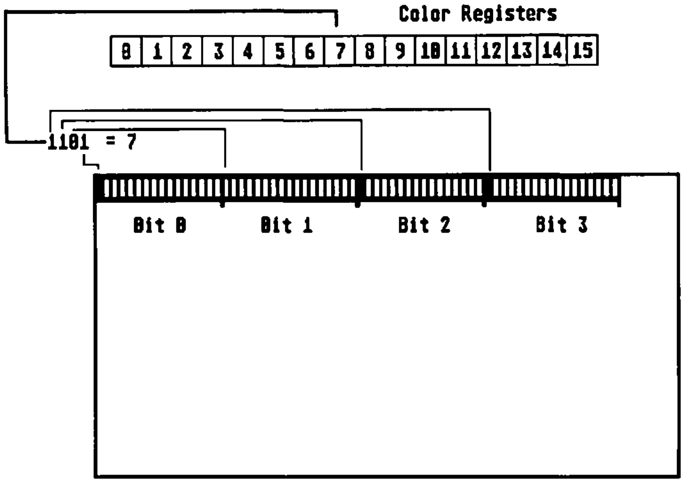
Bit Pattern	Value
00	0
01	1
10	2
11	3

Since each dot of color needs two bits of memory to describe it, each 16-pixel group on the screen is represented by two consecutive words in memory. The first word supplies the low bits of color information and the second supplies the high-color bits. In low-resolution mode, any dot can be one of 16 colors, so four bits are required to describe a single pixel. Each 16-pixel group on the screen is represented by four consecutive words in screen memory. The first word contains all of the low-color bits for the group, and each successive word holds the next higher bit (see Figure 4-2). In low- and medium-resolution modes, each row of dots requires 160 bytes of display memory, as opposed to the 80 bytes required for each line in the high-resolution mode. But since these modes use only 200 rows of dots, they each require the same 32,000 bytes of display memory as used by the high-resolution mode.

In monochrome mode, each bit can represent a specific color because there are two bit states (1 and 0) and 2 colors (black and white). But the ST color screen is capable of displaying 512 different colors. Clearly, in color mode, each set of bits can't represent a different color, using a code where 0 represents white, 1 stands for black, and so on. Instead, the number stored in the memory location that corresponds to a screen dot location refers to a hardware *color register*.

The color registers may be thought of as a set of 16 pens, each of which may be filled with ink that is colored in any of the 512 shades supported by the ST. Register 0 always

Figure 4-2. Lo-res color screen memory



holds the color we think of as the background color (which defaults to white on the ST). When you wish to use another color to draw a line or a point, specify the pen (color register) that will be used to draw it. Whatever color "ink" it currently contains is the color the pen will draw on screen.

Unlike ink, however, the color of a dot you have drawn on screen can change after you have drawn it. When the display memory for a screen dot holds the number of a particular pen, that dot is displayed in whatever color is in the pen at any given moment, not in the color that was in the pen at the time the dot was drawn. This means that if you use pen 1 to draw a line, and that pen contains the default color black, the line will be black. But if you change the color in pen 1 to green after you've drawn the line, the line you drew and everything else on the screen that was drawn with pen 1 will instantly become green.

The two factors that determine what colors are assigned to the figures you draw on the screen, therefore, are the register number used for the drawing, and the color that is currently contained in that register. Register colors are selected by specifying various levels of the color red, green, and blue. Each color register holds one of eight color levels for each of these colors, which means that there are 512 (8 × 8 × 8) possible colors to choose from. The default values in the color registers at power-up time are:

Table 4-1. Default Color Settings

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

You can change the color in an individual color register, however, using the XBIOS function Setcolor( ), whose syntax is

```
int oldcolor, register, newcolor;
  oldcolor = Setcolor(register, newcolor);
```

where *register* is the number of the hardware color register to change (0–15), and *newcolor* is a 16-bit color word, the low three nybbles of which are used for the red, green, and blue values respectively. For example, a value of 0x732 would indicate a red level of 7, a green level of 3, and a blue level of 2. The value contained in the color register prior to the call is returned in *oldcolor*. If *newcolor* is negative, the contents of the register are not changed; only the current color value is returned.

Since there are 16 hardware registers, setting them one at a time could be somewhat inconvenient, so the XBIOS provides a function that allows you to set an entire color palette at once. This function is called Setpalette( ), and its syntax is:

```
int palette[16];
  Setpalette(palette);
```

where *palette* is a pointer to an array that holds 16 words of data, each of which contains the color settings for one of the registers.

Since there are 512 possible color combinations, it's impossible to describe each one, or to explain exactly how to find a particular shade. In general, however, the higher the color level, the brighter the color will be, and the lower the level, the darker it will be. Whether the color displayed by a register tends toward red, green, or blue depends on which value has the highest brightness level. If all three values are equal, the color will be black, white, or a shade of gray. Thus, if red, green, and blue contain all 0s, the pen will be set to black, while a setting of straight 7s will make the register color white. You can lighten a shade by increasing the value of the two other colors in equal proportions. A setting of $700 selects bright red as the pen color, while $733 sets a much paler red. To darken the original red color, you could lower the red setting to 5 while keeping the two others at 0.

When you're unsure of what colors to mix, it may help to start with the nearest primary color mixture and experiment from there. The red, green, and blue values for these mixtures are

Table 4-2. Primary Colors

Color	Red	Green	Blue
Black	0	0	0
Blue	0	0	7
Green	0	7	0
Cyan	0	7	7
Red	7	0	0
Purple	7	0	7
Yellow	7	7	0
White	7	7	7

Unlike some computers, screen memory for the ST series is not fixed in any one particular place. To find the starting address of screen display memory, you may use the XBIOS function physbase( ):

```
long scraddr;
  scraddr = Physbase( );
```

where *scraddr* is a pointer to the beginning of the 32K memory block currently displayed. In addition to the physical display address, TOS also keeps track of a logical screen address. Operating System functions write on the logical screen when they output screen graphics. Normally, you'll want graphics output to go to the same memory area that is being displayed. There are some cases, however, where you'll want to draw a number of graphics objects sequentially and display them all at once so the user can't see you drawing each one in turn. In such cases, a separate logical screen comes in handy. To find the starting address of the logical screen memory area, use the logbase( ) function:

```
long logscraddr;
  logscraddr = Logbase( );
```

where *logscraddr* is the starting address of the logical screen.

The screen resolution mode is another important piece of information concerning the screen memory layout. Although all three ST screen modes use the same amount of screen memory (32K), different bits control different pixels in the various screen modes. The Getrez( ) function can be used to find the current screen resolution:

```
int rez;
  rez = Getrez( );
```

where *rez* is a code indicating the current screen resolution. The three *rez* values are

Resolution Number	Resolution
0	Low resolution (320 × 200 pixels, 16 colors)
1	Medium resolution (640 × 200 pixels, 4 colors)
2	High resolution (640 × 400 pixels, 2 colors)

We stated above that the addresses of the physical and logical screen memory block are not fixed. The user may change these addresses along with the resolution mode, using the function Setscreen( ). The format for this call is

```
int rez;
long logaddr, physaddr;
  Setscreen(logaddr, physaddr, rez);
```

where *logaddr* is the starting address for the logical screen, *physaddr* is the starting address for the physical screen display, and *rez* is the resolution mode (as shown above in the Getrez( ) description). A negative number in any of the three parameters will retain the current setting for that parameter. Changing the address of the logical screen allows you to write graphics to a part of memory that is not being displayed while changing the address of the physical screen lets you display another part of memory. In either case, the new screen memory block must begin on an even page (256-byte) boundary.

Changing the resolution mode only works if you're using a color monitor since TOS won't let you select a color mode if you're using a monochrome monitor, or vice versa. Therefore, you can only switch from medium to low resolution, or from low to medium. Even then, using Setscreen( ) to switch resolution modes only works for TOS programs. GEM cannot adjust to the switch. There is no known way to change a GEM screen from low to medium, or medium to low resolution modes, other than going back to the Desktop and using the Set Preferences menu option. Note that when you change resolution modes, the screen is cleared and certain other screen parameters are reinitialized.

The graphics display is maintained by an electron beam in the monitor that scans the screen from top to bottom. Changes in display memory made in midscan may show up as momentary glitches onscreen. To avoid this problem, display memory should be changed only during the interval that occurs when the scan reaches the bottom of the screen, before it starts over again at the top of the screen. This period is known as the *vertical blanking interval,* or *vblank.* On the ST, a priority level 4 interrupt occurs during each vblank. The XBIOS function Vsync( ) simply waits until the vblank interrupt is finished. When the function returns, it should be safe to change the screen. The format for Vsync is

```
Vsync( );
```

Vsync( ) can also be used in programs as a timed delay. Note, however, that the length of the Vsync( ) delay varies with the refresh rate of the monitor used. The monochrome monitor refreshes the screen 70 times per second, while the U.S. version of the color monitor refreshes the screen 60 times per second. The European (PAL) color monitor refreshes 50 times per second.

Contrary to Atari's documentation, the Physbase( ) function doesn't wait until the next vblank to return the physical screen location, nor does Setscreen( ) wait for a vblank to change the physical screen (at least not in the preblitter ROMs). This means that glitches can occur when changing from one physical screen to another, unless you call Vsync( ) before Setscreen( ). Some users report that these glitches may be eliminated by making sure the high-order word of the start address is the same for both screens (both screens are in the same, even 64K block of memory).

The following sample program (Program 4-1) gives a simple demonstration of page flipping. It sets up a second block of screen memory, writes a screenful of one character to that memory block, and then writes a screenful of a second character to the default screen memory area. When the user presses the space bar, the program toggles the display between the two screen areas. The program terminates when the user presses the *q* key.

Program 4-1. XSCREEN.C

```
/**********************************************/
/*                                            */
/*  XSCREEN.C -- Demonstrates use of the      */
/*  the XBIOS screen functions for "page      */
/*  flipping".                                */
/*                                            */
/**********************************************/

#include <osbind.h>     /* For macro definitions */
#define CON 2           /* device number of console keyboard */

main()
{
    long block, altscr, oldscr, screen;
    char ch=0;

    block = Malloc(0x7E00L);     /* allocate a second screen buffer */
    altscr = (block+256)&0x00FFFF00; /* align to page boundary */

    oldscr = Physbase();          /* find 1st (default) screen addr */
    Setscreen(altscr,-1L,-1L);    /* set 2nd logical screen */
    fill('X');                    /* fill it with X's */
    Setscreen(oldscr,-1L,-1L);    /* change back to 1st logical screen */
    fill('+');                    /* fill it with +'s */

    while(ch != 'q')              /* until 'q' is pressed */
      {
      ch = Bconin(CON);           /* wait for a key */
      if (ch==' ')                /* if it's the space bar */
         {
         if (altscr == Physbase())  /* toggle physical screen */
            screen = oldscr;
         else screen = altscr;
         Vsync();
         Setscreen(-1L,screen,-1L);
         }
      }
    Setscreen(oldscr,oldscr,-1L); /* at end, restore screens */
    Mfree(block);                 /* and de-allocate memory */
}
```

```
fill(ch)
   char ch;
{
   int x;

   Bconout(CON,27);  /* home cursor */
   Bconout(CON,'E');

   Bconout(CON,27);  /* turn on 'line wrap' */
   Bconout(CON,'v'); /* for console device output */

   for (x=0;x<1999;x++) /* fill screen with character x */
     Bconout(CON,ch);

}

/******** end of XSCREEN.c *****/
```

In Program 4-1, the GEMDOS Malloc( ) function was used to allocate memory for the second screen, and Mfree( ) was used to free up that memory when we finished. These functions will be explained more fully in Chapter 5. We also used one of the VT-52 escape codes to turn on the console screen's *line wrap feature.* This feature is explained in Appendix E.

Program 4-2 demonstrates how the preceding example might be translated into machine language.

Program 4-2. XSCREEN.S

```
************************************************************
*                                                          *
*     XSCREEN.S -- Demonstrates the use of the XBIOS       *
*     Setscreen() function from assembly language, as      *
*     well as the BIOS Conin() and Conout() routines.      *
*                                                          *
*                                                          *
************************************************************

***  Register usage:
***  a4 = pointer to 1st screen
***  a5 = pointer to 2nd screen
***  d4 = fill parameter for conout:
***  d5 = fill loop counter
***  d6 = toggle flag
***  d7 = fill character

*** Program starts here

.text

*** Make room for a 2nd screen, put screen pointer in a5

   sub.l    #32260,sp      * move stack down to make room for screen
   move.l   sp,d0          * copy stack pointer
   add.l    #258,d0        * round up to next page
   and.l    #$FFFFFF00,d0  * and down to even page boundary
   move.l   d0,a5          * place 2nd screen ptr in a5
```

```
*** Get pointer to 1st (default) screen in a4

   move.w   #2,-(sp)         * push XBIOS function # for Physbase()
   trap     #14              * call XBIOS
   addq.l   #2,sp            * clean stack
   move.l   d0,a4            * store pointer to default screen in a4

*** Set logical screen base to 2nd screen, and fill with X's

   move.w   #-1,-(sp)        * keep resolution the same
   move.l   #-1,-(sp)        * and keep physical base the same
   move.l   a5,-(sp)         * use alternate logical base
   move.w   #5,-(sp)         * function # for Setscreen()
   trap     #14              * call XBIOS
   add.l    #12,sp           * clean stack

   move.w   #'X',d4          * fill screen with X's
   jsr      fill

*** Set logical screen base to 1st screen, and fill with +'s

   move.w   #-1,-(sp)        * keep resolution the same
   move.l   #-1,-(sp)        * and keep physical base the same
   move.l   a4,-(sp)         * use default logical base
   move.w   #5,-(sp)         * function # for Setscreen()
   trap     #14              * call XBIOS
   add.l    #12,sp           * clean stack

   move.w   #'+',d4          * fill screen with +'s
   jsr      fill

*** use register d6 as a flag to tell what screen you're on

   move.b   #0,d6          * start on the default screen

*** wait for key press, check if it's a space or 'q'

mainloop:
   move.w   #2,-(sp)         * device number for console keyboard
   move.w   #2,-(sp)         * function # for Bconin()
   trap     #13              * call BIOS
   addq.l   #4,sp            * clean stack

testspace:
   cmp.b    #' ',d0          * was the space bar pressed?
   beq      toggle           * yes, toggle screen


testend:
   cmp.b    #'q',d0          * was the 'q' key pressed?
   bne      mainloop         * no, get next key
   move.b   #$80,d6          * yes, now flag signals you're at end

*** if it's a space, wait for vblank and toggle screen display

toggle:
   move.w   #36,-(sp)        * command number for Vsync()
   trap     #14              * call XBIOS
   addq.l   #2,sp            * clean stack

   move.l   a4,d0            * default screen for physbase
   cmp.b    #0,d6            * are you on the 1st screen now?
   bne      oldscreen        * no,change to screen 1
   move.l   a5,d0            * yes, change to screen 2

oldscreen:
   move.w   #-1,-(sp)        * keep resolution the same
   move.l   d0,-(sp)         * toggle the physical screen base
   move.l   #-1,-(sp)        * keep logical screen base the same
   move.w   #5,-(sp)         * function # for Setscreen()
   trap     #14              * call XBIOS
   add.l    #12,sp           * clean stack
   eor.b    #1,d6            * toggle flag

   bpl      mainloop         * and get next key (if not at end)
```

```
endprog:
   move.l   #0,-(sp)       * Push command number for terminate program
   trap     #1             * call GEMDOS.  Bye bye!

******** Fill subroutine.  Fills screen with a single character

fill:

*** First, print Esc-E to clear screen & home cursor

   move.w   #27,d7         * output ESC character
   move     #0,d5          * do only once
   jsr      conout         * do output

   move.w   #'E',d7        * output 'E'
   move     #0,d5          * do only once
   jsr      conout         * do output

*** Next, print ESC-v to turn on line-wrap

   move.w   #27,d7         * output ESC character
   move     #0,d5          * do only once
   jsr      conout         * do output

   move.w   #'v',d7        * output 'v'
   move     #0,d5          * do only once
   jsr      conout         * do output

*** Then, fill with parameter passed in d4

   move     d4,d7          * output the chosen character
   move     #1998,d5       * repeat 1999 times

conout:
   move.w   d7,-(sp)       * output character
   move.w   #2,-(sp)       * device number for console keyboard
   move.w   #3,-(sp)       * function # for Bconout()
   trap     #13            * call BIOS
   addq.l   #6,sp          * clean stack
   dbra     d5,conout      * do next letter
   rts                     * until done, then return

.end
```

While the source code for the machine language version of the program is over twice as long as that of the C version, the executable program it produces can be as small as one tenth the size of the C version of the program. Another interesting difference is that in the machine language version, the Malloc( ) function wasn't used to allocate memory for the second screen. As noted in Chapter 1, you can't allocate memory using this function unless you first relinquish part of the Transient Program Area (TPA) using the Mshrink( ) command.

Since machine language programs deal with memory more directly than C programs, it isn't necessary to use complex memory management techniques in such a simple example program. Since we know that the program sits at the bottom of free memory and the stack starts at the top of free memory and works its way down, it's simple to move the stack pointer down far enough to create a safe area at the top of memory and start the second screen on the even page boundary above the stack. As an alternative, we could have used the Declare Storage (.ds) directive to allocate 32K of storage at the end of the program for the second screen, but that would have increased the size of our program to over 32K. Using the method shown above, the entire program takes just over 300 bytes. Of course, for more complex programs, you will probably want to use more sophisticated memory management techniques that require you to shrink the size of the TPA with Mshrink( ). This topic will be discussed in more detail in Chapter 5 when the Mshrink( ) function is explained.

Sound Functions

The ST series of computers use a Yamaha YM-2149, a version of the General Instruments AY-3-8190 Programmable Sound Generator (PSG) chip for music and sound effects. This chip has three sound channels, that can produce continuous tones, various envelope waveforms, or noise. It also has two 8-bit I/O ports, which are used on the ST for output functions related to the floppy disks, RS-232 serial port, and Centronics parallel printer port.

The PSG chip has 16 internal registers, but it communicates to the processor through two external ports. The first port (which in the current STs is addressed at $FFFF8800) is used for selecting the internal register to read or write. To select a register, write a register number (0–15) to this port. After you have selected a register, read it through the first port, or change it by writing a the new value to the second port (which in the current STs is addressed at $FFFF8802). This procedure is complex, so the XBIOS provides a function that allows you to read or change any PSG register in a single step. This function is called Giaccess( ), and its called like this:

```
char regvalue, value;
int regnum;
  regvalue = Giaccess(value, regnum);
```

where *value* is the new 8-bit number to go into the register and *regnum* specifies the register number to use. If you are using this function to read the register, the register number (0–15) is used in *regnum*. If you're using the function to write a new value to the register, add 128 ($80) to the register number. In either case, the function returns the value stored in the register at the end of the call in the variable *regvalue*. The functions of each of the 16 registers of the PSG chip are shown in Table 4-3.

Table 4-3. The Functions of the 16 Registers of the PSG Chip

Register	Bits	Description
0	0–7	Low byte of Channel A tone period
1	0–3	High nybble of Channel A tone period
2	0–7	Low byte of Channel B tone period
3	0–3	High nybble of Channel B tone period
4	0–7	Low byte of Channel C tone period
5	0–3	High nybble of Channel C tone period
6	0–5	Noise period
7	0	0 = Enable tone for Channel A
	1	0 = Enable tone for Channel B
	2	0 = Enable tone for Channel C
	3	0 = Enable noise for Channel A
	4	0 = Enable noise for Channel B
	5	0 = Enable noise for Channel C
	6	0 = Enable I/O Channel A for input
	7	0 = Enable I/O Channel B for input
8	0–3	Channel A volume level (0–15) (for constant amplitude mode)
	4	Amplitude mode 0 = Keep volume constant 1 = Use waveform envelope to vary volume
9	0–3	Channel B volume level (0–15) (for constant amplitude mode)
	4	Amplitude mode 0 = Keep volume constant 1 = Use waveform envelope to vary volume
10	0–3	Channel C volume level (0–15) (for constant amplitude mode)
	4	Amplitude mode 0 = Keep volume constant 1 = Use waveform envelope to vary volume
11	0–7	Low byte of envelope period
12	0–7	High byte of envelope period
13	0–3	Waveform envelope (see Figure 3-4 for waveform shapes)
14	0–7	I/O Port A (Used to control floppy disk drive, RS-232, and Centronics parallel ports)
15	0–7	I/O Port B (Used for parallel port data)

To produce a continuous tone on one of the sound channels on the PSG chips, you must enable the channel for tone, set the volume level, and set the tone period to determine the pitch of the tone. Enable a channel to produce a tone by setting the corresponding tone bit in register 7 to 0. Be very careful when setting this register, however, since the upper two bits are used to determine the direction of data flow for the two onboard I/O ports. The correct procedure is to read the register first and change only the bits of interest, using logical AND or logical OR functions. Next, set the tone period registers. The tone period is a 12-bit number (0–4095) that determines the pitch of the tone. To set tone period place values in two registers, one for the low byte (bits 0–7), and one for the high nybble (bits 8–11). The relationship of this tone period to the pitch of the note is determined by the formula

period = clock / frequency (Hz) * 16

where *clock* is the clock speed of the PSG (2,000,000 cycles per second). This formula can be reduced to:

period = 125,000 / frequency (Hz)

or

frequency (Hz) = 125,000 / period

As the period value increases, the pitch becomes lower in tone. The period settings for the notes of the chromatic scale, beginning with middle C, are shown in Table 4-4.

Table 4-4. Relationship of Period, Note, and Frequency Across One Octave

Period	Note	Frequency
478	C	261.6
451	C#	277.2
426	D	293.7
402	D#	311.1
379	E	329.66
358	F	349.2
338	F#	370.0
319	G	392.0
301	G#	415.3
284	A	440.0
268	A#	466.2
253	B	493.9

To derive the period values for the next higher octave, divide the period values in this table in half. For the next lower octave, multiply each period value in the table by 2.

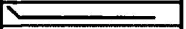
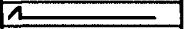
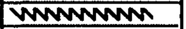
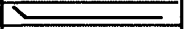
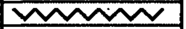
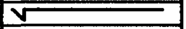
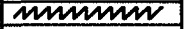
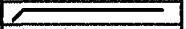
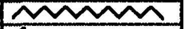
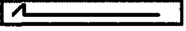
The final step in sounding a constant tone is to set the volume level in register 8, 9, or 10, depending on which channel you're using. The low four bits of this register are used to set the volume from 0 (silent) to 15 (maximum volume). Bit 4 should be set to 0 if you're trying to create a sound that has a constant value. Once you have enabled a tone channel, set the pitch and selected a volume, the sound will continue to play until you disable the channel or set the volume to 0.

It's also possible to create tones that use various waveform envelopes instead of a constant volume. These tones vary in volume level according to the waveform chosen. A single waveform may be selected for all three voice channels by setting register 13. A chart of the available waveforms is shown in Figure 4-3.

When using the waveform envelopes, the rate at which the volume level changes is determined by the waveform period, which is a 16-bit value placed in registers 11 and 12. The higher the period, the more slowly the volume level changes, and the lower the period value, the more quickly it changes. At high period values, the tones created are bell-like, while at low period values, they are very raspy. To create a tone using one of the waveform envelopes, set the frequency and enable the tone channel as with a constant tone. Then, set the waveform and waveform period in registers 11–13. Finally, set bit 4 of the proper volume register to one. Note that tones whose waveforms end in a low, flat line only sound once, while those that stay up or fluctuate up and down sound continuously until stopped.

The third type of sound that can be created by the PSG uses the noise generator. Noise channels vary in frequency, at a rate determined by register 6, the noise period setting. This is a 5-bit setting that varies from a thin, static-like sound (low settings) to a sound like the rushing wind (high settings). Noise may be enabled by setting the noise enable bits of register 7. Like tones, noise may have a constant volume or a volume that varies according to a waveform envelope. Noise whose volume is varied according to a waveform with a large period sounds percussive, like gunshots, drums, or cymbals. Those whose volume is varied according to a small-period waveform hum like a motor. Both noise and tones may be enabled in a single channel, which then produces both types of sound.

Figure 4-3. Waveform Shapes and the Sounds They Create

Register 13	Waveform Control Bits				Waveform Selected
	Bit 3	Bit 2	Bit 1	Bit 0	
Decimal Value	Continue	Attack	Alternate	Hold	
0-3	0	0	-	-	
4-7	0	1	-	-	
8	1	0	0	0	
9	1	0	0	1	
10	1	0	1	0	
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	
14	1	1	1	0	
15	1	1	1	1	

Envelope Period
(duration of one cycle)

Program 4-3 demonstrates the use of the XBIOS Giaccess( ) call to produce tones of a constant volume. It turns the top row of the ST keyboard into a musical keyboard.

Program 4-3. XGIACCES.C

```
/************************************************/
/*    XGIACCES.C -- Demonstrates use of        */
/*    PSG registers to produce musical         */
/*    tones. The program turns the top         */
/*    row of the keyboard into a 'piano'.      */
/*    To quit, press "q".                      */
/*                                              */
/*                                              */
/************************************************/

#include <osbind.h>     /* For macro definitions */
#define CON 2           /* console keyboard device no. */

unsigned notes[13] =    /* the period setting for the notes */
                        /* of the chromatic scale */
{
478,    /* C  = 261.6 Hz */
451,    /* C# = 277.2 Hz */
426,    /* D  = 293.7 Hz */
402,    /* D# = 311.1 Hz */
379,    /* E  = 329.6 Hz */
358,    /* F  = 349.2 Hz */
338,    /* F# = 370.0 Hz */
319,    /* G  = 392.0 Hz */
301,    /* G# = 415.3 Hz */
284,    /* A  = 440.0 Hz */
268,    /* A# = 466.2 Hz */
253,    /* B  = 493.9 Hz */
239     /* C  = 523.2 Hz */
};
main()
{
    int clickon(), clickoff(); /* address of "super" functions */
    unsigned period, x;
    char regvalue, ch=0;

    puts("Press keys on top row of keyboard to hear notes");
    puts("Press 'q' to quit");

    Supexec(clickoff); /* turn the keyboard click off */

    regvalue = Giaccess(ch,0x07); /* read current tone register */
    Giaccess(regvalue | 0x3E, 0x87);  /* turn on channel A tone */


    while(ch!=16)  /* until "q" key is pressed */
        {
        ch = (char)(Bconin(CON)>>16); /* wait til key press */
        if (ch < 14)                  /* & get scan code */
           {                          /* if on top row of keyboard */
           period = notes[ch-1];      /* find period */
           Giaccess(period & 255, 0x80);  /* set period low byte*/
           Giaccess(period >> 8,  0x81);   /* set period high byte*/
           Giaccess(15, 0x88);     /* set channel A volume */

           for (x=0;x<20;x++)Vsync(); /* let it play a while */
           Giaccess(0, 0x88); /* turn note off */
           }
        }
    Supexec(clickon); /* turn key click back on and quit */
}

clickoff()
{
    char *conterm;

    conterm = (char *)0x484L;
    *conterm &= 0xFE; /* turn click bit off */
}

clickon()
{
    char *conterm;

    conterm = (char *)0x484L;
    *conterm |= 0x01; /* turn click bit on */
}


/************* End of XGIACCES.C *************/
```

This program uses both the console keyboard and sound channel A. TOS, however, also uses sound channel A to produce a click sound when a console key is struck. To avoid conflict between your program and the TOS routines, shut off the key click sound at the beginning of the program and turn it back on at the end. Do this by altering the system variable *conterm*, which is stored at location 1156 ($484). Since all of the low-memory variables are protected from access by programs running in user mode, switch into supervisor mode to change this system variable. The XBIOS routine Supexec( ) (explained more fully in Chapter 3) is used to execute the clickoff( ) and clickon( ) routines in supervisor mode.

Making effective use of the ST sound capabilities often calls for sounds or music to be playing at the same time that other input/output events are occurring. For this reason, the XBIOS provides a routine that can be used to play sounds in the background, unattended. This function is called Dosound( ), and its syntax is as follows:

```
long commandlist;
  Dosound(commandlist);
```

where *commandlist* is a pointer to a data storage area that contains a number of queued sound commands. Each command is either two or four bytes in length. When Dosound( ) is called, a flag is set that tells the timer interrupt handler routine to start executing one sound command each time the timer interrupt occurs. From then on, the timer interrupt routine reads the next sound command on the list every 1/50 second and executes that command. This continues until the command to stop is issued. In this way, music may be made to play in the background while the rest of the program continues.

The structure of the commands contained in the command list is detailed in Table 4-5.

Although there are four separate commands, there are only three distinct command sequences, since commands 128 and 129 work together. The first command, which starts with a value ranging from 0 to 15, is very simple. It specifies that the corresponding sound register be loaded with the byte value that follows. The last command, which starts with any number larger than 129, is also simple. It specifies that the timer interrupt pause in its execution of sound commands for the number of timer ticks indicated by the following byte. This allows the user to start a note, let it play for a specified period of time, and then stop it.

The middle commands are a bit trickier. They allow the user to change a register from a beginning value to an ending value, using a specified increment value. This lets you automatically change the frequency or volume of a tone for

Table 4-5. Structure of Commands in Command List

Command Number First Byte	Second Byte	Third Byte	Fourth Byte
0–15 Load a byte value into the register specified by this command byte (0–15)	The byte value to load into the register (0–255)		
128 Store a value in a temporary register for use by command number 129 (see below)	The byte value to store in the temporary register (0–255). This is the starting register value for command 129.		
129 Load a register with the value stored in the temporary register (by command 128). Increment the value in the register each timer tick, until an end value is reached.	The number of the register to use (0–15).	Increment value (0–255). Added to register value each timer tick. Can be positive (0-127) or negative.	Final register value. Command ends with this value reached.
130–255: Pause for a specified number of timer ticks (1/50 second) before the next sound command is executed.	The number of timer ticks to pause (1–255). A value of 0 will end the processing of sound commands.		

effects like wailing sirens. The structure of these commands is like the FOR command in BASIC, which lets you run a loop a certain number of times by specifying a starting variable value, an ending value, and the number added to the variable each time through the loop. In BASIC, this command would look like this:

FOR ***value = start*** **TO** ***finish*** **STEP** ***increment***
POKE ***register, value***
NEXT ***value***

Since each sound command can only have a maximum of four bytes, and there are five parameters to pass here (the sound command, register number, starting value, ending value, and increment value), two commands are used. Command 128 is used to store the starting value in a temporary register, while command 129 supplies the register number, the increment, and the ending value. There are two important points to remember about these commands. First, twos-complement addition is used when adding the increment to the current register value, meaning that the numbers 128–255 are treated as negative numbers. Therefore, you can count a value down as well as up. Secondly, the end value must be matched exactly in order for the loop to end. If you specify 6 as your starting number and 25 as the ending number, with an increment of 2, the loop will never end, since you'll go from 24 to 26 without ever hitting 25. This feature can be used to your advantage for repeating loops.

Program 4-4 demonstrates the use of the Dosound( ) command to play a tune in the background while other processing continues. It plays the first few notes of *Twinkle, Twinkle, Little Star,* while printing out the words at the same time.

Program 4-4. XDSOUND.C

```
/************************************************/
/*                                              */
/*  XDSOUND.C -- Demonstrates use of XBIOS      */
/*  Dosound() command for executing a series    */
/*  of sound events via the timer interrupt     */
/*  (plays Twinkle, Twinkle Little Star)        */
/*                                              */
/*                                              */
/************************************************/

#include <osbind.h>    /* For macro definitions */

unsigned commands[]=  /* list of Dosound() commands */
{
0x73F,0x080F,0x00DE,0x0101,0x073E,0xFF11,0x0800,0xFF05,
0x080F,0xFF11,0x0800,0x003F,0x0101,0xFF03,0x080F,0xFF11,
0x0800,0xFF05,0x080F,0xFF11,0x0800,0x001C,0x0101,0xFF03,
0x080F,0xFF11,0x0800,0xFF05,0x080F,0xFF11,0x0800,0x003F,
0x0101,0xFF03,0x080F,0xFF22,0x0800,0xFF00
};

int scale;  /* scaling factor for delay loop.*/
            /* Timer interrupt is always 1/50th of a second. */
            /* Vsync() is 1/60th of a second on color monitor,*/
            /* 1/70th of a second on monochrome */
main()
{
    int x;

    scale = Getrez();         /* get monitor type, and set */
    if (scale < 2) scale=6;   /* scaling factor accordingly */
    else scale = 7;

    Dosound(commands);        /* play the song in the background */

    puts("Twinkle,\n");       /* print the words while it plays */
    wait(48);                 /* delay 48 timer ticks */
    puts("Twinkle,\n");
    wait(48);
    puts("Little\n");
    wait(48);
    puts("Star.\n");
    wait(200);
}

wait(y)         /* Delay a set number of timer ticks */
int y;
{
    int x;

    y = y*scale/5; /* convert timer ticks to vblanks */
    for(x=0;x<y;x++)Vsync;
}

/****************** End of XDOSOUND.C ********/
```

To synchronize the music with the words, the Vsync( ) function was utilized to wait a number of vertical blanking intervals. But vblanks don't coincide exactly with timer ticks. There are 70 vertical blank interrupts per second on the monochrome monitor, 60 per second on the color monitor, and only 50 timer interrupts per second with either kind of display. Therefore, the timer ticks had to be scaled to match the corresponding number of vblanks by multiplying them by 7/5 for the monochrome display and 6/5 for the color display.

Chapter 5

GEMDOS Device I/O and Process Control

The functions that make up the GEMDOS (GEM Disk Operating System) form the highest level of TOS. These functions are also sometimes referred to as the BDOS (using the old CP/M and MS-DOS operating system terminology). They include a wide variety of character device functions, several process control and memory management functions, and a large number of functions used to control file I/O and the filing system. The character device and system routines will be covered in this chapter, while the next chapter will be devoted to the filing system routines. In many cases, GEMDOS functions are modeled after similar commands available under the MS-DOS operating system used on the IBM PC. In fact, most of them share the same function numbers as their DOS counterparts. That's why GEMDOS functions are often referred to by a hexadecimal function number, just like MS-DOS functions.

Like the BIOS and XBIOS functions, GEMDOS routines can be called from user mode. As with those functions, GEMDOS uses registers A0–A2 and D0–D2 as scratch registers; assume that it changes their contents. If you are programming in machine language and your program uses these registers, save their contents before making a GEMDOS call and restore them after the call terminates. Each of the GEMDOS routines is associated with a command number, and some use command parameters that specify more precisely what they should do. For example, the GEMDOS function to write a character to the console screen has a command number of 2. It requires a single command parameter that tells the function which character to print.

To call a GEMDOS function from machine language, push the command parameters onto the stack, followed by the command number, and execute a TRAP #1 statement. The TRAP #1 instruction puts the program into supervisor mode and begins executing the instructions found at the address stored in exception vector 33, whose address is 132 ($84). This exception vector contains the address of the GEMDOS handler which reads the command number on the top of the stack and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register D0. In most cases, the value is returned as a longword, but there are exceptions. Some error codes are returned as words, so it's best to test only the low-order words when checking for errors. Also be aware that sometimes a GEMDOS function will return a BIOS error number (between −1 and −31). When a GEMDOS function call is completed, the calling program is responsible for adjusting the stack to remove the command parameters and command number.

The following program fragment demonstrates how to print the character *A* on the console screen using GEMDOS command 2:

```
move.w   #'A',-(sp)   * push the character value on stack
move.w   #2,-(sp)     * push GEMDOS command number on
                      * stack
trap     #1           * call GEMDOS handler
addq.l   #4,sp        * pop parameters (4 bytes) off stack
```

Calling the GEMDOS routines from C is much simpler. Most C compilers come with a library routine called gemdos( ) that stacks the parameters and executes the TRAP #1 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement:

```
gemdos(2, 'A');
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OSBIND.H that defines macros for all of the GEMDOS functions. For example, the macro definition for GEMDOS command 2 is:

```
#define Cconout(a)   gemdos(0x2,a)
```

Therefore, after you #include OSBIND.H in your program, you can call your sample function like this:

```
Cconout('A');
```

Since this format is the more readable of the two, it will be used in the macros in the discussion of GEMDOS routines and sample programs. To use GEMDOS functions in your C programs, link your program with the compiler library that contains the gemdos( ) function, and #include OSBIND.H if you use the macros.

Character Device I/O

Like the BIOS and XBIOS, GEMDOS includes a number of functions that allow you to read characters from a character device, write characters or strings to such devices, and check their input and output status. These functions aren't implemented in exactly the same way as the BIOS and XBIOS versions, however. Where the BIOS and XBIOS tend to use one function for many devices and require a device number as an input parameter, GEMDOS provides a separate function for each device. GEMDOS also throws in a number of variations. Some keyboard input functions echo the character to the screen, some don't. Some wait for a keypress to return, others return immediately.

There are three functions that wait for a character to be entered at the console keyboard and return the value of the character that was entered. The macro for the first of these functions, Cconin( ), is called like this:

```
long keycode;
  keycode = Cconin( );
```

When called, this function doesn't return until a key is pressed, unless a keypress is already waiting in the queue. The *keycode* the function returns is a longword containing both the ASCII value of the key(s) pressed and the scan code. The ASCII value is returned in the low byte of the low word of *keycode,* while the scan code is returned in the low byte of the high word. See Appendix J for a complete list of keycodes. This call is much like the BIOS function Bconin( ), except that Cconin( ) sends a copy of the key that was pressed out to the console screen.

The other two character input functions, Crawcin( ) and Cnecin( ), don't echo the key that is pressed. These functions are called just like Cconin( ):

```
long keycode;
  keycode = Crawcin( );
long keycode;
  keycode = Cnecin( );
```

Again, both functions wait until a key is pressed and then return the ASCII character and scan code. Atari documentation indicates there is slight difference between these functions in that Crawcin( ) is supposed to pass all control codes, and Cnecin( ) is supposed to act on control codes like Control-S, Control-Q, and Control-C. This is meant to mirror MS-DOS where DOS function 8 checks for the Control-Break key combination. In the current version of TOS, however, both of these functions pass all control codes without acting on them.

The other character device input command is used to read characters from the AUX: device, which is the RS-232 serial port. This function waits until a character is completely received before it returns, but does not echo the character to the screen. The function Cauxin( ) is called like this:

```
char ch;
  ch = Cauxin( );
```

where *ch* is the ASCII character received.

One of the problems with using these input functions is that if no character is available from the device, the function will not return until one is available. This leaves your program stuck until the input is received, and if that input doesn't come, it remains stuck forever, forcing you to turn off the computer to regain control. To prevent this situation, the GEMDOS includes status functions that let you determine whether there's a character waiting to be received. These functions are Cconis( ) and Cauxis( ), and their syntax is

```
int status;
  status = Cconis( );
int status;
  status = Cauxis( );
```

where *status* is a flag that indicates whether there is a character waiting. The value returned in *status* is a 0 if there are no characters waiting, and $FFFF (−1) if there is at least one character ready to be received. By calling the status functions, it's possible to determine whether the input functions will return immediately. If the call to the status function shows there are no characters ready, your program may omit the input call, go on to do something else, and then check the input device again later.

GEMDOS also contains a number of functions for writing characters to the character devices. Each output device has its own output function. The first, Cconout( ), is used to send characters to the console screen. Its syntax is

```
char ch;
  Cconout(ch);
```

where *ch* is the ASCII character to write to the screen. Machine language programmers should note that they will pass the character to be printed as a word-length value, the low byte of which contains the character to be printed, and the high byte of which has been cleared to 0. As with the BIOS routine, Bconout( ), VT-52 control characters and escape sequences are treated as commands rather than printed as characters. For a complete list of VT-52 style escape code, see Appendix E.

The character output functions for the other two output devices are very similar. The Cprnout( ) call is used to send characters to the printer, while Cauxout is used to send output to the serial port. The syntax for these calls is:

```
char ch;
int status;
  status = Cprnout(ch);
char ch;
  Cauxout(ch);
```

where *ch* is the character to be sent. Both of these functions will wait for the character to be sent before they return. In the case of the printer, the *status* return will be −1 if the character has been sent correctly. If, for some reason, the character cannot be sent, however (paper out, printer off line, and so on), a value of 0 will be returned in *status* after the time-out period. Note that like Bconin, Bconout doesn't return until the character is actually sent. As with the input functions, you can avoid sending a character to a device that is not ready to receive it by first testing the device's output status. The output status functions are Cconos( ), Cauxos( ), and Cprnos( ), and they all share the same syntax:

```
int status;
  status = Cconos( );
int status;
  status = Cauxos( );
int status;
  status = Cprnos( );
```

These functions all return a 0 in *status* if the device is not ready to accept a character, and $FFFF (−1) if it is ready.

In addition to the normal character input and output functions, GEMDOS provides an unusual function that allows you either to send characters to the console device or receive them. The macro name for this function is Crawio( ), and it's called like this:

```
int chin, chout;
  chin = Crawio(chout);
```

where *chout* is a word whose high byte is 0, and whose low byte contains either a character to be printed, or a flag that signals the function to check the console keyboard for input. If a value from 0–254 is placed in the low byte of *chout,* that character is printed on the console screen at the current cursor position. Control codes and escape sequences are interpreted normally (see Appendix E for a complete description). If the low byte of *chout* contains a value of 255 (0xFF), the function tests the console keyboard for input. If no characters are available for input, both the high and low bytes of *chin* are set to 0. If a character can be read, however, the low byte of *chin* contains the ASCII value of the character, while the high byte contains its scan code (see Appendix J for a complete list of key codes). Thus, the input portion of Crawio( ) combines both status and input functions in a single call. Crawcio( ) does not echo the character that it reads to the screen.

In addition to functions that input or output a single character at a time, GEMDOS contains functions that let you use the console device to input or print a whole string of characters at once. The output function is called Cconws( ), and its syntax is as follows:

```
int length;
char *string;
  length = Cconws(string);
```

where *string* is a pointer to a null-terminated string of text characters of any length. Control characters and escape sequences are interpreted as usual. Upon return, *length* contains the number of characters that were printed.

The read string function, Cconrs( ), is a little more complex, and a little more peculiar. It not only reads in a whole string, echoing each character out to the screen as it is read, but provides some line-editing functions as well. A typical call to this function might look like this:

```
char buffer[82];
int length;
  buffer[0] = 80;
  length = Cconrs(buffer);
  buffer[length+2] = 0;
```

Before calling the function, first set up a buffer, into which the function may read the characters. You must also store the maximum number of characters that may be read in the first byte of the buffer. Here, an 82-byte buffer was declared to hold a maximum of 80 input characters. The extra two bytes are for the string formatting information that appears at the beginning of the buffer. The first byte is the maximum string length, which your program must set, and the second byte is the actual length of the string returned by Cconrs( ). If you plan to output the string, using a function such as Cconws( ), terminate it with a 0 character. To print the line, you could use the statement

```
Cconws(buffer+2);
```

When your program calls Cconrs( ), you may start entering characters from the keyboard. These characters will appear on the screen as you type them. The Cconrs( ) call does not return until the user signals that the entire string has been entered. He does this by pressing one of the terminating key combinations such as Return (carriage return), Control-J (linefeed), or Control-M (carriage return). The function also returns if the user enters the maximum number of characters, as stored in the first byte of the buffer.

While the user is entering characters, certain control characters act as line-editing functions. These are similar to the console device line-editing functions available under the CP/M and MS-DOS operating systems. The table below shows the line-editing characters and their functions:

Control Character	**Editing Function**
Control-C	End input and terminate program
Control-H	Backspace
Control-I	Tab
Control-J	Linefeed, end input
Control-M	Carriage return, end input
Control-R	Skip to next line, reprint original line
Control-U	Skip to next line, start new line
Control-X	Erase to beginning of line

The Control-C key combination not only causes the Cconrs( ) function to return, but also terminates the entire program. This feature makes Cconrs( ) extremely dangerous, and suitable only for quick and dirty programs that you write for your own use. Any program written for use by others should implement its own input routine in a manner that doesn't allow the user to exit the program easily by mistake.

Program 5-1 demonstrates some of the character I/O functions discussed above.

Program 5-1. GCHARDEV.C

```
/***********************************************/
/*                                             */
/*   GCHARDEV.C                                */
/*                                             */
/*   Demonstrates some of the GEMDOS           */
/*   character device functions.               */
/*                                             */
/***********************************************/

#include <osbind.h>    /* For GEMDOS macro definitions */
#define MAXLEN 80      /* maximum input line length */

main()
{
   char buf[MAXLEN+3], ch;
   int len=0;

/* first input a line a character at a time, using Cconin */

   Cconws("Enter a line of text, and hit Return:\n\n\r");

   do /* keep getting characters */
   {
   if ( (ch=Cconin()) != 8) /* if this isn't a backspace, */
      buf[len++]=ch;        /* add the character to the buffer */
   else
      {                     /* if it is a backspace, */
      Cconws(" \010");      /* rub out last character...*/
      if (len>0)            /* and if there are any chars to delete */
         len--;             /* delete one */
      }
   } while( (ch!=13) && (len<MAXLEN) ); /* do til CR or end of line */


   buf[len]=0; /* terminate line with ASCII */
   Cconws("\n\nYour line of text was:\n\r");
   Cconws(buf);
   printf("\nAnd it was %d characters long\n\n\n",len);

/* Now, input the whole line at a time */

   Cconws("Enter another line of text, and hit Return:\n\n\r");
   buf[0]=MAXLEN;
   len =Cconrs(buf);
   buf[len+2]=0;
   Cconws("\n\nYour line of text was:\n\r");
   Cconws(buf+2);
   printf("\nAnd it was %d characters long\n",len);

/* if you're running this from the Desktop, you may want to
   add a pause to give the user a chance to read the output */
   Cconws("\n\n\rPress any key to end");
   Cconin();
}

/******** end of GCHARDEV.C *****/
```

System Functions

In addition to character device functions, GEMDOS also contains a number of system control functions. These include routines to manage system memory, to execute and terminate programs, and to get and set the DOS clock and calendar. The first of these allow you to change the 68000 processor's privilege mode, or to find what the current mode is. The 68000 allows certain operations to be performed in supervisor mode that cannot be performed from the normal user mode. On the ST, these include reading or writing to system variables stored in memory locations below 2048 ($800), and reading or writing to hardware registers located above 167444482 ($FF80000). The macro name of the function used to switch modes is Super( ), and its syntax is:

```
long stack, oldstack;
  oldstack = Super(stack);
```

The meaning of *stack* will vary depending on what you want the function to do. If you set *stack* to −1L (0xFFFFFFFF), the function returns a code in *oldstack* which specifies the current privilege mode. If the value returned in *oldstack* is 0, the processor is user mode. If the value returned is 1, it's in supervisor mode.

If *stack* is set to any value other than −1L, the function will toggle the current privilege mode. If the processor is in user mode, it will be set to supervisor mode, with the supervisor stack set to the address passed in *stack*. If you wish to make the supervisor stack address the same as that of the user stack, pass a value of 0L in *stack*. The address of the previous supervisor stack is returned in *oldstack*. This value should be saved, so you may restore the old supervisor stack value when you switch back to user mode.

If the function is called when the processor is in supervisor mode, GEMDOS sets it to user mode and sets the supervisor stack value back to the address passed in *stack*. This address should be the same one returned in *oldstack* when you first sent the processor into supervisor mode. It's important that the supervisor stack exist in memory outside the control of your program, so when your program terminates, the system still has a workable stack area. If you fail to set the supervisor stack back to its original value before your program terminates, you may crash the system. An example of proper use of the Super( ) function may be found in the sample program TOGGLE.S, below.

Memory Management Functions

The management of free memory is another common system function made easier by the DOS commands. When a GEM or TOS application program is first loaded and run, it takes control of the entire application RAM space. As you will see below, however, it may want to give back some of that memory so that other applications, desk accessories, or resident programs may use it. Because an application can never know how much free memory will be available, or where that free memory is, the proper way to gain access to additional memory space is through the system memory manager. When a program wants a hunk of free memory in which to store data the user has input, or that it has retrieved from a disk file, it requests the memory manager to allocate a certain number of bytes. If that amount of memory is available, the memory manager removes it from the free list and passes its starting address to the application. When the application is finished with the memory, it tells the memory manager, which returns it to the free pool.

The GEMDOS function used to allocate free memory is known by the macro name Malloc( ). This function is called as follows:

```
long address, bytes;
  address = Malloc(bytes);
```

where *bytes* specifies the number of bytes of memory to allocate. If *bytes* is set to −1L (0xFFFFFFFF), the function simply returns the size of the largest block of free memory in *address*. Otherwise, the function tries to allocate the amount of memory specified. If it is able to allocate the requested memory block, it returns the starting address of the block in *address*. If there isn't sufficient free memory to allocate the block that was requested, a value of 0 is returned instead.

When the program is finished using the memory it has allocated, it should return it to the system by using the Mfree( ) call

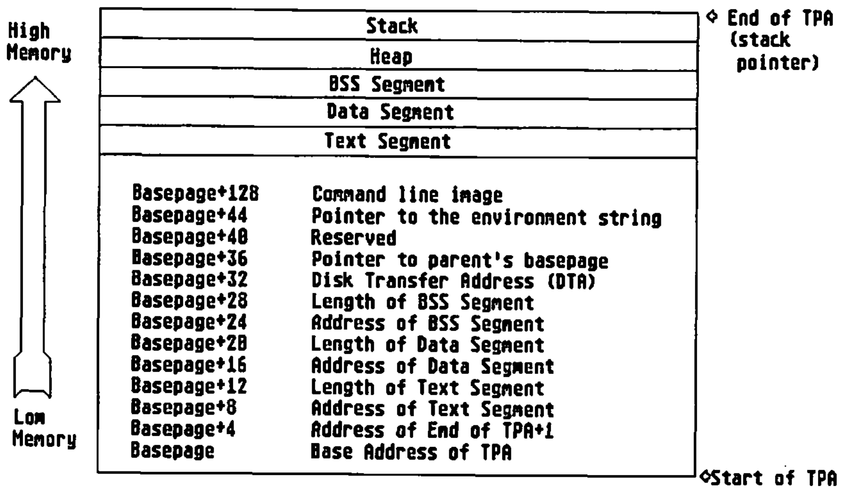
```
int status;
long address;
  status = Mfree(address);
```

where *address* is the starting address of a block of memory allocated with Malloc( ). If the memory is successfully returned to the system, a 0 is returned in *status*. A negative value in *status* indicates that an error occurred. If a program fails to release one or more blocks of memory, GEMDOS will automatically return them to the free pool when the program terminates.

Atari documentation states that if more than 20 blocks of memory are allocated by Malloc( ) at one time, the memory management system will fail. Since some library functions like fopen( ) use Malloc( ) to allocate blocks for their own use, your program should use Malloc( ) very sparingly. If your program needs more than one memory area allocated, it is best to get one big block of memory and divide it up yourself, rather than requesting many small areas from Malloc( ).

As stated above, when a program is loaded and run, it takes control of the entire Transient Program Area (TPA), which consists of all of the memory from the program's Base Page Address on up to the top of free RAM. This area includes the base page, which stores information about the process, the program code, the program data, and program variable storage areas. The composition of the TPA is detailed in Figure 5-1 below.

Figure 5-1. Composition of the Transient Program Area



Basepage+128	Command line image
Basepage+44	Pointer to the environment string
Basepage+40	Reserved
Basepage+36	Pointer to parent's basepage
Basepage+32	Disk Transfer Address (DTA)
Basepage+28	Length of BSS Segment
Basepage+24	Address of BSS Segment
Basepage+20	Length of Data Segment
Basepage+16	Address of Data Segment
Basepage+12	Length of Text Segment
Basepage+8	Address of Text Segment
Basepage+4	Address of End of TPA+1
Basepage	Base Address of TPA

This diagram shows that the memory area not actually used by the program is divided between the stack and the heap. The stack is a temporary storage area used by programs for passing parameters, saving register contents, and saving subroutine return addresses. The heap is the free memory pool from which Malloc( ) doles out memory blocks. The stack starts at the top of free memory and works its way down, while the heap starts at the bottom of free memory and works its way up.

Most programs don't need to retain control of all free memory. They only need enough heap space for the Malloc( ) calls they make, and enough stack space to cover temporary storage needs. The rest may be returned to the system pool of free memory. Good programming practices dictate that your program only retain control of the memory it needs. In addition, there are a couple of specific reasons for returning extra memory. If your program uses any of the GEM library calls, it should return at least 8K of memory for use by GEM. If your program uses the Pexec( ) function to run another program (see below), there must be sufficient free memory for that other program and its data.

The function used to shrink a program's memory block is called Mshrink( ), and it's called like this:

```
int status;
long address, size;
  status = Mshrink(0, address, size);
```

where *address* is the starting address of the memory block to retain, and *size* is the number of bytes to retain. Note that you must pass a 0, 16 bits in length, as the first parameter. If the operation is successful, a value of 0 is returned in *status*. If it is unsuccessful, an error code of −40 (invalid memory block address) or −67 (memory block growth failure) will be returned.

To use Mshrink( ), determine where the starting address of program memory is, and how much memory the program occupies. Finding the starting address of program memory isn't too difficult—when you start a program, the second word on the stack points to that location.

Finding the size of the program requires a little more knowledge of how program storage space is allocated. As mentioned above, the memory area in which a program resides is known as the Transient Program Area (TPA), and at the beginning of the TPA is a 256-byte segment known as the *basepage*. As shown in Figure 5-1, the basepage contains information about the size and address of each program segment, as well as the command line that is passed to the program (these are the extra characters you type in when you run a TOS Takes Parameters program whose name ends in .TTP).

The actual program code comes after the basepage , followed by the data area, and the BSS (block storage segment) which is used to store uninitialized data. To determine the total size of the program area, look in the basepage to find the size of the code and add to that the size of the data and BSS segments, along with the size of the basepage itself. Since you need a stack and heap area for the program, it makes sense to add the sizes of these segments to the end of the program and reserve the combined program, heap and stack area together. Once you calculate the size of this area, set the stack pointer to the top of program memory and make the Mshrink( ) call. Once that's done, continue with your program.

Program 5-2, a program fragment, shows how to start an application program that needs to give back some of the TPA memory, either because it uses GEM calls or uses Pexec( ) to run another program.

Program 5-2. MSHRINK.S Program Fragment

```
************************************************************
*                                                          *
*  MSHRINK.S  -- Shows how to begin a program by           *
*  shrinking the memory used for TPA.                      *
*                                                          *
************************************************************

*** Program equates

bpadr    =     4  * Stack offset to base page address
codelen  =    12  * Base page offset to Code segment length
datalen  =    20  * Base page offset to Data segment length
bsslen   =    28  * Base page offset to BSS  segment length
stk      = $4000  * size of our stack and heap area (16K)
bp       =  $100  * size of base page

*** Program starts here.  Get base page address in a5

   .text
   move.l   a7,a5           * dupe a7 so you can get the base  page address
   move.l   bpadr(a5),a5    * a5 now = basepage address

*** Calculate the total amount of memory used by
*** your program (including stack space) in d0

*                           * total memory used =
   move.l   codelen(a5),d0  *   length of code segment
   add.l    datalen(a5),d0  * + length of data segment
   add.l    bsslen(a5),d0   * + length of uninitialized storage segment
   add.l    #stk+bp,d0      * + (size of base page + stack/heap)

*** Calculate the address of your stack
*** and move it to the stack pointer (a7)

*                         * new stack address =
   move.l  d0,d1          *   size of program memory
   add.l   a5,d1          * + program's base address,
   and.l   #-2,d1         * pick off odd bit to make sure that the
*                         * stack starts on a word boundary (it must).
   move.l  d1,a7          * set stack pointer to your stack
*                         * which is stk bytes above end of BSS

*** Use the GEMDOS Mshrink() call to reserve the area of memory
*** actually used for the program and stack, and release the
*** rest back to the free memory pool.

   move.l d0,-(sp)        * push the size of program memory
*                         * (first Mshrink() parameter) on the stack.
   move.l a5,-(sp)        * push the beginning address of the
*                         * program memory area (2nd Mshrink() parameter).
   clr.w  -(sp)           * clear a dummy place-holder word
   move   #$4a,-(sp)      * finally, push the GEMDOS command number
*                         * for the Mshrink() function
   trap   #1              * call GEMDOS
   add.l  #12,sp          * and clear your arguments off the stack.
```

Process Control

One of the Disk Operating System's major functions is to load a program (also known as a *process*) and start it. After the program finishes, GEMDOS also has the responsibility for terminating the process, reclaiming its memory, and returning to the GEM Desktop (or whatever other application happens to be operating as a command shell). GEMDOS includes four process control functions, one for loading and executing a process, and three for terminating one.

The function used to load a program file and execute it is called Pexec( ). The syntax for this function is

***char file, command, env;**
int mode;
long status;
  status = Pexec(mode, file, command, env);

The meaning of the function parameters varies according to the value of *mode*. There are four different modes of operation for the Pexec( ) function. These are:

Table 5-1. Four Modes for Pexec( ) Function

Mode Number	Function	File	Command	Env
0	Load and execute	Pointer to filename string	Pointer to command string	Pointer to environment string

Table 5-1. Four Modes for Pexec( ) Function (continued)

Mode Number	Function	File	Command	Env
3	Just load, do not execute	Pointer to filename string	Pointer to command string	Pointer to environment string
4	Just execute	Unused	Basepage address	unused
5	Create basepage	Unused	Pointer to command string	Pointer to environment string.

The mode used most often is 0, load and go. In this mode, *file* is a pointer to a string containing the complete path name of the program file to load and execute. This string is a series of ASCII characters, ending with ASCII character 0 (NUL). Example path names are PROGRAM.PRG, C:\PROGRAMS\PRG1.TOS and B:\MYPROG. Path names are discussed more fully in Chapter 6, which deals with the GEMDOS file system. The *command* parameter contains a pointer to a command tail string. This type of string is used with a .TTP (TOS Takes Parameters) type program to specify more fully what the program should do. For example, a program called COPY.TTP that copies one file to another might be executed with the command tail A:FILE1 B: FILE2, which tells the program to copy a file named FILE1 on the A: drive to a file called FILE2 on the B: drive.

The command tail string is not the usual C language string, that consists of a series of ASCII characters ending in NUL. Instead, it is more like strings used in Pascal, where the first character of the string is a binary number specifying the length of the string and the rest of the string is composed of the actual ASCII characters. Such a string does not normally end in an ASCII 0 character. The final parameter for a mode 0 Pexec( ) call, *env,* is a pointer to the process' environment string. This is a series of null-terminated ASCII strings with an additional ASCII 0 character at the end of all of the strings. The environment string is used by some programs to get information about the program environment. A typical environment string might be PATH = C:\, which lets the program know the default directory to use to search for data files. If a 0 is passed in the *env* parameter, the new process receives as its environment string a copy of the environment string used by the program that called it.

When mode 0 of Pexec( ) is used, the call loads the new program, sets up its basepage, passes the arguments and environment to it, and executes it. This *child process,* as it's called, inherits the *parent's* standard file descriptors (handles 0–5; see Chapter 6 for more information on standard file handles). When the program ends, the call returns an exit code from the program (if it passes one) in *status.* A negative value in *status* indicates that the program could not be executed (because there wasn't enough memory to run it, or some other reason). Error returns in *status* are negative longwords, while return codes passed from the child process are word length, with 0s in the upper 16 bits of the longword.

Modes 3 and 4 split the functions of mode 0. Mode 3 of Pexec( ) takes the same parameters as mode 0, but it just loads the specified file, sets up its basepage, and returns a pointer to the basepage in *status.* It does not execute the process. To run the program after it has been loaded with mode 3, you may use mode 4 of Pexec( ). This mode uses the pointer to the program's basepage that was returned by mode 3 as its only parameter. This pointer is passed in *command.* The last mode of Pexec( ), mode 5, allocates the largest free block of memory and creates most of a basepage for it. Some basepage values, such as the size and address of text, data, and BSS section, obviously can not be filled in.

Program 5-3 shows how to load and execute a separate program from within the current one. It demonstrates the basic principal used by *command shell*-type programs, which provide an MS-DOS command-line environment for the ST.

Program 5-3. GPEXEC.C

```
/*********************************************/
/*                                           */
/*  GPEXEC.C                                 */
/*                                           */
/*  Demonstrates use of the GEMDOS           */
/*  Pexec() function to run other programs.  */
/*                                           */
/*********************************************/

#include <osbind.h>    /* For GEMDOS macro definitions */

char file[81];    /* file name buffer */
char buffer[83]; /* input line buffer */
char *command;    /* pointer to command tail string */
```

```
main()
{
   int status;
   int len, done=0;
   int index=1;

 /* Get command line */

   Cconws("Enter command line [include filename extension]\n\r");

   buffer[0]=80;   /* prepare buffer for command string entry */
   buffer[82]=0;
   Cconws("> ");            /* prompt */
   len = Cconrs(buffer);    /* get command string */
   Cconws("\n\n\r");        /* skip a couple of lines */

/* Split it into command file and command tail */

   while (!done)   /* check for 1st space character */
   {
   if (buffer[++index]==' ')  /* if it's there, end */
      done = 2;
   if (index == (len+1))      /* if you're at end of string, end */
      {
      done=1;
      index++;                /* move index past last character */
      }
   }

buffer[index]=0;          /* replace separator with ASCII zero */
strcpy(file,buffer+2);    /* put filename in buffer */

if (done==1)             /* if there wasn't a command tail */
   command=0L;           /* set command address to zero */
else                     /* if there was, put tail length ...*/
   {
   buffer[index] = len-strlen(buffer+2); /* at beginning of string */
   command=buffer+index;                 /* and set pointer */
   }

status = Pexec(0, file, command, 0L);   /* execute the program */
printf ("Command status = %lx\n",status);

} /* end of main */


/******** end of GPEXEC.C *****/
```

Again, note that the startup segment (GEMSTART.O on the Alcyon compiler, or its equivalent on other compilers) must return enough memory with Mshrink( ) to make room for the program to be loaded.

The functions used to terminate a process are mostly of interest to machine language programmers, since the startup modules supplied with C language compilers automatically call one of these functions when the C program terminates. The first and simplest of the functions used to terminate a process is called Pterm0( ). Its calling syntax is

Pterm0( );

This terminates the current process, closes all open files, releases any allocated memory, and exits with a return code of

0. To pass a return code other than 0 to the calling process, you must use the function Pterm( ):

int retcode;
  Pterm(retcode);

where *retcode* is a binary number you wish to pass to the calling program. This code may be used to inform the calling program of the results of the child process.

The last process termination function, Ptermres( ), is used for a special class of programs known as Terminate-and-Stay-Resident programs (TSR). These programs remain loaded in memory even after they terminate. TSR programs can steal the vectors used by exception handlers, such as the system timer, vertical blank interrupt, or the keyboard Alternate-Help screen dump routine, to create *pop-up* or *hot-key* applications. Ptermres( ) is called like this:

long keepsize;
int retcode;
  Ptermres(keepsize, retcode);

where *keepsize* indicates the number of bytes of memory to keep resident, starting at and including the 256-byte basepage. The *retcode* value is the exit code that the program passes back to the parent process when it terminates. Memory allocated by the program with Malloc( ) will not be returned when the process terminates, but open files are closed upon termination of the process.

Program 5-4 is a machine-language program that shows how to steal the screen-dump vector at location $502, and to install your own program, which will execute every time you press the Alternate-Help key combination. For the sake of simplicity, this program toggles the key-click sound on and off. The first use will turn off the clicking noise that you hear when you press a key, and the next use will turn it back on.

Program 5-4. TOGGLE.S

```
*************************************************************
*                                                           *
*     TOGGLE.S -- Demonstrates the use of the GEMDOS        *
*     Ptermres() function from machine language, to         *
*     install a RAM-resident utility.  This short           *
*     example toggles the key-click sound when you          *
*     press Alternate-Help.                                 *
*                                                           *
*                                                           *
*************************************************************
```

```
*** Program equates

conterm  = $484
scr_dump = $502

*** Program starts here

.text

*** Branch to install code at end of program

start:
    bra.w   init

*** The ID here lets us check to see if the resident program was
*** already installed once, so you don't try to do it again if the
*** program is run more than once.
ID:
    .dc.b 'TOGGLE'

*** This code is executed only when Alternate-Help is pressed.
*** you're in supervisor mode here.

toggle:
   movem.l   d0-d7/a0-a7,-(a7)  * save registers
   move      sr,-(a7)           * and status register
   ori.w     #$700,sr           * bump the Interrupt Priority Level
*                               * to 7 (so you don't get interrupted)

   eori.b     #1,conterm    * bit 0 of conterm controls key click

keyclr:                     * clear keyboard so you don't repeat
   clr.w    -(a7)           * clear all keyboard shift bits
   move.w   #$B,-(a7)       * shift bits function
   trap     #$D             * call BIOS
   addq.l   #4,a7           * clean stack

   move     (a7)+,sr            * restore status register
   movem.l  (a7)+,d0-d7/a0-a7   * restore rest of registers
   rts                          * and end toggle routine

**** Initialize and install routine--this is only done once
**** and then this code is thrown away

init:

*** Print sign-on message

   pea      msg1          * push address of sign-on string
   move.w   #$9,-(a7)     * Cconws
   trap     #$1           * call GEMDOS
   addq.l   #6,a7         * pop args from stack

*** Switch to Supervisor mode to read protected memory

   clr.l    -(a7)         * switch to super mode, same stack
   move.w   #$20,-(a7)    * Super function
   trap     #$1           * call GEMDOS
   addq.l   #6,a7         * clean up stack
   move.l   d0,oldstack   * save old stack address

*** Check to see if this program has been installed once.
*** If it hasn't, install it.  If it has, end

   movea.l  scr_dump,a0   * get screen dump vector
   subq.l   #6,a0         * move back 6 characters
   lea      ID,a1         * check to see if you're already installed
   move.w   #$5,d0        * 6 letters in ID

checkid:
   cmpm.b   (a0)+,(a1)+     * compare ID to resident program
   bne.b    install         * if ID doesn't match, install toggle
   dbf      d0,checkid      * if it does match, ...

   move.l   oldstack,-(a7)  * push old stack address
   move.w   #$20,-(a7)      * out of super mode
   trap     #$1             * call GEMDOS
   addq.l   #6,a7           * clean stack
```

```
    pea       msg2           * push "already installed" msg address
    move.w    #$9,-(a7)      * write it (CCONWS)
    trap      #$1            * call GEMDOs
    addq.l    #6,a7          * clean stack

    bsr.w     delay          * wait a bit to let them read message
    clr.w     -(a7)          * end program
    trap      #$1

install:

    move.l    #toggle,scr_dump  * install toggle in screendump vector
    move.l    oldstack,-(a7)    * push old stack
    move.w    #$20,-(a7)        * out of super mode
    trap      #$1               * call GEMDOS
    addq.l    #6,a7             * clean stack
      pea       msg3            * print install message
      move.w    #$9,-(a7)       * CCONWS
      trap      #$1             * call GEMDOS
      addq.l    #6,a7           * clean stack

      bsr.w     delay           * wait a bit to let them read it

      clr.w     -(a7)           * terminate and stay resident
      move.l    #init,d0        * take address of end of resident part
      sub.l     #start,d0       * -start address of program
      add.l     #$100,d0        * plus 256 bytes for basepage
      move.l    d0,-(a7)        * push on stack
      move.w    #$31,-(a7)      * ptermres
      trap      #$1             * call GEMDOS to end program

   *** delay subroutine--just loop around a while

   delay:
      move.w    #$20,d0         * loop about (32 * 65536) times
   dloop:
      dbf       d1,dloop
      dbf       d0,dloop
      rts

   *** Text data for messages

   .data

   msg1:
      dc.b    $1B,'E'
      dc.b    'KEYCLICK TOGGLER..',$0D,$0A
      dc.b    'Replaces Alternate-Help screen dump',$0D,$0A,$00

   msg2:
      dc.b    $07,'Already '

   msg3:
      dc.b    'Installed',$0D,$0A,$00,$00

   *** Temporary storage for old user stack pointer
   .bss

   oldstack:
      ds.l    1

      end
```

Time and Date Functions

GEMDOS keeps track of the date and time to date stamp a file, showing when it was created or last updated. The GEMDOS functions used to get and set the DOS time and date are very similar to those provided by the XBIOS to get and set the IKBD controller's time and date. The main difference is that separate functions are provided here to set the time and date, while the XBIOS functions get or set both at once. The DOS clock and IKBD clock are not necessarily the same, although in later versions of TOS (with ROMs that support the blitter chip), the DOS clock is reset from IKBD clock at the termination of each process.

The functions used to get and set the DOS time are known as Tgettime( ) and Tsettime( ), respectively. These functions are called as follows:

```
int time;
  time = Tgettime( );
int time;
  Tsettime(time);
```

where *time* is a 16-bit code that indicates the time to set with Tsettime( ), or the time returned by Tgettime( ). In either case, the meaning of this code is interpreted as follows:

Bit Number	Description	Range
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23

Likewise, the DOS functions used to get and set the date are called Tgetdate( ) and Tsetdate( ):

```
int date;
  date = Tgetdate( );
int date;
  Tsetdate(date);
```

where *date* is a 16-bit code that specifies the date to set with Tsetdate( ), or the date read with Tgetdate( ). The meaning of this code is:

Bit Number	Description	Range
0–4	Day	1–31
5–8	Month	1–12
9–15	Year	0–119*

* Year value is added to 1980 to get current year.

Since the format of the time and date information stamped on disk files is the same as the above, you can use the code provided in the program GDIR.C in Chapter 6 as an example of how to extract the various fields of information from their bit-packed storage format.

GEMDOS also provides a function that returns the GEMDOS version number. This number refers only to the version of GEMDOS, not to the GEM or TOS version in general. The function used to find the GEMDOS version number is Sversion( ), and it's called like this:

```
int version;
  version = Sversion( );
```

where *version* is a 16-bit code which indicates the GEMDOS version number. This number is stored in the same format as MS-DOS uses for its version number, with the major version number in the low byte, and the minor revision number in the high byte. The first version of the TOS ROMs all show the GEMDOS version to be 0x1300, or version 0.19.

Chapter 6

GEMDOS File System Functions

In addition to the character device and system functions, many of which are similar to calls found in the BIOS and XBIOS, GEMDOS provides a number of unique functions involved with the disk filing system. Disk drives are unique among the I/O devices because they are random-access devices. Each disk contains a fixed amount of storage space and can retrieve information from anywhere on the disk. Rather than reading the information as one continuous stream, the disk starts at the beginning and works its way to the end each time. Disk drives are also unique because they are divided into both physical storage units (*tracks* and *sectors*) and logical storage units (such as *directories* and *files*).

A blank disk contains a magnetic medium with nothing significant recorded on it. When you format the disk, the drive controller encodes information on the disk that divides it into tracks and sectors. On the ST, each sector holds 512 bytes of data. There are normally nine sectors per track, and 80 tracks per side of the disk. This means that single-sided disks can hold 368,640 bytes (512 × 9 × 80) of data, and double-sided drives can store twice that amount.

Although nine tracks per sector and 80 sectors per side is the default format for disks, it is possible to format a disk with ten or even eleven sectors per track, and 81 or 82 tracks per side. Because of variations in manufacturing tolerances from drive to drive, the default values are the safest ones to use, particularly for disks that will be used on more than one disk drive.

Dividing the disk into tracks and sectors in effect gives you hundreds of thousands of little boxes, each of which may hold one character of information. It would not be practical, however, to expect the user to keep track of what information is in each box. Can you imagine having to tell the computer the letter you wrote with your word processing program is stored at locations 40,658 to 41,949? To avoid this problem, GEMDOS goes farther than merely dividing the disk into tracks and sectors. It also divides it into logical units known as files and directories. This filing system allows you to give a name to a collection of storage locations on the disk. Thereafter, whenever you wish to read or write to that collection of data, you can refer to it by that name, and GEMDOS takes care of keeping track of the actual physical storage locations to which the name refers.

To implement this filing system, GEMDOS must store several internal data structures on each disk, allowing it to keep track of the disk organization, filenames, and locations occupied by each file. These data structures are nearly identical to those used by MS-DOS, and as a result, it is possible to read MS-DOS format disks on the ST. The first of these structures, known as the Boot Sector, is located on the first sector on the disk (Track 0, Sector 1). The boot sector performs a couple of different functions. First, it lets GEMDOS know how the disk is organized so it knows how to read the data stored on it. The ST supports single-sided 3½-inch floppy drives, double-sided 3½-inch floppy drives, 5¼-inch floppies, and hard drives, and these drives can be formatted with varying numbers of sectors per track, and tracks per side. If it weren't for the boot sector, GEMDOS wouldn't know where to begin looking for the data. Much of this information is the same as that stored in the BIOS Parameter Block, which may be retrieved with the BIOS Getbpb( ) command. The second function of the boot sector is to allow disks to be *bootable*—that is, to take control and run a certain program as soon as the computer is turned on. It does this with a program fragment called the *boot code*. The boot code is usually just a tiny program that loads another larger program from disk into the computer memory and turns control over to that program.

The organization of the Boot Sector is shown in Table 6-1.

Table 6-1. Organization of the Boot Sector

Byte Number	Description
0–1	If disk is bootable, a BRA.S instruction to boot code starting at byte 31 is stored here. If disk is not bootable, two 0 bytes are stored here.
2–7	Reserved for OEM use (unused on ST)
8–10	Volume serial number 24 bits long (used to determine is a disk was changed)
11–12	Number of bytes per sector (must be 512 under current GEMDOS)
13	Number of sectors per cluster (must be two under current GEMDOS)
14–15	Number of reserved sectors at the start of media (including the boot sector)
16	Number of File Allocation Tables (FATs) on disk
17–18	Number of entries in the root directory
19–20	Total number of sectors on disk
21	Media Descriptor (not used on ST)
22–23	Number of sectors used by each File Allocation Table (FAT)
24–25	Number of sectors per track (normally 9)
26–27	Number of disk heads (one for single-sided, two for double-sided)
28–29	Number of hidden sectors (not used on ST)
30–509	Boot code (if any)
510–511	Checksum

Directory Blocks

GEMDOS also requires the *root directory* to keep track of named files. This directory contains a number of 32-byte directory entries, each one having information about a specific file, such as its name, its file attributes, the time and date of its creation, its length, and the file's starting cluster number. The starting cluster number helps GEMDOS find the first sector used to store the file's data.

When GEMDOS allocates disk space for a file, it does so in increments called *clusters*. The current cluster size is two sectors, which means that even if a file is only 10 bytes long, it still occupies 1024 bytes (two sectors) on disk. Once GEMDOS knows the starting data cluster for a file, it can find the rest of the file by using the File Allocation Table (FAT).

The root directory of a floppy disk has space for 112 directory entries. The format of each 32-byte entry is shown in Table 6-2.

Table 6-2. Format for Directory Entry

Bytes	Contents
0–7	Eight-character primary name (ASCII text) (if file is deleted, first character is $E5)
8–10	Three-character extension (ASCII text)
11	Attribute byte, contains following bit flags Bit 0 = read-only file (can't be deleted or written to) Bit 1 = hidden file (excluded from normal directory searches) Bit 2 = system file (excluded from normal directory searches) Bit 3 = volume label (can only exist in root) Bit 4 = subdirectory Bit 5 = archive bit Bit 6 = reserved Bit 7 = reserved
12–21	Reserved for future use
22–23	Date of creation Bits 0–4 = Day of month (1–31) Bits 5–8 = Month (1–12) Bits 9–15 = Year (−1980)
24–25	Time of creation Bits 0–4 = Seconds divided by 2 (0–29) Bits 5–10 = Minutes (0–59) Bits 11–15 = Hours (0–23)
26–27	Starting cluster number (in 8088 order, low-byte first)
28–31	File length in bytes (in 8088 order, low-byte first)

Subdirectories are structured as files that contain directory entries. Since these files may expand in size like any other data files, there is no limit to the number of entries in a subdirectory, as there is with the root directory. The first two entries in a subdirectory are always dot (.), which stands for the current directory and dot-dot (..), whose entry points to the first cluster in the parent directory (or is 0 if the parent is the root directory).

File Allocation Tables (FATs)

The last internal data structure used by GEMDOS is known as the *File Allocation Table,* or FAT. The FAT is used to keep track of which clusters belong to which file. The system works like this: For each cluster on the disk, there's a corresponding FAT entry. The directory entry for each file contains a pointer to the first file cluster. When you look up the FAT entry for that first cluster, you get a pointer to the next cluster. The FAT entry for that cluster contains a pointer to the next one, and so on until you get to a FAT entry that contains a last-cluster marker.

FAT entries come in two sizes. Floppy disks use FAT entries that are 12-bits in length, while hard disks use 16-bit entries. You can tell which kind of FAT entries are used for a particular disk by looking at the bflag byte of the BIOS Parameter Block (see the discussion of the Getbpb( ) function in Chapter 2). The meaning of each type of FAT entry is shown in Table 6-3.

Table 6-3. Meaning of FAT Entries

12-Bit Value	16-Bit Value	Description
$000	$0000	Free cluster (unused)
$001	$0001	Invalid value
$002–$FEF	$0002–$7FFF	Next cluster number
	$8000–$FFEF	Invalid value
$FF0–$FF7	$FFF0–$FFF7	Bad cluster
$FF8–$FFF	$FFF8–$FFFF	End of file

To convert cluster numbers to logical sector numbers, subtract 2 and multiply by the number of sectors per cluster (2). Thus, if the directory entry for the first file shows that it starts at cluster 4, it tells you that the first data block for that file starts at sector 4 of the data area (which starts with the first sector of track 2). To find the next data block for the file, look up the FAT entry for cluster 4, which gives you the cluster number for that block. When you get a FAT entry of $FF8 ($FFF8 for 16-bit FATs), you know you've reached the last block of the file.

The FAT is an absolutely critical part of the disk. If the sectors containing the FAT become unreadable, the file system will not be able to follow the directory chain, and will not be able to read the files on the disk. As a security measure, two FAT areas are maintained simultaneously, so if one becomes unreadable, the other may be used to find the files.

Data Area

The last logical division of the disk is called the *data area* or *files area.* The sectors in this area of the disk are viewed as clusters of two sectors each. As a file is created (or extended), GEMDOS searches the FAT for free clusters and assigns as many of them to the file as necessary.

The physical location of the various disk data structures may be determined by values found in the BIOS Parameter Block. The boot sector(s) occupy logical sectors 0 through (fatrec − fsiz − 1). The first FAT starts at (fatrec − fsiz). The second FAT starts at fatrec. The root directory starts at (fatrec + fsiz). For a nonbootable floppy, the first two tracks are used for these data structures. Sector 1 of track 0 holds the boot block. The first FAT occupies sectors 2–6 of track 0, while the second FAT is split between sectors 7–9 of track 0, and sector 1 and 2 of track 1. The root directory takes up sectors 3–9 of track 1. The rest of the disk, starting with track 2, is made up of data clusters, with two sectors per cluster.

File I/O Functions

The most fundamental of the file system functions are those used to create a file, write data to it, and read data back from the file. The GEMDOS functions used for these purposes are modeled after Unix-style file commands. In fact, if you program in C you'll probably notice that the macro names for the GEMDOS file functions differ only in capitalization from those of the C compiler's own library functions. Thus, the GEMDOS function Fread( ) is roughly equivalent to, but not exactly the same as, the C compiler's own fread( ) function.

The GEMDOS file functions identify the file upon which they operate by a number known as a *file handle*. There are three types of file handles (see table below). The first type (handles −3 through −1) belongs to the character devices. The console device (consisting of the screen and keyboard), the serial device, and the parallel printer device may all be treated like disk files for I/O purposes. This makes it possible to redirect file output to a device like the printer (see the section of Fdup( ) and Fforce( ), below). The next type of handle (0–5) is reserved for standard system files. These are modeled after devices that MS-DOS makes available to any program. MS-DOS initializes handles 0–4 to point to standard devices that can be used for input, output, error reports, and listings. While GEMDOS reserves the same range of handles for its own standard devices, it doesn't automatically initialize them. This is usually performed by the C compiler, which at least assigns a standard input and output file (sometimes an error and list device as well). When a program spawns a child process with Pexec( ), that child inherits the parent's standard files. The third type of file handle is assigned to user files on a temporary basis. When a file is created or opened, a small positive number greater than five is assigned to it as temporary ID number for the file. The file handle is used to identify the file for the purposes of any subsequent GEMDOS file operation. When the program has no more operations to perform on a particular file, it can close it, which relinquishes the handle and allows it to be reassigned. All files are closed automatically when the process that opened them terminates, or when the media which contained them is replaced.

Handle Number	Device or File Assignment
−1 (0xFFFF)	CON: (console device)
−2 (0xFFFE)	AUX: (RS-232 serial port)
−3 (0xFFFD)	PRN: (Centronics parallel port)
0	Standard input (usually CON:)
1	Standard output (usually CON:)
2	Standard error
3	Auxiliary
4	Standard list
6 and up	User disk files

The first step in writing a file is to create it. The GEMDOS call used for this function is Fcreate( ):

```
char *fname;
int handle, attr;
  handle = Fcreate(fname, attr);
```

where *fname* is a pointer to a null-terminated ASCII string containing the name of the file to create. The name can be a simple file name (such as LETTER.DOC), or a complete path name (like C:\WORDPROC\LETTERS\BILL.DOC). The *attr* value is a flag that specifies the file's attributes. The meaning of the flag bits is described in Table 6-4.

Table 6-4. Meaning of Flag Bits Used in File Creation

Bit Number	Bit Value	Description
0	1	Read-only file (can't be deleted or written to)
1	2	Hidden file (excluded from normal directory searches)

Table 6-4. Meaning of Flag Bits Used in File Creation (continued)

Bit Number	Bit Value	Description
2	4	System file (excluded from normal directory searches)
3	8	Volume label (can only exist in root)

If the file didn't exist previously, Fcreate( ) both creates a directory entry for the new file and opens it for writing only. If the function succeeds, the file handle of the new file is returned in *handle*. If not, a negative GEMDOS error number is returned, such as −34 (Path Not Found), −35 (No Handles Available), or −36 (Access Denied). If the file existed previous to the Fcreate( ) call, it will be truncated to a length of 0 before it is opened, which destroys its previous contents.

After the file has been created, it can be written to with the command Fwrite( ):

```
long status, bytes;
int handle;
char *buffer;
  status = Fwrite(handle, bytes, buffer);
```

where *handle* is the file handle returned when the file was created or opened, *buffer* is a pointer to the buffer that holds the data to write to the file, and *bytes* indicates the number of bytes to transfer from the buffer to the file. If the write is successful, the number of bytes actually written is returned in *status*. If unsuccessful, a GEMDOS error number is returned instead.

When all of the data is written to the file, the file must be closed with the function Fclose( ):

```
int handle, status;
  status = Fclose(handle);
```

where *handle* is the file handle of the file to close. If the operation succeeds, 0 is returned in *status*. Otherwise, it contains the appropriate GEMDOS error number. Close a file when you are finished with it because it isn't possible to open the file for reading or writing again until it's closed.

To read or write to an existing file, you must first open it. The GEMDOS command used for this purpose is Fopen( ), and its calling syntax is as follows:

```
char *fname;
int handle, mode;
  handle = Fopen(fname, mode);
```

where *fname* is a pointer to the null-terminated ASCII filename of the file to open, and *mode* is a flag that specifies which operations will be available once the file has been opened. Possible values for *mode* include:

Mode Number	Operations
0	Read only
2	Write only
3	Read or write

If the file can be opened, a file handle is returned in *handle*. If it can't be opened (for instance, if GEMDOS can't find some element of the path name), the appropriate GEMDOS error number will be returned. For a complete list of GEMDOS errors, see Appendix D.

Once a file has been opened for reading and/or writing, it's possible to read the contents of that file by using the Fread( ) function:

```
long status, count;
int handle;
char *buffer;
  status = Fread(handle, count, buffer);
```

This function reads *count* number of bytes from the file whose file handle is stored in *handle*, and places that data into the buffer whose address is stored in *buffer*. If the function is successful, the actual number of bytes read is returned in *status*. If the function attempts to read past the end of the file, a 0 is returned. For any other error, a negative GEMDOS error number is returned (see Appendix D).

Unless otherwise specified, file reads and writes start at the beginning of the file and progress sequentially towards the end of the file. Let's say, for example, that you open a file that's 1000 bytes long for reading. The internal file pointer of GEMDOS, which keeps track of where you are in a file, is initially set at the beginning of the file, so the first time you call Fread( ) to read 100 bytes, you'll get the first 100 bytes in the file. After you read that 100 bytes, the file pointer is moved to the end of the block that you read. The next time you read 100 bytes, therefore, you'll get the second 100, and so on until you reach the end of the file. Since disks are basically random-access devices, however, you're not confined to reading files in order. You can move the file pointer wherever you want in the file by using the Fseek( ) function. Fseek( ) is called like this:

```
long position, offset;
int handle, seekmode;
  position = Fseek(offset, handle, seekmode);
```

Fseek( ) moves the file pointer for the file referred to by *handle* by *offset* number of bytes, in a manner designated by *seekmode*. The movement of the file pointer is made relative to one of three points:

- The beginning of the file
- The end of the file
- The current file pointer position

The *seekmode* specifies where the move begins:

Seekmode	File Pointer Movement
0	Relative to beginning of file
1	Relative to current position
2	Relative to end of file

The *offset* is a signed number that moves the pointer a positive or negative number of bytes from its starting point. If *offset* is positive, the pointer moves toward the end of the file. If negative, it moves towards the beginning. For example, if *seekmode* is 0 and *offset* is 100, the file pointer is set to byte 100 of the file, and the next read starts with byte 101. If *seekmode* is 1 and *offset* is −100, the file pointer moves back 100 bytes from its current position. After the Fseek( ) call, the absolute current position of the file pointer (expressed as an offset from the beginning of the file) is returned in *position*. Therefore, when you call Fseek( ) with a *seekmode* of 2 and an *offset* of 0, the length of the file is returned in *position*.

Program 6-1 demonstrates many of the file I/O functions explained above. It writes out a short test file to the floppy disk, reads it back, changes it, and reads it again.

Program 6-1. GFILEIO.C

```
/************************************************/
/*                                              */
/*  GFILEIO.C                                   */
/*                                              */
/*  Demonstrates the GEMDOS file-creation,      */
/*  write, read, and close functions            */
/*                                              */
/************************************************/
```

```
#include <osbind.h>     /* For GEMDOS macro definitions */
#define F_ATTR  0       /* file attribute for FCreate()   */
#define APPEND  3
#define READ    0
#define WRITE   2


char fname[]= "A:\TEST.FIL";
char test[]= "This is a test file";
char add[]= "nice";

main()
{
   int handle;
   long status;

   handle = Fcreate(fname, F_ATTR); /* create the file */
   if(handle<0)                   /* if you can't, quit */
      Cconws("Could not create file\n\r");
   else
      {                          /* otherwise, write test string */
      status = Fwrite(handle, 19L, test);
      printf("\n%ld characters written\n\n",status);
      Fclose(handle);

      handle = Fopen(fname, READ); /* read it to be sure it's there */
      status = Fread(handle, 19L, test);
      test[status]=0;
      Cconws(test);              /* and print it out */
      printf("\n\r%ld characters read\n\n",status);
      Fclose(handle);

      handle = Fopen(fname, APPEND);  /* now change it */
      status = Fseek(10L, handle, 0);
      status = Fwrite(handle,4L,add);

      status = Fseek(0L, handle, 0); /* and read the change */
      status = Fread(handle, 19L, test);
      test[status]=0;
      Cconws(test);              /* print it out */
      printf("\n\r%ld characters read\n",status);
      Fclose(handle);
      }
/* if you're running this from the Desktop, you may want to
   add a pause to give the user a chance to read the output */

   Cconws("\n\n\rPress any key to end");
   Cconin();
}

/******** end of GFILEIO.C *****/
```

Notice how the count values for reads, writes, and seeks were specified as 32-bit longwords by placing the character *L* (as in 19L) next to them? It's very important to pass arguments of the correct size in these functions. If you pass an integer as the count byte, it will be assumed to be the first word of the longword, and you'll end up with a file length in the millions of bytes. What was intended to be only a short file can soon fill up a floppy disk if you aren't careful.

Disk and Directory Path Functions

Besides simple reading and writing of files, GEMDOS supports a number of disk and directory path functions. These functions facilitate navigation through a system where there may be several drives attached, each having several subdirectories. A common example is the function used to set the default drive. This is the drive that GEMDOS assumes is referred to when only a filename is used. For example, if you ask to open a file called MYFILE, rather than using an entire pathname like C:\FILES\MYFILE, GEMDOS will look for MYFILE in the default directory of the default drive. The macro name for the function used to set the default directory is Dsetdrv( ). This function also returns information about the number of logical drives recognized by the system. To call this function, use the following format:

```
long drives;
int default;
  drives = Dsetdrv(default);
```

where *default* is the drive number of the drive you wish to make the current default (drive A: = 0, drive B: = 1, and so on). A list of known logical drives (those on which a directory has been used) is returned as a bit flag in *drives*. Each bit that corresponds to a known drive is set to 1. For example, the number 7 has bits 0, 1, and 2 set, which indicates that drives A:, B:, and C: are connected. Note that logical drives do not have to be separate physical devices. For example, a single floppy system will still have two logical drives, since the floppy can be accessed as either drive A: or drive B:. It's also possible to partition a single hard drive into several logical units called C:, D:, E:, and so forth. Finally, portions of memory may be partitioned into logical RAM drives as well. On the current version of GEMDOS, up to 16 drives may be connected, though future versions may support up to 32.

GEMDOS also allows you to find the drive number of the default drive, with the function Dgetdrv:

```
int default;
  default = Dgetdrv( );
```

where *default* is the drive number (0–15) of the current default drive. This function is useful for building a default path string to remember where a program should look for data files, for example.

A related function allows you to set the current default directory on a drive. As stated above, this is the directory where GEMDOS will first search for a named file. GEMDOS keeps a default directory path for each drive in the system. The Dsetpath( ) function is the one used to set a default directory for the current drive:

```
int status;
char *path;
  status = Dsetpath(path)
```

where *path* is a pointer to a null-terminated ASCII string specifying the default directory path to set for the current default drive (foe example, WORDPRO\FRED\LETTERS). If the path name begins with a drive letter and a colon, the path is set for that drive rather than the current default drive (for instance, C:\DATABASE\CLIENTS).

Once a default directory path has been set, you can use the Dgetpath( ) function to find that path setting:

```
word drivenum;
char *buffer;
  Dgetpath(buffer, drivenum);
```

where *drivenum* is the drive number (0–15) of the disk whose default directory path you wish to find. An ASCII string containing the path name is returned in *buffer*. Since GEMDOS does not specify a maximum length for path names, make sure to supply a buffer large enough to contain the whole name. A buffer of about 128 bytes should be big enough, unless the path goes down through more than ten levels of subdirectories.

The last two directory path functions are those used to create or delete a subdirectory. The Dcreate( ) function is used to create a new subdirectory. The syntax used to call this function is:

```
int status;
char *pathname;
  status = Dcreate(pathname);
```

where *pathname* is a pointer to a null-terminated ASCII directory path (for example, C:\NEWDIR). If GEMDOS is able to create the directory, a 0 is returned in *status*, otherwise, a negative GEMDOS error number is returned. This function will fail if some part of the path name doesn't exist, the named subdirectory already exists, or the parent directory is the root directory of a disk, all of whose directory entries have been used.

The Ddelete( ) function is used to remove a subdirectory:

int status;
char *pathname;
status = Ddelete(pathname);

where *pathname* points to a null-terminated ASCII string which contains the path name of the directory to delete. If GEMDOS is able to delete the directory, a 0 is returned in *status,* otherwise, a negative GEMDOS error number is returned. Note that a subdirectory may only be deleted if it is empty.

Program 6-2 below demonstrates the entire process for obtaining a directory listing in C.

Program 6-2. GDIR.C

```
/***********************************************/
/*                                             */
/*  GDIR.C--Demonstrates how to get a          */
/*  directory listing using GEMDOS             */
/*  file functions.                            */
/*                                             */
/***********************************************/

#include <osbind.h>    /* For GEMDOS macro definitions */

struct diskbufr        /* data structure for DTA */
   {
   char resvd[21];
   char attr;
   int  ftime;
   int  fdate;
   long fsize;
   char fname[14];
   } dta;

char blank[13]="            "; /* 12 spaces--for name padding */

main(argc,argv)
   int argc;        /* number of command arguments */
   char *argv[];    /* array of pointers to command strings */
{
   struct diskbufr *olddta;
   char *temp;

   if (argc>2)      /* if more than one command argument */
      {
      Cconws("Too many arguments\r\n");
      exit(0);      /* complain and quit */
      }

   if (argc==2)
       temp=argv[1];   /* if only one argument, use it */
   else temp = "*.*";  /* or default to "*.*"   */

   olddta = (struct diskbufr *)Fgetdta(); /* save old DTA */
   Fsetdta (&dta);                        /* and use new DTA */

   if (Fsfirst(temp,0x10))           /* get first file */
      puts ("File(s) not found");

   else
      {
      print_entry();                 /* and print entries.. */

   while(!Fsnext())                  /* until done */
      print_entry();
   }
   Fsetdta (olddta);                    /* then restore DTA */

/* if you're running this from the Desktop, you may want to
   add a pause to give the user a chance to read the output */

   Cconws("\n\n\rPress any key to end");
   Cconin();

}  /* end of main */


print_entry()   /* format output for directory listings */
{
   char temp[20];

   Cconws(dta.fname);                 /* print file name */
   Cconws(blank+strlen(dta.fname)); /* pad to 12 spaces */

if(dta.attr==0x10)        /* if this is a directory */
   {
   Cconws("<DIR>\r\n"); /* don't print size or date */
   }

else
   {
   sprintf(temp,"%7ld",dta.fsize); /* else print file size */
   Cconws(temp);
   Cconws(" bytes  ");

   sprintf(temp,"%02d/%02d/%d",  /* print file date */
           dta.fdate&0x1F,(dta.fdate &0x1E0)>>5,
           (dta.fdate>>9)+1980);
   Cconws(temp);
   Cconws("  ");

   sprintf(temp,"%02d:%02d:%02d",  /* print file time */
           (dta.ftime>>11)&0x1F,(dta.ftime &0x7E0)>>5,
           (dta.ftime&0x1F)*2);
   Cconws(temp);
   Cconws("\r\n");
   }

} /* end of print_entry() */

/****************** end of GDIR.C ***********/
```

After compiling this program, you should rename it GDIR.TTP to indicate that it may take parameters. To obtain a specific directory listing, enter the directory name as the parameter, including wildcard characters, like C:\PROGRAMS\*.PRG. You may also run this program from within the GPEXEC.TOS program described in Chapter 5. Once GPEXEC prompts you for a command, enter it in the format:

GDIR.TTP A:*.*

Directory Listings and Free Space Functions

One of the most common disk operations is obtaining a listing of the directory contents. Since a disk may have hundreds of files and subdirectories, it might be difficult setting up one big buffer to get all of the names at once. Therefore, GEMDOS uses a system by which you set up a buffer large enough to hold only the information for a single file, and then proceed to read the directory information a file at a time.

The first step is to set the Disk Transfer Address (DTA) that points to the buffer used for disk read operations. This buffer is used as a scratch area for directory searches. To set the buffer, use the Fsetdta( ) function:

```
char dta[44];
  Fsetdta(dta);
```

where *dta* is a pointer to the Disk Transfer Address buffer that will be used from now on. You may wish to save the default DTA, and restore it when you're finished. To find the current DTA before setting your own, use the Fgetdta( ) function

```
char *dta;
  dta = Fgetdta( );
```

which returns a pointer to the current Disk Transfer Address buffer in *dta*.

The next step in obtaining a directory listing is to perform a search for the first file on the list. When asking for a directory listing, you may specify the file attributes of files to be searched for, as well as names and/or extensions. The Fsfirst( ) function used for this purpose is called in the following manner:

```
int status, attributes;
char *filespec;
  status = Fsfirst(filespec, attribs);
```

where *filespec* points to a null-terminated ASCII string containing the file specification to be searched for. Since wildcards may be used in a file specification, it may refer to a single file (for example, C:\MYPROG.PRG), a class of files (for example, *.BAS), or any file at all (for example, *.*). The *attributes* argument specifies the types of files for which to search. As you've seen above, file types include read-only, system, hidden, volume name, and directory. When a particular bit of the *attributes* flag is set, the directory search only includes files with that attribute. The various attribute bits are shown in Table 6-5.

Table 6-5. Attribute Bits

Bit Number	Attribute
0	Read-only file (can't be deleted or written to)
1	Hidden file (excluded from normal directory searches)
2	System file (excluded from normal directory searches)
3	Volume label (can only exist in root)
4	Subdirectory
5	Archive bit

If the *attributes* argument is 0, the Fsfirst( ) function only searches for normal files (no subdirectories, hidden files, or volume labels). If *attributes* has the volume-label bit set, only volume labels are searched for.

When Fsfirst( ) matches the file specification and attribute type to an existing file, it returns a 0 byte in *status*. It also writes a 44-byte data structure to the buffer pointed to by the DTA. The contents of that data structure are shown below:

Byte Number	Contents
0–20	Reserved for internal use (must not be altered)
21	File attributes
22–23	Time stamp
24–25	Date stamp
26–29	File size
30–43	Filename and extension

C programmers may find it helpful to declare the DTA buffer as a data structure, in the following format:

```
struct diskbuf
    {
    char resvd[21];         /* reserved for internal user */
    char attr;              /* file attribute byte */
    int ftime;              /* file time stamp */
    int fdate;              /* file date stamp */
    int fsize;              /* file size in bytes */
    char fname[14];         /* file name and extension*/
    };
```

If no match for the file is found, EFILNF or some other appropriate negative GEMDOS error number is returned. See Appendix D for a complete list of GEMDOS errors.

Once the first file matching the filename and attributes set in Fsfirst( ) is found, you can obtain information about additional files with the function Fsnext( ):

```
int status;
  status = Fsnext( );
```

where *status* indicates whether another file having the file specification and attributes given in the Fsfirst( ) call was found. If such a file was found, a 0 is returned in *status*, and information about the file is stored in the buffer pointed to by the DTA. If no matching file was found, a DOS error number such as ENMFIL is returned (see Appendix D). Use of this call assumes that the DTA points to a buffer that contains information from a previous Fsfirst( ) call. The filename used by the previous Fsfirst( ) call must also have contained one or more wildcards (? or *) in order for Fsnext( ) to succeed.

The amount for free space is another useful bit of information you might wish to obtain about a disk. Find this with the function whose macro name is Dfree( ). The function is called like this:

```
long buffer[4];
int drivenum;
  Dfree(buffer, drivenum);
```

where *drivenum* specifies the drive to check (A = 0, B = 1, and so on). The information about the drive is returned in four longwords in the *buffer* array. The information contained in this array is:

Element Number	Contents
0	Number of free clusters
1	Total number of clusters on drive
2	Sector size (in bytes)
3	Cluster size (in sectors)

For large hard drives, this function is extremely slow, and may take as much as several seconds to complete.

File Manipulation Functions

GEMDOS supports some common file manipulation operations like deleting a file or renaming it, and some not-so-common ones, like setting or getting the time and datestamp, or file attributes. The function Fdelete( ) is used to delete a file. It's called like this:

```
int status;
char *filename;
  status = Fdelete(filename);
```

where *filename* is a pointer to a null-terminated ASCII string that contains the name of the file to be deleted. If the file is successfully deleted, a 0 is returned in *status*. If the operation fails, a negative error number is returned instead.

Renaming a file is the job of a function called Frename( ):

```
int status;
char *oldname;
char *newname;
  status = Frename (0, oldname, newname);
```

where *oldname* is a pointer to a string containing the name of the file to be changed, and *newname* is a pointer to the string containing the new name. Wildcard characters may not be present in either the source or destination filenames. The destination filename may, however, be in another directory, which effectively "moves" the file from one directory to the other. Of course, you can't rename a file to another directory if that directory already contains a file of the same name. Note that a dummy 16-bit 0 argument must be used at the beginning of the call as a place holder. If the rename operation is successful, a 0 is returned in *status*. If the operation fails, a negative error code is returned.

It is possible to read or change a file's attributes with the Fattrib( ) function. The function is called like this:

```
int attributes, mode, newattr;
char *filename;
  attributes = Fattrib(filename, mode, newattr);
```

Fattrib( ) sets or gets the attributes of the file whose name is contained in the string pointed to by *filename,* according to the setting of *mode.* If *mode* is set to one, the file attributes are changed to those passed in *newattr,* and no value is returned in *attributes.* If *mode* is set to 0, the current file attributes are returned in *attributes.* Both *newattr* and *attributes* are bit flags, in which each bit corresponds to a particular attribute, as shown in Table 6-5.

You may get or set a file's time and date stamp with the function Fdatime( ), which is called as follows:

```
int handle, mode;
long *timeptr;
  Fdatime(timeptr, handle, mode);
```

Fdatime( ) gets or sets the timestamp for the file referred to by *handle,* according to the setting of *mode.* If *mode* is set to 1, the file's time stamp is set according to the value stored in the buffer pointed to by *timeptr.* If *mode* is set to 0, the file's time stamp is read into that buffer. In either case, *timeptr* contains the address of a 32-bit buffer that holds the time and date stamp information. The first word of the buffer contains the date in the following format:

Bit Number	**Description**	**Range**
0–4	Day	1–31
5–8	Month	1–12
9–15	Year	0–119*

* Year value is added to 1980 to get current year.

The second word holds the time, as follows:

Bit Number	**Description**	**Range**
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23

File I/O Redirection

The standard file handles (standard input, standard output) are commonly used for console-driven TOS applications. Sometimes, however, you may want to redirect the output of a TOS program to a device other than the screen. For example, if you're running a program to list a disk directory on the screen, you may want to send that output to a printer. This type of redirection is a common feature of command line interface shell programs.

GEMDOS includes a couple of functions that make it possible to do this kind of I/O redirection. This first is used to create a user-designated file handle that duplicates one of the standard file handles. In other words, it takes a file handle with a value of five or less, and returns a file handle with a value of six or more, which points to the exact same device. This is useful for two reasons.

The first reason is that if you change a standard device handle, you want to have some way of changing it back. Therefore, use the handle duplicate function to create a temporary copy of the original handle before you redirect the I/O for that device. Then, after you finish the redirection, you can change the handle back, using the temporary copy.

The second reason for duplicating a handle is that it gives you a temporary copy that can be used to substitute for another standard device. If you're going to replace the screen output with output to the printer, for instance, you'll need two handles for the printer, one for the standard list device, and one for the standard output device.

To create a user-designated handle that duplicates one of the standard file handles, use the Fdup( ) function:

```
int newhandle, handle;
  newhandle = Fdup(handle);
```

where *handle* is one of the standard device handles (0–5). If Fdup( ) is successful, it returns a user-designated file handle (greater than five) that refers to the same file.

To redirect I/O from a standard device handle to a user-designated one, use the function Fforce( ):

```
word status, standard, user;
  status = Fforce(standard, user);
```

where *standard* is the file handle of the standard file, and *user* is the handle of the user-designated file that replaces that standard file. If the operation is successful, a 0 is returned in *status*. Otherwise, a GEMDOS error code such as EIHNDL is returned (see Appendix D).

Chapter 7

Line A Routines

All three screen display modes on the ST are bit-mapped, rather than character-oriented. This means that everything the computer displays, including text, is composed of a series of dots or *pixels* (picture elements). To help the programmer cope with the ST's heavy emphasis on graphics, the Operating System offers the programmer a wide range of graphics functions. High level support for graphics operations on the ST is provided by the GEM VDI, as described in *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.* TOS, however, includes a set of graphics primitives known as the line A routines. These are the low-level graphics routines called by the VDI.

The line A routines and the GEM VDI have advantages and disadvantages. Using the VDI results in a more portable code than using the line A routines, since the VDI routines may be accessed from C language programs that can be compiled and executed on either the ST or MS-DOS computers running under GEM. The VDI is also a little easier to use because C compilers for the ST include GEM bindings that allow programmers to use VDI functions as if they were standard C functions. This ease of use has a performance cost, however, since the computer wastes time translating the commands from a format that is convenient for the user into a format more convenient for the Operating System.

While the line A routines require a little more effort to use than the VDI, they offer slightly faster performance in return. They also require less overhead, since it isn't necessary to perform GEM initialization routines or open a VDI workstation in order to use them. They also offer a few options not found in the VDI routines, such as a choice of 16 logic operations for text printing. Though the line A routines are specific to the ST, they provide full compatibility within the ST family. This means that the same routines will work on future models of the ST, even if the graphics hardware changes significantly. A good example of this is that software using line A routines automatically benefits from the blitter hardware in the Mega ST line, since the blitter ROMs use that hardware to implement the line A routines.

The line A routines operate by taking advantage of a feature of the 68000 microprocessor's exception handling. No valid computer instruction *(opcode)* on the 68000 starts with the binary digits 1010 (or the ASCII character *A* in hexadecimal notation). Therefore, when the processor encounters an instruction that starts with that number, it triggers the Opcode 1010 Emulation exception. The processor is thrown into Supervisor mode and program execution is routed through exception vector 10, meaning that the processor starts executing the program whose address is stored at location 40 ($28). On the ST, this exception vector points to the line A handler, which routes execution to the proper graphics primitive. After the line A function is performed, program execution resumes at the instruction immediately following the line A opcode.

A 1K section of RAM is set aside on the ST for the storage of graphics-related variables. Whenever you need to pass information to one of the line A routines, such as where to draw a line or what color to make the line, store that information directly in the line A variable table (see Appendix H for a complete description of line A variables). Information stored in the table generally remains intact from call to call, which means that you don't have to store it in the table again when you know it's already there. Note, however, that GEM calls use the same variable table, so if you call GEM routines between calls to line A, you may find that some of the variables have changed in value.

When all of the information you need for a line A operation is stored in the variable table, call the desired line A routine by placing its opcode in your machine language program. Since there are no official opcode mnemonics for these instructions, code them in the form

```
dc.w   $A00x
```

where *x* is the opcode number from $0 to $F. For example, to call function 0, the line A initialization function, you would use the instruction

```
dc.w   $A000
```

It's technically possible to call the line A routines from C, given the proper library`routines. At this time, however, only the *Mark Williams* C compiler and *Megamax Laser* C provide such library routines as a standard feature. There are a number of reasons you might not want to use the line A routines from C. First, the GEM VDI routines are easier to call from C and provide adequate performance in most circumstances. Second, there is no standard way to call these routines from C, making your code less portable. Finally, in cases where you wish to use the line A routines to improve performance, you'll probably want to avoid the overhead added by the C compiler and write directly in machine language instead. For these reasons, we'll confine our discussion of the line A routines to machine language programming.

Use of the line A routines requires a fair understanding of the system used by the ST to produce graphics. An in-depth examination of this subject could fill a book of its own, and is therefore beyond the scope of this work. Since the line A routines were written to support the GEM VDI functions, however, much of the material that deals with VDI graphics is applicable to the line A routines as well. The reader who wishes to learn more about the line A functions should look to the *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI,* which covers the VDI graphics functions in detail.

Line A Initialization Command

Most line A commands require some input values to be placed in the line A variable table. The location of this table will vary depending on which version of TOS you're using, so find the base address of the Line A variable table before using line A calls. To do this, use the $A000 opcode, which is referred to as the line A initialization function. When you include the instruction

```
dc.w.   $A000
```

in your program, it calls the line A handler, which returns three useful pointers in some of the data and address registers. The registers used, and the values returned in them, are listed below:

Register	Contents
D0	Pointer to base address of line A variable table
A0	Pointer to base address of line A variable table
A1	Pointer to a null-terminated table of pointers to the system font headers
A2	Pointer to a null-terminated table of pointers to the line A routines

The first value, a pointer to the base address of the line A variable table, is returned in both the D0 and A0 registers. All addresses in the line A variable table are calculated as offsets from this value. For example, the INTIN variable is located at base + 8. Therefore, after the $A000 call, you could access INTIN by using the expression 8(A0). Since some of the line A routines alter the contents of register A0, you may wish to save the base address in memory, or in a "safe" register like A3–A5. That way, once you've made the $A000 call and established the base address of the variable table, you needn't call it again later in your program.

The second value the Init call returns is a pointer to the system font header table in register A1. This information is useful for setting the font used by the TextBlt function ($A008) to print text. The font header table is a list of addresses of font headers for the various system fonts. Each item in the list is a four-byte longword address and there's an entry of 0L to mark the end of the list. Currently, this table has three entries that point to the headers for the three ST system fonts: The 6 × 6 font; the 8 × 8 font; and the 8 × 16 font. Each header is 87 bytes long and contains such information as the name of the font, the font ID, the font size, first character, last character, cell width and height, and so on. Complete information about the font headers may be found in Appendix C of *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

The last item returned by the Init call is a function table which is placed in register A2. This table contains 16 entries, each of which is the longword address of one of the line A routines. By indexing into this table, you may call the line A subroutines directly, using the JSR instruction: For instance, use JSR 4(A2) to call function $A001. This saves time normally spent handling the line A exception. To call these subroutines directly, however, you must first be in supervisor mode.

Drawing Routines

The simplest of the line A functions are those used to draw single points or straight lines on the screen. Function $A001, Put Pixel, is used to color a single point. It requires that three input values be set in the variable table. The first is the horizontal *(x)* position of the dot, the second is the vertical *(y)* position, and the third is the color register to use for the drawing pen. The *x* and *y* positions are stored in an array of 16-bit words (in that order), and a pointer to the array is placed in the variable PTSIN. Horizontal coordinate values range from 0 to 639 in high- and medium-resolution modes, and from 0 to 319 in low-resolution mode. Vertical coordinates range of 0 to 199 in low- and medium-resolution modes, and from 0 to 399 in high resolution. The color register is stored as a 16-bit word, and a pointer to that word is placed in the INTIN variable. Color register numbers range from 0 to 15 in low resolution, 0 to 3 in medium resolution, and 0 to 1 in high resolution mode.

Program 7-1 uses the Put Pixel opcode to cover a rectangular area with black dots.

Program 7-1. PUTPIX.S

```
*************************************************************
*                                                           *
*     PUTPIX.S -- Demonstrates the use of the line A        *
*     Put Pixel ($A001) routine.  Covers a rectangular      *
*     area with black dots.                                 *
*                                                           *
*                                                           *
*************************************************************

*** Variable table offsets

INTIN  =  $08
PTSIN  =  $0C

*** Function Equates
Init     = $a000
PutPixel = $a001

*** Program starts here

.text
   dc.w     Init                 * get base address of variable table
   move.l   #color,INTIN(a0)     * pass address of color reg. to use
   move.l   #point,PTSIN(a0)     * pass address of coordinate array

   move.w   #49,d3               * draw 50 dots across
   move.w   d3,point             * starting at x of 51
```

```
nextx:
   addq.w    #2,point          * skip even columns

   move.w    #49,d4            * draw 50 dots vertically
   move.w    d4,point+2        * starting at y of 51
nexty:
   addq.w    #2,point+2        * skip even rows
   dc.w      PutPixel          * draw a point

   dbra      d4,nexty          * if the column's not done, do next
   dbra      d3,nextx          * if the row's not done, do next

*** wait for key press, then end

   move.w    #1,-(sp)         * call conin() to wait for key press
   trap      #1
   addq.l    #2,sp

   move.l    #0,-(sp)          * GEMDOS terminate command
   trap      #1                * call GEMDOS and exit

*** Data for input is stored here
.data
color:       dc.w     1        * use color register 1 (black)
.bss
point:       ds.w     2        * storage for x,y coordinates

.end
```

The inverse operation of Put Pixel is Get Pixel ($A002). Given a set of *x,y* coordinates, Get Pixel finds the color register used to display the dot located at that point. As with Put Pixel, the *x* and *y* coordinates are placed in an array of 16-bit words and a pointer to that array is placed in the PTSIN variable. Upon return from the Get Pixel call, the color register used for the specified pixel is returned in register D0.

The next step in complexity from drawing points is drawing straight lines. The main routine for drawing lines is called Arbitrary Line ($A003), because it can be used to draw a straight line between any two points. This function requires a lot more input than Put Pixel. The variables used (and their offsets from the base address of the table) are shown below:

```
COLBIT0     = $18   * bit value (0 or 1) for color plane 0
COLBIT1     = $1A   * bit value (0 or 1) for color plane 1
COLBIT2     = $1C   * bit value (0 or 1) for color plane 2
COLBIT3     = $1E   * bit value (0 or 1) for color plane 3
LSTLIN      = $20   * draw last pixel of the line?
                    * (0 = yes, 1 = no)
LNMASK      = $22   * line pattern mask
WMODE       = $24   * writing mode (0 = replace,
                    * 1 = transparent, 2 = XOR, 3 = reverse
                    * transparent)
X1          = $26   * starting x coordinate
```

```
Y1          = $28   * starting y coordinate
X2          = $2A   * ending x coordinate
Y2          = $2C   * ending y coordinate
```

The last four variables hold the coordinates for the starting point and ending point on the line. The first four are used to store the color register value. Instead of storing this value as one word, for the purposes of this function, store it as bit values in each of four words. For example, to select a register value of 8, you would store a 1 in COLBIT3, and a 0 in each of the other three variables.

The other variables introduce some new features. LNMASK is used to create a line pattern for dotted or dashed lines. This is a 16-bit value which is logically ANDed with the line. All the points on the line that correspond to 1 bits in the mask are drawn in the selected pen color, while all of the points that correspond to 0 bits in the mask remain in the background color. Therefore, a mask of $FFFF produces a solid line, while $5555 produces a dotted line. Note that the least significant bit of LNMASK is aligned with the right-most endpoint of the line.

The WMODE variable is used to determine the drawing mode is used. The four modes are replace, transparent, XOR, and reverse transparent. In mode 0, replace mode, all 1 bits in the mask are drawn in the selected foreground color, while all 0 bits are drawn in the background color. In transparent mode (mode 1), 1 bits are drawn in the foreground color while the colors in areas corresponding to 0 bits are left alone. In mode 3, reverse transparent, the reverse happens. Areas corresponding to 1 bits in the mask are left alone, while 0 bits are drawn in the foreground color.

The last mode, number 2, is called XOR mode. In this mode, neither the foreground nor background colors are used to draw the line. Instead, wherever there is a 1 bit in the line pattern, the color that already exists on the screen at that spot is logically complemented, using the XOR operation. That means that the values in the color bit planes are all reversed. A color value of 5 (0101), for example, becomes a color value of 10 (1010).

The XOR mode has some unique properties. If you color a green background with a green line, you won't see the line. But if you use an XOR line, it will always show up, since it changes whatever color is there. Secondly, the XOR operation cancels itself the second time it's used. When a color is complemented, it changes to its opposite color, but when it's complemented again, it changes back to the original.

To prevent this property from erasing a common end point of two lines that are both drawn in XOR mode, there is a variable called LSTLIN. This is a flag that lets you choose whether the last pixel of the line will be drawn. If LSTLIN is zero, the last pixel of the line will be drawn, but if it is nonzero, it will not be drawn. If you are drawing a series of connected lines in XOR mode, where the last point of one line segment is used for the first point of the next line segment, you'll want to place a 1 in LSTLIN.

Program 7-2 shows how to draw a series of dashed, connected lines using function $A003. Try changing the line pattern and the color planes (if you have a color monitor) to see what the effect is.

Program 7-2. LINE.S

```
************************************************************
*                                                          *
*     LINE.S -- Demonstrates the use of the line A         *
*     Arbitrary line ($A005) function.  Draws 6            *
*     dashed, connected lines.                             *
*                                                          *
*                                                          *
************************************************************


*** Variable table offsets

COLBIT0 = $18
COLBIT1 = $1A
COLBIT2 = $1C
COLBIT3 = $1E
LSTLIN  = $20
LNMASK  = $22
WMODE   = $24
X1      = $26
Y1      = $28
X2      = $2A
Y2      = $2C


*** Function Equates
Init     = $a000
Aline    = $a003

*** Program starts here

.text
   dc.w     Init                 * get base address of variable table

   move.w   #1,COLBIT0(a0)       * 1 is lsb
   move.w   #0,COLBIT1(a0)       * 0 in all other bits
   move.w   #0,COLBIT2(a0)       * = 0001
   move.w   #0,COLBIT3(a0)     * or color 1
   move.w   #0,LSTLIN(a0)      * draw last point on line
   move.w   #$EEEE,LNMASK(a0)  * dashed line
   move.w   #0,WMODE(a0)       * replace mode

   move.w   #5,d3              * draw 6 lines
   move.l   #points,a3         * get address of array
nextline:
   move.l   (a3)+,X1(a0)      * starting x & y
   move.l   (a3),X2(a0)       * ending x & y
   dc.w     Aline             * draw the line
   dbra     d3,nextline       * if not at end, draw next line

*** wait for key press, then end

   move.w   #1,-(sp)          * call conin() to wait for key press
   trap     #1
   addq.l   #2,sp

   move.l   #0,-(sp)          * GEMDOS terminate command
   trap     #1                * call GEMDOS and exit

*** Data for input is stored here
.data

points:       dc.w     150,49,250,99,250,149,150,199
              dc.w     50,149,50,99,150,49

.end
```

There is another line A routine ($A004) used only for drawing horizontal lines. If you know that your line will be horizontal, you may wish to choose that function, as it's slightly faster for horizontal lines. Since that function uses some of the filled shape drawing variables, however, it will be discussed in the following section on drawing filled shapes.

Drawing Filled Shapes

Line A provides a number of functions for drawing filled shapes. These functions introduce some new concepts. The first is the *fill pattern*. A fill pattern is an array of line patterns that is repeated in two dimensions. A pattern is 16-bits wide and as high as you wish to make it, though the pattern should be a power of two in length (one line, two lines, four lines, sixteen lines, and so on). It's also possible to create multicolor fill patterns. In these patterns, there is a separate array for each color bit plane.

Another concept used by the filled shape routines is the *clipping rectangle*. When you set up a clipping rectangle, specify that drawing operations will only be carried out within a designated portion of the screen. If you set a clipping rectangle that extends from coordinates 0,0 to 150,150, and turn clipping on, only the part of the filled shape that falls within this area will actually be drawn.

The first of the routines that use a fill pattern is the Horizontal Line function ($A004). This function is similar to Arbitrary Line, except it is only used when Y1 and Y2 are the same. It is a bit faster in execution than Arbitrary Line. The input variables required for this function are shown in Table 7-1.

Table 7-1. Input Variables for Horizontal Line Function

Offset Name	Offset	Description
COLBIT0	$18	Bit value for color plane 0
COLBIT1	$1A	Bit value for color plane 1
COLBIT2	$1C	Bit value for color plane 2
COLBIT3	$1E	Bit value for color plane 3
WMODE	$24	Writing mode
X1	$26	Starting *x* coordinate
Y1	$28	Starting (and ending) *y* coordinate
X2	$2A	Ending *x* coordinate
PATPTR	$2E	Pointer to fill pattern array
PATMSK	$32	Pattern index (length − 1)
MFILL	$34	Multicolor fill pattern flag (zero = single plane, nonzero = multi-plane)

The first seven of these variables should be familiar from the Arbitrary Line function, above. The last three, however, pertain to the fill pattern. PATPTR contains a pointer to the array of line masks that form the fill pattern mask. PATMSK is used as an index into the pattern array and should contain the length of the array (in words), minus one. MFILL is a flag that indicates whether the pattern contains a single color plane, or multiple bit planes. A zero indicates a single bit plane, in which case WMODE as well as the COLBIT variables determine the pattern color. A nonzero value indicates a multiplane fill, which replaces the destination bitplanes without regard for WMODE.

The next filled shape function is Filled Rectangle ($A005). This function draws a series of filled horizontal lines of equal length. The input values it requires are shown in Table 7-2.

Table 7-2. Input Values for Filled Rectangle Function

Offset Name	Offset	Description
COLBIT0	$18	Bit value for color plane 0
COLBIT1	$1A	Bit value for color plane 1
COLBIT2	$1C	Bit value for color plane 2
COLBIT3	$1E	Bit value for color plane 3

Table 7-2. Input Values for Filled Rectangle Function (continued)

Offset Name	Offset	Description
WMODE	$24	Writing mode
X1	$26	Starting *x* coordinate
Y1	$28	Starting *y* coordinate
X2	$2A	Ending *x* coordinate
Y2	$2C	Ending *y* coordinate
PATPTR	$2E	Pointer to fill pattern array
PATMSK	$32	Pattern index (length − 1)
MFILL	$34	Multi-color fill pattern flag (zero = single plane, nonzero = multiplane)
CLIP	$36	Clipping flag (zero = off, nonzero = on)
XMINCL	$38	Coordinate of left side of clip rectangle
YMINCL	$3A	Coordinate of top of clip rectangle
XMAXCL	$3C	Coordinate of right side of clip rectangle
YMAXCL	$3E	Coordinate of bottom of clip rectangle

These inputs are the same for Horizontal Line, with the addition of the clipping rectangle variables. XMINCL, YMINCL, XMAXCL, and YMAXCL are used to designate the borders of the clipping rectangle. The CLIP variable is used to indicate whether the clipping actually takes place. If CLIP is set to 0, clipping is turned off, and the rectangle designated by the other four variables is ignored. If CLIP is not 0, clipping is turned on, and only the portion of the filled rectangle that lies within the clipping rectangle will be drawn.

A similar call to Filled Rectangle is Filled Polygon ($A006). Instead of drawing a filled box, this call draws a filled shape with an arbitrary number of sides. It does this by drawing a series of filled horizontal lines of unequal size. The input values used by Filled Polygon are shown in Table 7-3.

Table 7-3. Input Values Used by Filled Polygon

Offset Name	Offset	Description
CONTRL	$04	Pointer to a word array. The second member of the array, CONTRL[1], contains the number of polygon vertices
PTSIN	$0C	Pointer to a word array. This array contains polygon vertex pairs in the format (x1, y1), (x2, y2), and so on.
COLBIT0	$18	Bit value for color plane 0
COLBIT1	$1A	Bit value for color plane 1
COLBIT2	$1C	Bit value for color plane 2

Table 7-3. Input Values Used by Filled Polygon (continued)

Offset Name	Offset	Description
COLBIT3	$1E	Bit value for color plane 3
WMODE	$24	Writing mode
Y1	$28	*y* coordinate of horizontal line segment(s) to draw
PATPTR	$2E	Pointer to fill pattern array
PATMSK	$32	Pattern index (length-1)
MFILL	$34	Multicolor fill pattern flag (zero = single plane, nonzero = multiplane)
CLIP	$36	Clipping flag (0 = off, nonzero = on)
XMINCL	$38	Coordinate of left side of clip rectangle
YMINCL	$3A	Coordinate of top of clip rectangle
XMAXCL	$3C	Coordinate of right side of clip rectangle
YMAXCL	$3E	Coordinate of bottom of clip rectangle

This function draws a filled shape that can have any number of vertices. The *x* and *y* coordinate pairs are placed into an array and a pointer to that array is placed in the PTSIN variable. Since these points must describe a closed figure, the last point must be the same as the first one. The total number of points in the figure must be placed in the second word of an array and a pointer to this array must be placed in the CONTRL variable.

The Filled Polygon function only draws one horizontal line of the figure at a time. The value in Y1 determines which line is drawn. To draw the entire filled shape, place the top line number of the shape in Y1 and call Filled Polygon repeatedly, incrementing the value of Y1 after each call until the bottom line of the figure has been reached. Note that each call to Filled Polygon changes the value in the line A variable X1 and X2 and destroys the A0 register.

The final line A fill routine is called Seed Fill ($A00F). This function is equivalent to the VDI's Contour Fill function, which is used to fill an enclosed polygon with the current fill pattern and color. A polygon can be filled in either of two modes. In outline mode, the fill spreads from an initial point in all directions until it comes to an outline of a given color. In color mode, the fill spreads from the initial point until it reaches a color other than that of the initial point. The function uses the line A variables for input shown in Table 7-4.

Table 7-4. Line A Variables Used by Seed Fill

Offset Name	Offset	Description
CUR_WORK	−$64	Pointer to the current virtual workstation variable table. The VDI fill color index is the 16th member of this word array.
INTIN	$08	Pointer to a word array that contains the *x* and *y* coordinates of the initial fill point, in that order
PTSIN	$0C	Pointer to a word value that represents the VDI color index of the polygon outline. If this value is negative, color mode is used instead of outline mode.
WMODE	$24	Writing mode
PATPTR	$2E	Pointer to fill pattern array
PATMSK	$32	Pattern index (length − 1)
MFILL	$34	Multicolor fill pattern flag (zero = single plane, nonzero = multiplane)
CLIP	$36	Clipping flag is ignored by this function. It always clips to the rectangle described below, whether this variable is set on or off.
XMINCL	$38	Coordinate of left side of clip rectangle
YMINCL	$3A	Coordinate of top of clip rectangle
XMAXCL	$3C	Coordinate of right side of clip rectangle
YMAXCL	$3E	Coordinate of bottom of clip rectangle
SEEDABORT	$76	Pointer to a routine which is called at the end of each fill line. If this routine returns a value of zero in register D0, Seed Fill continues filling the next scan line. If this routine return a nonzero value in D0, Seed Fill aborts. At minimum, a pointer to a routine that moves a zero to D0 and then executes an RTS instruction must be placed in this variable, or Seed Fill will crash.

The Seed Fill routine operates quite similarly to its VDI counterpart. The coordinates for the initial fill point must be placed in a word array, and a pointer to that array must be stored in PTSIN. A pointer to a word flag must be placed in INTIN. If this value is negative, the color fill mode is used. If it is positive, outline mode is used, and this value is interpreted as the VDI color index number of the polygon outline. You should note that VDI color index numbers are not the same as the hardware color registers numbers. The correspondence between the two is shown in Table 7-5.

Table 7-5. Correspondence Between Hardware Color Registers and VDI Color Indices

Color Index	Color Register	Default Color
0	0	White
1	15*	Black
2	1	Red
3	2	Green
4	4	Blue
5	6	Cyan
6	3	Yellow
7	5	Magenta
8	7	Low White
9	8	Gray
10	9	Light Red
11	10	Light Green
12	12	Light Blue
13	14	Light Cyan
14	11	Light Yellow
15	13	Light Magenta

* Color register 3 in medium resolution (4-color) mode

The fill pattern and writing mode used for fill are determined by the same variables as the other line A fill routines. The color of the filled object is not specified by the COLBIT variables, however. The fill color is taken from the current VDI virtual screen workstation. If you haven't opened a VDI workstation, however, you can still use the Seed Fill function by creating a dummy virtual workstation variable table. This is just an array of 16 words, the last of which contains the VDI color index of the fill color. A pointer to this array must be stored in the CUR_WORK variable.

There are two final points to note about the Seed Fill routine. This routine always evaluates the XMINCL, YMINCL, XMAXCL and YMAXCL variables, whether or not you've set CLIP on. Therefore, make sure your clipping rectangle is set correctly before making this call. Secondly, this function calls a user-defined subroutine after each horizontal line has been filled. Since a complex fill can be a lengthy process, this hook was added to allow the programmer to abort the Seed Fill function before it finishes. If the subroutine returns a zero in register D0, Seed Fill keeps going, but if it returns a nonzero value, Seed Fill aborts. Since Seed Fill will JSR through the SEEDABORT vector at the end of each scan line, you must place a pointer to subroutine in this vector, or Seed Fill will bomb after the first line. At a minimum, this subroutine should clear register D0 and execute an RTS instruction.

Program 7-3 demonstrates all four of the line A filled shape functions.

Program 7-3. FILLDRAW.S

```
***********************************************************
*                                                         *
*     FILLDRAW.S -- Demonstrates the use of the line A    *
*                   filled shape functions.               *
*                                                         *
***********************************************************


*** Variable table offsets
CUR_WORK = -464
CONTRL   = $04
INTIN    = $08
PTSIN    = $0C
COLBIT0 = $18
COLBIT1 = $1A
COLBIT2 = $1C
COLBIT3 = $1E
LNMASK   = $22
WMODE    = $24
X1       = $26
Y1       = $28
X2       = $2A
Y2       = $2C
PATPTR   = $2E
PATMSK   = $32
MFILL    = $34
CLIP     = $36
XMINCL   = $38
YMINCL   = $3A
XMAXCL   = $3C
YMAXCL   = $3E
SEEDABORT = $76


*** Function Equates
Init      = $a000
Aline     = $a003
Hline     = $a004
RectFill = $a005
PolyFill = $a006
SeedFill = $a00f

*** Program starts here

.text
   dc.w     Init                 * get base address of variable table
   move.l   a0,a5                * save base address...
*                                * because PolyFill destroys A0

   move.w   #0,COLBIT0(a5)       * color bits set to register
   move.w   #1,COLBIT1(a5)       * 0010
   move.w   #0,COLBIT2(a5)       * or color 2 (green)
   move.w   #0,COLBIT3(a5)       *
   move.w   #0,WMODE(a5)         * replace mode
   move.l   #love,PATPTR(a5)     * set fill pattern
   move.w   #15,PATMSK(a5)       * and length of fill pattern
   move.w   #0,MFILL(a5)         * multiplane fill off
   move.w   #1,CLIP(a5)          * clipping on
   move.w   #0,XMINCL(a5)        * set clip rectangle..
```

```
    move.w   #$d0,XMAXCL(a5)     *  to (0,0)-($d0.$b0)
    move.w   #0,YMINCL(a5)       *
    move.w   #$b0,YMAXCL(a5)     *


*** draw filled box with Horizontal line function

    move.l   #$00800030,X1(a5)   * x1=$80, y1=$30
    move.w   #$0120,X2(a5)       * x2=$120
    move.w   #31,d4              * draw 32 horiz. lines

nextline:
    dc.w     Hline               * draw a line
    addq.w   #1,Y1(a5)           * go down one line
    dbra     d4,nextline         * if not done, draw next

*** draw a box using Rectangle Fill function

    move.w   #1,COLBIT0(a5)      * 1 in all color bits
    move.w   #1,COLBIT1(a5)      * = 1111
    move.w   #1,COLBIT2(a5)      * or color 15 (black)
    move.w   #1,COLBIT3(a5)      *

    move.w   #3,WMODE(a5)        * reverse transparent mode
    move.l   #$00200020,X1(a5)   * x1=$20, y1=$20
    move.l   #$00600060,X2(a5)   * x2=$60, y2=$60
    dc.w     RectFill            * draw a filled box

*** Polygon fill
    move.w   #0,WMODE(a5)        * draw with replace mode
    move.l   #pat2,PATPTR(a5)    * set solid fill pattern
    move.w   #0,PATMSK(a5)       * 1 word long

    move.l   #points,PTSIN(a5)   * addr of points array to PTSIN
    move.l   #length,CONTRL(a5)  * number of line segments to CONTRL

    move.w   #$70,Y1(a5)         * 1st (top) line to draw in Y1
    move.w   #$50,d4             * draw 80 lines
fillnext:
    dc.w     PolyFill            * fill one horizontal line
    addq     #1,Y1(a5)           * go to next line
    dbra     d4,fillnext         * if not done, fill it too

*** draw triangle outline with Arbitrary line function

    move.w   #$FFFF,LNMASK(a5)   * solid line
    move.w   #0,WMODE(a5)        * replace mode

    move.w   #2,d3               * draw 6 lines
    move.l   #points1,a3         * get address of array
drawnext:
    move.l   (a3)+,X1(a5)        * starting x & y
    move.l   (a3),X2(a5)         * ending x & y
    dc.w     Aline               * draw the line
    dbra     d3,drawnext         * if not done, draw next line

*** do a contour fill of the triangle in outline mode

    move.l   #love,PATPTR(a5)         * set old fill pattern
    move.w   #15,PATMSK(a5)           * and length of fill pattern
    move.l    #vwork,CUR_WORK(a5)     * set pointer to fill color
    move.l    #fillpt,PTSIN(a5)       * set pointer to fill coordinate
    move.l    #fillcol,INTIN(a5)      * set pointer to outline color
    move.l    #NoAbort,SEEDABORT(a5)  * set pointer to abort routine
    dc.w      SeedFill                * fill the triangle

*** do a contour fill of the filled polygon, in color mode

    move.l    #fillpt1,PTSIN(a5)    * set new fill coordinate
    move.l    #fillcol1,INTIN(a5)   * set negative fill color
    dc.w      SeedFill              * fill 'er up


*** wait for key press, then end

    move.w   #1,-(sp)            * call conin() to wait for key press
    trap     #1
    addq.l   #2,sp
```

```
   move.l    #0,-(sp)          * GEMDOS terminate command
   trap      #1                * call GEMDOS and exit


*** SEEDABORT must point to this subroutine BEFORE
*** you can use the SeedFill function

NoAbort:
   move.l    #0,d0     * return FALSE
   rts                 * and let SeedFill do the next line

*** Data for input is stored here
.data

*** fill pattern in the shape of the word "LOVE"

love:        dc.w     $0000, $3078, $30CC, $30CC
             dc.w     $30CC, $30CC, $3E78, $0000
             dc.w     $66FC, $66C0, $66C0, $32F8
             dc.w     $1EC0, $0EC0, $06FC, $0000

*** Solid fill pattern

pat2:        dc.w     $FFFF

*** Vertices for polygon fill, and number of vertices

points:      dc.w     $70, $70, $F0, $C0, $10, $90, $70, $70
length       dc.w     0,3

*** Vertices for line draw

points1:     dc.w     $80, $60, $120, $60, $120, $C0, $80, $60

*** Fill coordinates and color for outline mode fill
*** Note that the color is VDI color index, not the
*** color register number.

fillpt:      dc.w     $90, $64
fillcol:     dc.w     1

*** Fill coordinates and color for color mode fill

fillpt1:     dc.w     $70, $72
fillcol1:     dc.w     -1

*** If you haven't opened a GEM Virtual Workstation,
*** you must point CUR_WORK here before using SeedFill.
*** Since SeedFill takes the fill color from the current
*** Virtual Workstation, you must supply a dummy one
*** with the fill color in the 16th word.

vwork        dc.w     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1


.end
```

Bit Block Transfer Operations

Among the most complex and powerful of the line A functions are those that perform bit block transfer operations, known more commonly as bit blitting. These functions allow you to combine source bitplanes with destination bitplanes, using a number of different logic operations. Their main use is moving graphics objects around on the screen quickly.

The main line A bit blitting function is called BitBlt ($A007). This function requires so many input values that the programmer must place them in their own 76-byte parameter block and put the address of this block in the A6 register before making the call. The format of the BitBlt parameter block is shown in Table 7-6.

Table 7-6. The Format of the BitBlt Parameter Block

Offset	Name	Description
00–01	B_WD	Block width (in pixels)
02–03	B_HT	Block height (in pixels)
04–05	PLANE_CT	Number of consecutive bit planes
06–07	FG_COL	Foreground color (bit 1 of logic op index)
08–09	BG_COL	Background color (bit 1 of logic op index)
10–13	OP_TAB	Logic ops for all four combinations of foreground and background
14–15	S_XMIN	Left position of source rectangle
16–17	S_YMIN	Top position of source rectangle
18–21	S_FORM	Base address of source image
22–23	S_NXWD	Offset to next word in source line (in bytes)
24–25	S_NXLN	Offset to next line in source plane (in bytes)
26–27	S_NXPL	Offset from start of current source plane to next source plane (in bytes) (this value is 2 for ST screen images)
28–29	D_XMIN	Left position of destination rectangle
30–31	D_YMIN	Top position of destination rectangle
32–35	D_FORM	Base address of destination image
36–37	D_NXWD	Offset to next word in destination line (in bytes)
38–39	D_NXLN	Offset to next line in destination plane (in bytes)
40–41	D_NXPL	Offset from start of current destination plane to next destination plane (in bytes) (this value is 2 for ST screen images)
42–45	P_ADDR	Address of pattern image data buffer (0 = no pattern)
46–47	P_NXLN	Offset to next line in pattern (in bytes)
48–49	P_NXPL	Offset to next plane in pattern (in bytes)
50–51	P_MASK	Pattern index mask
52–75	SPACE	A buffer the blit operations require for temporary storage. This space must be reserved, or the memory which follows the parameter block will be mangled.

BitBlt takes a bit image from one memory area and combines it with a bit image in another memory area, according to the logic operation selected. The source and destination image rectangles are the same size. The width of this block (in pixels) is stored in B_WD, and its height is stored in B_HT, while the number of bit planes is passed in PLANE_CT (this value may be changed by the BitBlt call). The starting address of the source image block goes into S_FORM, while the starting address of the destination image block is placed in D_FORM. These image blocks must start on word boundaries.

The *x* and *y* offsets of the source image are passed in S_XMIN and S_YMIN, while the corresponding destination offsets are stored in D_XMIN and D_YMIN. These images may overlap one another without harm, but bear in mind that there is no clipping or boundary checking performed; it's up to you to make sure that the blit stays within the image memory block.

Two other values concerning the image block must also be passed. The S_NXWD and D_NXWD variables should contain the number of bytes between data words that belong to the same bit plane. For image blocks in ST screen memory, this value is 2 for the high resolution screen, 4 for medium resolution, and 8 for low resolution. S_NXLN and D_NXLN should contain the width of each line in the image block in bytes.

If the entire screen is used, this width is 80 bytes for the monochrome screen and 160 bytes for each of the color screens. S_NXPL and D_NXPL should contain byte offsets from a word in one bit plane to a word in the next bit plane. Because of the way the ST screen memory is interleaved, this value is 2 for all ST screen resolution modes.

The way source and definition blocks are combined is determined by the raster operation code (opcode) selected. There are 16 possible logic operations. These are shown in Table 7-7.

Table 7-7. The 16 Possible Logic Operations for Combining Source and Definition Blocks

Opcode	Logic Operation*	Description
0	D1 = 0	Clear destination block
1	D1 = S AND D	
2	D1 = S AND (NOT D)	

Table 7-7. The 16 Possible Logic Operations for Combining Source and Definition Blocks (continued)

Opcode	Logic Operation*	Description
3	D1 = S	Replace mode
4	D1 = (NOT S) AND D	Erase mode
5	D1 = D	Destination unchanged
6	D1 = S XOR D	XOR mode
7	D1 = S OR D	Transparent mode
8	D1 = NOT (S OR D)	
9	D1 = NOT (S XOR D)	
10	D1 = NOT D	
11	D1 = S OR (NOT D)	
12	D1 = NOT S	
13	D1 = (NOT S) OR D	Reverse transparent mode
14	D1 = NOR (S AND D)	
15	D1 = 1	Fill destination block

S is the source image, D is the destination image, and D1 is the destination image after the operation.

The BitBlt function allows you to choose a separate logic operation for each bit plane. Up to four opcodes, one byte each in length, may be stored in the OP_TAB variable. The opcode to select for a particular plane is determined by the bit value of FG_COL and BG_COL for that plane. BG_COL holds bit 0 of the opcode selection, and FG_COL holds bit 1. Let's say, for example, you've selected a FG_COL of 5 and a BG_COL of 6. The binary equivalents are

FG = 0101
BG = 0110

In this example, plane 0 will use the opcode stored in byte 2 of OP_TAB, since bit 0 of FG is 1 and bit 0 of BG is 0 (10 binary = 2). Plane 1 will use the opcode stored in byte 1 of OP_TAB, plane 2 will use the opcode stored in byte 3, and plane 3 will use the opcode stored in byte 0 of OP_TAB.

Logic operations can be confusing, particularly when they involve multiple bit planes. If you wish to transfer a single-plane source image to a multiplane destination, set S_NXPL to 0, so the same source will be moved to all destinations. To use FG_COL and BG_COL to represent actual foreground and background colors (replace mode), set OP_TAB to $F740. If you want the destination to use FG_COL for the foreground color with a transparent background color (transparent mode), set OP__TAB to $7744. Note that FG__COL and BG__COL may be changed by the BitBlt call.

The BitBlt function also allows the use of fill patterns, which are ANDed with the source prior to the logic operation. Patterns are snapped to an imaginary grid that starts at the upper left corner of the destination image block. They are 16 bits wide and repeat every 16 pixels, horizontally. Their height must be an even power of 2, (1 line, 2 lines, 4 lines, 16 lines, and so on), and they repeat at a frequency equal to their height. The variable P__ADDR points to the beginning of the pattern image data block. If it is set to 0, no pattern is used. P__NXLN is the distance between consecutive words in the pattern (in bytes) and should be a power of 2. P__NXPL is the distance (in bytes) between pattern bit planes. If a single-plane pattern is used for a multiplane destination, it should be set to 0, so the same pattern will be used for all bit planes. Finally, P__MASK is used to specify the length of the pattern. It is used in conjunction with the value in P__NXLN, which must be an even power of 2. P__MASK = (pattern length in words − 1) << n, where P__NXLN = 2^*n*. Since P__NXLN will be 2 if the pattern data is consecutive, *n* will equal 1, and P__MASK will just be the pattern length in words − 1.

Program 7-4 shows how to blit a simple image on to the screen:

Program 7-4. BITBLTS.S

```
**************************************************************
*                                                            *
*     BITBLT.S -- Demonstrates the use of the line A         *
*     BitBlt ($A007) function.                               *
*                                                            *
*                                                            *
**************************************************************


*** Function Equates
Init     = $a000
BitBlit  = $a007

*** Program starts here

.text

*** find screen address

   move.w   #2,-(sp)            * use XBIOS call #2
   trap     #14                 * to get screen address
   addq     #2,sp               * clean stack
   move.l   d0,screen           * put screen address in blit block
```

```
*** adjust blit parameters for proper resolution mode

   dc.w      Init                  * init line A
   move.w    (a0),d0               * get number of bit planes
   cmp.w     #1,d0                 * is it monochrome?
   bne       skip                  * no, leave screen width alone
   move.w    #$50,nxln             * yes, change screen width value
skip:
   lsl.w     #1,d0                 * bit planes *2
   move.w    d0,nxwd               * = value to place in nxwd

*** do the blit

   lea       blit,a6               * address of parameter table in a6
   dc.w      BitBlit               * do Bit Blit

*** wait for key press, then end

   move.w    #1,-(sp)              * call conin() to wait for key press
   trap      #1
   addq.l    #2,sp

   move.l    #0,-(sp)              * GEMDOS terminate command
   trap      #1                    * call GEMDOS and exit

*** Data for input is stored here
.data
blit:        dc.w    $0020         * width of source image in pixels
             dc.w    $0017         * height of source image in pixels
             dc.w    $0001         * number of planes to blit
             dc.w    $0001         * fg color (bit 1 of logic op index)
             dc.w    $0000         * bg color (bit 0 of logic op index)
             dc.l    $07070707     * logic ops
             dc.w    $0000         * source x
             dc.w    $0000         * source y
             dc.l    alien         * base address of source image
             dc.w    $0002         * byte offset to next word of source
             dc.w    $0004         * source width = 4 bytes
             dc.w    $0002         * source plane offset

             dc.w    $0080         * destination x
             dc.w    $0080         * destination y
screen:
             dc.l    $00000000     * screen address goes here
nxwd:                              * byte offset to next word in line
             dc.w    $0008         * 2 for hi, 4 for med, 8 for lo
nxln:                              * byte offset to next line in planes
             dc.w    $00a0         * $50 for hi, $a0 for med and lo
             dc.w    $0002         * byte offset to next plane (always 2)
             dc.l    $00000000     * addr of pattern buf (0=no pattern)
             dc.w    $0000         * byte offset to next line in pattern
             dc.w    $0000         * byte offset to next plane in pattern
             dc.w    $0000         * pattern index mask

space:                             * you must reserve 24 bytes here
             dc.w    $0000, $0000, $0000, $0000
             dc.w    $0000, $0000, $0000, $0000
             dc.w    $0000, $0000, $0000, $0000

alien:                             * image data for space creature
             dc.w    $0030, $0C00, $001C, $3800
             dc.w    $0006, $6000, $0006, $6000
             dc.w    $001F, $F800, $003F, $FC00
             dc.w    $C0FF, $FF03, $C0FF, $FF03
             dc.w    $E3E3, $C7C7, $7FEB, $D7FE
             dc.w    $3FE7, $CFFC, $03FF, $FFC0
             dc.w    $03FF, $FFC0, $00F8, $1F00
             dc.w    $00FC, $3F00, $00FF, $FF00
             dc.w    $0077, $EE00, $0030, $0C00
             dc.w    $0030, $0C00, $0030, $0C00
             dc.w    $0060, $0600, $00C0, $0300
             dc.w    $0380, $01C0

 .end
```

The Copy Raster function ($A00E) does much the same thing as BitBlt, but in a format compatible with the Copy Raster Opaque and Copy Raster Transparent functions of the VDI. Copy Opaque is used where the source and destination have the same number of bit planes, while Copy Transparent is used to combine a single-plane source with a multiplane destination. The inputs for Copy Raster are the same ones used by their VDI counterparts. For both modes, set up word arrays known as a Memory Form Definition Blocks (MFDBs), and place the addresses of these arrays in another array, whose address is placed in CONTRL (offset 4 from the line A variable table base address). Bytes 14–17 of the CONTRL array hold the address of the source MFDB, while bytes 18–21 of the array hold the address of the destination MFDB. The composition of these 20-byte data blocks is shown in Table 7-8.

Table 7-8. Composition of Memory Form Definition Blocks (MFDBs)

Byte Offset	Contents
00–03	Pointer to image data storage block
04–05	Raster image width in pixels
06–07	Raster image height in lines
08–09	Raster image width in words
10–11	Image format flag
	0 = ST specific format (must be used)
	1 = Standard GEM format
12–13	Number of color bit planes
14–15	Reserved for future use
16–17	Reserved for future use
18–19	Reserved for future use

For both modes, you must also specify the top-left and bottom-right coordinates of both the source and destination rectangles. Do this by creating an array of eight words. The first four words contain the *x* and *y* coordinates for the source rectangle (SX1, SY1, SX2, SY2), while the last four contain the coordinates for the destination rectangle (DX1, DY1, DX2, DY2).

By storing a value in the line A variable COPYTRAN (offset 116), you may select either the Copy Opaque or Copy Transparent mode. A value of 0 selects the opaque mode and any other value chooses transparent mode. For either mode, you must set up a 16-bit word that contains a logic operation value and put a pointer to that word in the line A variable

INTIN. For opaque copies, the logic opcode is a number from 0–15 which corresponds exactly to the BitBlt opcodes. Transparent copies use a logic mode number from 0–3, which corresponds to the writing modes available from the VDI. In transparent mode, the foreground color for the copy is taken from the second word in the INTIN array, and the background color from the third word in that array. For more information on the VDI bit-blit routines, see Chapter 6 of *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

Mouse and Sprite Operations

The line A sprite and mouse pointer routines are specialized cases of the bit blit functions. On the ST, the system must frequently move small graphics objects such as the mouse pointer around on the screen. To make this easier, without getting involved in all of the complexities of the BitBlt function, line A provides support for software sprites. These are graphics objects that are 16 pixels wide by 16 lines high.

Each sprite has two components, the actual image data, and a *mask*. The mask is blitted onto the destination screen first, which allows you to "scoop out" a part of that destination and change it to the background color before laying down the sprite image. This allows you to decide whether the 0 bits in the 16 × 16 image block are transparent (leave the existing destination image intact) or opaque (replace the existing destination image with the pointer background color). Therefore, any portion of the pointer may be one of three colors: foreground color, background color, or transparent.

A sprite also has an attribute known as the *hot spot*. This is the single point considered to be the sprite's location on the screen, even though the sprite may be much larger than a single point. This hot spot is expressed as an offset (in pixels) from the top left corner of the sprite.

Before the sprite is drawn on the screen, the image that occupies the space on the screen is moved to a buffer area known as the sprite save block. When the sprite is undrawn, the image is moved back from that buffer to the screen. If you are using multiple sprites, you must remember to undraw them in the reverse order of drawing, since any portion of a sprite that is covered by another sprite ends up in the second sprites save block. It's also a good idea to use the XBIOS Vsync() function before drawing sprites to avoid drawing them in the middle of a screen refresh.

The Draw Sprite function ($A00D) is used to draw a software sprite on the screen. Its inputs are all passed directly in address and data registers. These input parameters are shown in Table 7-9.

Table 7-9. Input Parameters for Draw Sprite Function

Register	Contents
D0	Horizontal sprite position
D1	Vertical sprite position
A0	Pointer to sprite definition block
A2	Pointer to sprite save block

The sprite save block is the buffer used to save the underlying screen image. Its size must be 64 bytes per bit plane, plus 10 bytes. This means that for the high resolution screen, the save block must be 74 bytes long, while the medium and low resolution screens require 138 and 266 bytes, respectively. The sprite definition block contains 37 words of data concerning the sprite. The layout of this array is shown in Table 7-10.

Table 7-10. Layout of Sprite Definition Block

Byte Number	Description
00–01	*x* offset of hot spot
02–03	*y* offset of hot spot
04–05	Drawing format flag (1 = VDI mode, −1 = XOR mode) The different colors produce by various combinations of image and mask bits for both modes are shown below:

Image	Mask	VDI mode color	XOR mode color
0	0	Transparent	Transparent
0	1	Background	Background
1	0	Foreground	XOR destination
1	1	Foreground	Foreground

Byte Number	Description
06–07	Color register for background color
08–09	Color register for foreground color
10–73	Thirty-two words of sprite image and mask data. Sprites interleave this data, so that the first two words contain the mask data and image data for line 0 of the sprite, the next two words contain the mask data and image data for line 1, and so on

To move a sprite, you must undraw it, and then draw it in a new position. The Undraw Sprite function ($A00C) is used to remove the sprite and restore the background. The only parameter it requires is a pointer to the save block, which is passed in register A2.

The mouse pointer is a special type of sprite that the operating system automatically moves in response to movement of the mouse. Line A routines allow you to hide the mouse pointer (terminate its display), show the mouse pointer (display it), or transform the pointer (change its shape). The Hide Mouse function ($A00A) requires no inputs. When you call Hide Mouse, the mouse pointer is turned off. To make it reappear, you must call Show Mouse ($A009). If Hide Mouse has been called more than once, you must call Show Mouse an equal number of times before the mouse pointer is displayed. The depth at which the mouse pointer is hidden is stored in the line A variable HIDE_CNT (offset −598). This variable holds the number of times the mouse has been hidden, or a 0 if the mouse is currently being displayed. If you wish to display the pointer regardless of how many times it has been hidden, you may set a pointer to a 16-bit 0 value in the INTIN variable (offset 8) before calling Show Mouse.

The final line A mouse function is Transform Mouse ($A00B). This function allows you to change the shape of the mouse pointer. As input, it requires that the address of a mouse pointer definition block be placed in INTIN. This mouse pointer definition block is almost exactly the same as the sprite definition block used for Draw Spite, above. The mouse pointer must be drawn using VDI mode, however. In addition, the image data and mask data are handled differently. Instead of being interleaved as in the case of sprites, these data planes are kept separate for the mouse pointer. Bytes 10–41 of the data block contain 16 consecutive words of mask data, while bytes 42–73 contain 16 consecutive words of image data.

There are several line A variable locations which may assist in dealing with the mouse pointer. The current mouse pointer definition is stored in sprite format starting at M_POS_HX (offset −856). You may save this data before changing the mouse shape, so you can restore it later. The variable MOUSE_FLAG (offset −153) determines whether the mouse interrupt processing is enabled. By changing this value to 0, you may disable mouse cursor updates while changing its shape (though you should restore it to its original value afterwards). The current *x* and *y* positions of the mouse pointer are stored in GCURX (offset −602) and GCURY (offset −600). The mouse button status is contained in MOUSE_BT (−596). Bit 0 covers the left button status, while bit one covers the right button. A 1 bit indicates that the button is currently pressed.

Program 7-5 demonstrates the use of both sprites and the mouse pointer. It copies the pointer arrow shape to a sprite, and bounces the sprite around the screen until you press the right mouse button. It also changes the shape of the mouse pointer to a cross and changes it back when you press the button.

Program 7-5. MOUSPRIT.S

```
**************************************************************
*                                                            *
*    MOUSPRIT.S -- Demonstrates use of the line A            *
*                  mouse pointer and sprite calls.           *
*                                                            *
*                                                            *
**************************************************************


*** Variable table offsets

MOUSE_BT = -596     * offset for mouse button status variable
M_POS_HX = -856     * offset for start of mouse pointer sprite data
INTIN    = $08

*** Function Equates
Init     = $a000
MShow    = $a009
MHide    = $a00a
MTrans   = $a00b
Undraw   = $a00c
Draw     = $a00d

*** Program starts here

.text
   dc.w      Init                * get base address of variable table
   move.l    a0,a5               * save base address,
*                                * in case other calls destroy A0

*** save old mouse pointer sprite data
*** both in sprite format, and pointer format
*** (pointers use separate data planes, sprites interleave them)

   move.w    #4,d0               * save 5 words of mouse data
   lea       M_POS_HX(a5),a1     * get starting address to save
   lea       psave,a2            * start address of pointer save area
   lea       ssave,a4            * & start address of sprite save area
savhead:
   move.w    (a1),(a4)+          * move 5 word header to both buffers
   move.w    (a1)+,(a2)+         * (header is same for sprites & pointers)
   dbra      d0,savhead


   move.w    #15,d0              * save 32 words of mouse data
   lea       psave+42,a3
savdata:
   move      (a1),(a4)+          * even words in sprite format
   move.w    (a1)+,(a2)+         * even words in pointer format
   move.w    (a1),(a4)+          * odd words in sprite format
   move.w    (a1)+,(a3)+         * odd words in pointer format
   dbra      d0,savdata
```

```
*** Change mouse pointer to cross shape

   move.l   #mdata,INTIN(a5)    * address of mouse data block to INTIN
   dc.w     MHide               * hide the pointer
   dc.w     MTrans              * change its shape
   move.l   #mdata+6,INTIN(a5)  * make mouse show unconditional
   dc.w     MShow               * and show the mouse pointer

*** determine x and y screen limits
   move.w   #304,d6             * right edge - pointer width
   move.w   #184,d7             * bottom edge - pointer height
   move.w   (a5),d0             * find number of bit planes in display
   cmp.w    #4,d0               * is it low-res?
   beq      narrow              * yes, keep right limit
   add      d6,d6               * no, double screen width
narrow:
   cmp      #1,d0               * is it high res?
   bne      short               * no, keep bottom limit
   add      d7,d7               * yes, double screen height
short:

*** set up sprite move
   move.w   d6,d0               * initial x position = right limit
   asr.w    #1,d0               * divided by 2.
   move.w   d0,a3               * save x position
   move.w   #1,d4               * initial x increment = 1
   move.w   d4,d5               * initial y increment = 1
   move.w   d4,a4               * initial y position = 1

sprite:

   move.w   a3,d0               * get x position
   move.w   a4,d1               * get y position

   add.w    d4,d0               * increment x position
   beq      xchange             * if at left, change directions
   cmp      d6,d0               * at right edge?
   bcs      no_xchange          * no, keep going
xchange:
   eori.w   #$FFFE,d4           * yes, change direction
no_xchange:
   add.w    d5,d1               * increment y position
   beq      ychange             * if at top, change directions
   cmp      d7,d1               * at bottom?
   bcs      no_ychange          * no, keep going
ychange:
   eori.w   #$FFFE,d5           * yes, change direction
no_ychange:
   move.w   d0,a3               * save x position
   move.w   d1,a4               * save y position
   move.w   #37,-(sp)           * wait for end of vertical blank
   trap     #14
   addq.l   #2,sp

   lea      savebuf,a2          * use save buffer pointer,
   dc.w     Undraw              * erase the current sprite

   lea      savebuf,a2          * use save buffer pointer,
   lea      ssave,a0            * use the old mouse pointer shape
   move.w   a3,d0               * get x position
   move.w   a4,d1               * y position
   dc.w     Draw                * and draw sprite in new position


*** wait for mouse button press

   move.w   MOUSE_BT(a5),d3     * check button status
   btst     #1,d3               * is right button down?
   beq      sprite              * no, move sprite

*** When right mouse button is pressed,
*** restore mouse pointer to original shape

   move.l   #psave,INTIN(a5)    * address of saved mouse data to INTIN
   dc.w     MHide
   dc.w     MTrans
   dc.w     MShow
```

```
*** And wait for mouse button release

wait:
   move.w   MOUSE_BT(a5),d3
   btst     #1,d3
   bne      wait

*** end program
   move.l   #0,-(sp)            * GEMDOS terminate command
   trap     #1                  * call GEMDOS and exit

*** Data for input is stored here
.data

*** mouse pointer data

mdata:          dc.w      8,8,1,0,2
                dc.w      $0FF0, $0FF0, $0FF0, $FFFE
                dc.w      $FFFE, $FFFE, $FEFE, $FC7E
                dc.w      $FEFE, $FFFE, $FFFE, $FFFE
                dc.w      $0FF0, $0FF0, $0FF0, $0000

                dc.w      $0000, $0000, $0380, $0380
                dc.w      $0380, $3FF8, $3EF8, $3C78
                dc.w      $3EF8, $3FF8, $0380, $0380
                dc.w      $0380, $0000, $0000, $0000

*** storage for old mouse data and sprite save area

.bss

psave:
                ds.w      37

ssave:          ds.w      37

savebuf:        ds.w      133  * enough for 4 bit planes

.end
```

If you position the mouse pointer over the sprite while it's moving, a bit of residue will be left on the screen. That's because part of the pointer is saved to the sprite's save block, and restored when the sprite moves on. To avoid this problem, try hiding the mouse pointer before undrawing the sprite, and showing it again after drawing the sprite.

Text

Since text characters on the ST are really drawn on the graphic screen, dot by dot, text rendering turns out to be another special case of bit blitting. The function used to copy a single text character to the screen is called TextBlt ($A008). Because there are many variations on text printing on the ST, TextBlt uses a lot of input variables. The primary line A variables that influence text printing are shown in Table 7-11.

Table 7-11. The Primary Line A Variables That Influence Text Printing

Variable Name	Offset	Description
WMODE	$24	Writing mode (0–3 = VDI modes, 4–19 = BitBlt modes)
CLIP	$36	Clipping flag (0 = off, 1 = on)
XMINCL	$38	Left edge of clip rectangle
XMAXCL	$3A	Right edge of clip rectangle
YMINCL	$3C	Top of clip rectangle
YMAXCL	$3E	Bottom of clip rectangle
XDDA	$40	Accumulator for text scaling. Should be set to $8000 each time you do a TextBlt that requires scaling.
DDAINC	$42	Scaling increment. For scaling up, DDAINC = 256 * Size2 − Sizel) / Sizel. For scaling down, DDAINC, = 256 * (Size2) / Sizel, where Size 1 is the actual character point size, and Size 2 is the scaled character size.
SCALDIR	$44	Text scaling direction (0 = down, 1 = up)
MONO	$46	Monospaced font flag. 0 = font is proportional, or size may vary due to special effects 1 = font is monospaced, and uses no special effects other than thickening (boldface)
SOURCEX	$48	*x* coordinate of character to be printed within font data table
SOURCEY	$4A	*y* coordinate of character to be printed within font data table
DESTX	$4C	*x* coordinate of text character on screen
DESTY	$4E	*y* coordinate of text character on screen
DELX	$50	Width of character
DELY	$52	Height of character
FBASE	$54	Pointer to font data table
FWIDTH	$58	Font form width (sum of the widths of all of the characters in the font, in bytes)
STYLE	$5A	Special effects 1 = bold 2 = light 4 = italics 8 = underline 16 = outline (TextBlt does not do underlining)
LITEMASK	$5C	Mask used to lighten text (usually $5555)

Table 7-11. The Primary Line A Variables That Influence Text Printing (continued)

Variable Name	Offset	Description
SKEWMASK	$5E	Mask used to italicize (usually $5555)
WEIGHT	$60	Width by which to thicken text for boldface
ROFF	$62	Offset above baseline for italicizing
LOFF	$64	Offset below baseline for italicizing
SCALE	$66	Scaling flag (0 = no scaling)
CHUP	$68	Character rotation (0 = no rotation, 900 = 90 degree rotation, and so on)
TEXTFG	$6A	Text foreground color
SCRTCHP	$6C	Pointer to first of two contiguous special effects buffers
SCRPT2	$70	Offset from SCRTCHP to beginning of second special effects buffer
TEXTBG	$72	Text background color

The information you must pass to TextBlt can be broken down into four general categories:

- Information about the font used to print the character (location, size, and so on)
- Information about the character to be printed (which character, at what position, what color, and so on)
- Information about special effects
- Information about text scaling

The first thing needed to print text character is the address of a VDI-style text font. If you wish to use one of the system fonts, get this information from the line A Initialization call. As stated before, this call returns the address of a pointer table. This table consists of pointers to the font headers for the system fonts.

Currently, there are three system fonts on the ST. The first entry in the header table is the address of the 6 × 6 system font header, followed by the address of the headers for the 8 × 8 system font and the 8 × 16 system font.

The font header is an 87-byte data block of information about the font. Much of the information which must be placed in the line A variables for TextBlt may be derived from the font header. For example, the address of the font's character image data block must be placed in the line A variable FBASE.

This address may be found at an offset of 76 bytes from the beginning of the font header. Likewise, you must place the number of bytes required to hold the combined widths of all characters in the font in the variable FWIDTH. This information may be found in the word that starts at an offset of 80 from the font header. You must also tell TextBlt if the font is monospaced (all characters have the same width) by placing a 1 in the variable MONO if it's monospaced, and a 0 if it isn't. This information may be taken from bit 3 of the word that starts at an offset of 66 from the beginning of the font header.

If you're using a system font, however, you need not do this, since all system fonts are normally monospaced. Note, however, that the MONO flag should not be set if you are using special effects that may vary the width of the characters. Some of the more useful information found in the header file is shown in Table 7-12 below. For complete information on VDI fonts and font headers, see Appendix C of *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

Table 7-12. Information Found in the Font Header

Name	Offset	Description
first_ade	36	ASCII value of first displayable character in the font
left_offset	54	Number of pixels added to left by special effects
right_offset	56	Number of pixels added to right by special effects
thick_width	58	Number of pixels added to width by thickening (bold) special effect
lite_mask	62	Mask used for lightening effect
skew_mask	64	Mask used for italicizing
off_table	72	Pointer to text data offset table
data_table	76	Pointer to font data table
form_width	80	Total width of font
form_height	82	Total height of font

When printing a character with TextBlt, you must supply information about the character to be printed. As with any blit operation, you must specify the top left corner of the source and destination rectangles, and the width and height of the rectangle to blit. The source *x* coordinate may be found using the font header's character offset table, a pointer to which is found at off_table, 72 bytes from the start of the header.

This table contains each character's offset from the beginning of the font data table. By taking the ASCII value of the character to print, subtracting the ASCII value of the first printable character in the font (which is also found in the header), and multiplying by 2, you come up with the beginning *x* offset for the character. SOURCEY is usually set to 0, indicating that you wish to print the character from the top line. DESTX and DESTY are set to the *x* and *y* screen position at which you wish the character to be printed.

If you're using the system fonts, text printing will be significantly faster if characters are printed on byte boundaries, since the pixels of the text characters won't have to be shifted. The height of the blit rectangle goes in DELY and can be taken from the text header offset form_height. To find the width of the character, you may again use the off_table value used to find the character's *x* position. By subtracting the *x* position of the next character from that of the current character, you can determine the width of the current character.

Once you've specified the size and position of the source and destination rectangles, you're ready to specify the blit mode. TextBlt allows you to use either the VDI blit modes or the BitBlt modes. A value of 0–3 in WMODE indicates one of the VDI modes, while a value of 4–19 specifies of the BitBlt modes. If using the VDI modes, the foreground and background colors for the text are stored in TEXTFG and TEXTBG. The normal clipping variables may be used to set a clipping rectangle for TextBlt.

Text characters may be printed out normally, or special effects may be added to alter their appearance, making them print out in boldface, italics, lightened, or outlined. Various bits in the STYLE variable are used to select these effects (see the input parameter table, above).

If you're using special effects, set aside a scratch buffer that TextBlt can use to distort the character. This buffer should be twice as large as the width of the distorted character. A pointer to the buffer should be placed in SCRTCHP and the offset to the second half of the buffer placed in SCRPT2.

Depending on the effect, you may need to set some other variables as well. For skewing (italics), set SKEWMASK to the value found in the font header variable SKEWMASK. Also set LOFF and ROFF from the font header variables left_offset and right_offset. For thickening (boldface), set

WEIGHT from the font header variable thick_width. For lightening, set LITEMASK to the value found in the font header at lite_mask. Text can also be rotated in increments of 90 degrees. If CHUP is set to 0, the text will be printed normally, but if set to a multiple of 900, the text will be rotated 90, 180, or 270 degrees.

The normal appearance of text may also be altered through *font scaling*. This allows you to print characters either larger or smaller than normal. To use font scaling, place a nonzero value in the SCALE variable. The scaling accumulator, XDDA, should be initialized to a value of $8000 before each call to TextBlt when scaling.

The SCALDIR variable should be set to indicate whether you are scaling the character up (1) or down (0). Finally, the fractional amount to scale the character should be stored in the DDAINC variable. When scaling down, DDAINC should be set to a value equal to 256 times the scaled size divided by the actual size of the character. When scaling up, set DDAINC to 256 times the scaled size minus the actual size, divided by the actual character size.

Program 7-6 shows how to print a text string with TextBlt, using the skewing special effect.

Program 7-6. TEXTBLT.S

```
************************************************************
*                                                          *
*   TEXTBLT.S -- Demonstrates use of the line A            *
*                text blit routine to print text.          *
*                                                          *
*                                                          *
************************************************************


*** line A variable table offsets

WMODE      = $24   * Writing mode (0-3=VDI modes, 4-19=Bitblt modes)
CLIP       = $36   * Clipping flag (0=off, 1=on)
XMINCL     = $38   * Next four specify clipping rectangle
XMAXCL     = $3A
YMINCL     = $3C
YMAXCL     = $3E
XDDA       = $40   * Accumulator for text scaling
DDAINC     = $42   * Fractional scaling amount
SCALDIR    = $44   * Text scaling direction (0=down)
MONO       = $46   * Monospaced font? (0=no, 1=yes)
SOURCEX    = $48   * X coordinate of character within font data table
SOURCEY    = $4A   * Y coordinate of character within font data table
DESTX      = $4C   * X coordinate of text character on screen
DESTY      = $4E   * Y coordinate of text character on screen
DELX       = $50   * Width of character
DELY       = $52   * Height of character
FBASE      = $54   * Pointer to font data table
FWIDTH     = $58   * Font form width (width of all characters, in bytes)
STYLE      = $5A   * Special effects flag (0=bold, 2=light, 4=italics)
LITEMASK   = $5C   * Mask used to lighten text (usually $5555)
```

```
SKEWMASK   = $5E  * Mask used to italicize (usually $5555)
WEIGHT     = $60  * Width by which to thicken text for boldface
ROFF       = $62  * Offset above baseline for italicizing
LOFF       = $64  * Offset below baseline for italicizing
SCALE      = $66  * Scaling flag (0=no scaling)
CHUP       = $68  * Character rotation (0=no rotation)
TEXTFG     = $6A  * Text foreground color
SCRTCHP    = $6C  * Ptr to 1st of 2 contiguous special effects buffers
SCRPT2     = $70  * Offset to beginning of second buffer
TEXTBG     = $72  * Text background color

*** Font Header offsets

first_ade    = 36  * ASCII value of first displayable character
left_offset  = 54  * Number of pixels added to left by effects
right_offset = 56  * Number of pixels added to right by effects
thick_width  = 58  * Number of pixels added to width by bold effect
lite_mask    = 62  * Mask used for lightening effect
skew_mask    = 64  * Mask used for italicizing
off_table    = 72  * pointer to text data offset table
data_table   = 76  * pointer to font data table
form_width   = 80  * total width of font
form_height  = 82  * total height of font

*** Function Equates
Init    = $a000
Textblt = $a008

*** Program starts here

.text
   dc.w     Init                  * get base address of variable table
   move.l   a0,a5                 * save base address...
*                                 * because Textblt destroys a0

*** Initialize text blit data

   move.w   #0,WMODE(a5)          * use VDI mode 0 (replace)
   move.w   #0,CLIP(a5)           * Clipping off
   move.w   #1,TEXTFG(a5)         * Foreground color = red
   move.w   #0,TEXTBG(a5)         * Background color = white
   move.w   #48,DESTX(a5)         * Print starts at 48 horizontal
   move.w   #80,DESTY(a5)         * and 80 vertical
   move.w   #0,SCALE(a5)          * No font scaling used

   move.w   #4,STYLE(a5)           * use skew (italics) special effect
   move.w   #0,MONO(a5)            * not monospaced, 'cuz skewed
   move.l   #buffer,SCRTCHP(a5)  * ptr to effects scratch buffer1
   move.w   #6,SCRPT2(a5)          * offset to scratch buffer2

*** Initialize font table info

   move.l   4(a1),a4              * get font header base address
   move.l   data_table(a4),FBASE(a5)  * get font data table address
   move.w   form_width(a4),FWIDTH(a5) * get font width total
   move.w   form_height(a4),DELY(a5)  * get font height total

   move.w   left_offset(a4),LOFF(a5)   * width added to left by effects
   move.w   right_offset(a4),ROFF(a5)  * width added to right by effects
   move.w   skew_mask(a4),SKEWMASK(a5)  * get skewing mask
   move.w   thick_width(a4),WEIGHT(a5)  * width added by bold effect
   move.w   lite_mask(a4),LITEMASK(a5)  * get lightening mask


*** Print the string

   lea.l    string,a6               * get string address
   move.l   off_table(a4),a3        * and address of x offset table
   clr.l    d0                      * initialize d0
blit_one:
   move.b   (a6)+,d0                * get character from string
   beq      exit                    * if at end, exit
   sub.w    first_ade(a4),d0        * get offset of this letter in font
   lsl.w    #1,d0                   * multiply by 2 for word offset
   move.w   0(a3,d0),SOURCEX(a5)    * get x location of this character
   move.w   2(a3,d0),d0             * x location of next character..
   sub.w    SOURCEX(a5),d0          * minus x location of this character
   move.w   d0,DELX(a5)            * = width of this character
   clr.w    SOURCEY(a5)            * start at top line of character
   dc.w     Textblt                * print the character
   bra      blit_one               * then do the next one

*** wait for key press, then end

exit:
   move.w   #1,-(sp)           * call conin() to wait for key press
   trap     #1
   addq.l   #2,sp

   move.l   #0,-(sp)           * GEMDOS terminate command
   trap     #1                 * call GEMDOS and exit

*** String data is stored here
.data

string:      dc.b     "I'm feeling a bit skewed today...",0

*** Temporary storage area for special effects buffer
.bss

buffer       ds.w    6


.end
```

Appendix A

BIOS Functions

The ST BIOS routines can be called from user mode, and are reentrant to three levels. They use registers A0–A2 and D0–D2 as scratch registers, which means that if you're programming in assembly language, and you're using these registers to store important information, you must save their contents before making a BIOS call, and restore them after the BIOS call. Each of the BIOS routines is associated with a command number (called an opcode), and, optionally, command parameters that specify more precisely what it should do. For example, the BIOS function to output a character to a device is command number 3. It requires two command parameters: One tells the function which character to print and the other specifies the output device to use.

To call a BIOS function from machine language, you must push the command parameters onto the stack, followed by the command number, and execute a TRAP #13 statement. The TRAP #13 instruction puts the program into supervisor mode, and begins executing the instructions found at the address stored in exception vector 45, whose address is 180 ($B4). This exception vector contains the address of the BIOS handler, which reads the command number on the top of the stack, and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register D0. When a BIOS function call is completed, it's the responsibility of the calling program to adjust the stack in order to remove the command parameters and command number. You should note that the BIOS changes the command number and return address on the stack.

The following program fragment demonstrates sending the character *X* out to the console device using BIOS command number 3:

```
move.w   #'X',-(sp)   * push character value on stack
move.w   #2,-(sp)     * push console device number on stack
move.w   #3,-(sp)     * push BIOS command number on stack
trap     #13          * call BIOS handler
addq.l   #6,sp        * pop parameters (6 bytes) off stack
```

Calling the BIOS routines from C is much simpler. Most C compilers come with a library routine called bios( ), which stacks the parameters and executes the TRAP #13 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement

```
bios(3,2,'X');
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OSBIND.H that defines macros for all of the BIOS functions. For example, the macro definition for BIOS command 3 is

```
#define Bconout(a,b)   bios(3,a,b)
```

Therefore, after you #include OSBINDS.H in your program, you can call the sample function like this:

```
Bconout(2,'X');
```

To use BIOS functions in your C programs, you must #include OSBIND.H if you use the macros, and you must link your program with the library that contains the bios( ) function.

Get Memory Parameter Block

Getmpb( ) **Opcode = 0**

This call is used by GEMDOS to initialize the memory management system. It creates a data structure that contains memory managment information.

C macro format

```
long buffer[3];
   getmpb(buffer);
```

Machine language format

```
move.l      buffer,-(sp)
move.w      #0,-(sp)
trap        #13
addq.l      #6,sp
```

Inputs

buffer	long	The address of a 12-byte buffer

Results

buffer[0]	long	Pointer to memory free list MD
buffer[1]	long	Pointer to memory allocated list MD
buffer[2]	long	Pointer to roving MD

All three of these pointers point to memory descriptor (MD) data structures. The composition of these structures is

link	long	Pointer to next MD [NULL]
start	long	Pointer to start address of the block
length	long	Length of the block in bytes
own	long	Pointer to MD owner's process descriptor [NULL]

Get Input Device Status

Bconstat( ) **Opcode = 1**

This function allows you to determine whether there is a character waiting to be received from a particular input device. Since the Conin( ) call doesn't return until a character has been received, Bconstat( ) can be used to insure that a call to Conin( ) will return immediately.

C macro format

```
int devnum;
long status;
   status = Bconstat(devnum);
```

Machine language format

```
move.w      #devnum,-(sp)
move.w      #1,-(sp)
trap        #13
addq.l      #4,sp
```

Inputs

devnum	word	The device number of the input device to check: 1 = AUX: (RS-232 port) 2 = CON: (Console keyboard) 3 = MIDI: (MIDI input)

Results

D0	status	long	Input status of the device 0 = no characters ready −1 = one or more characters ready

See also

Bconin( ), Bcostat( ), Bconout( )

Read a Character

Bconin( ) **Opcode = 2**

Bconin( ) waits for a single character to become available from one of the input devices, and then reads that character. If a character is available at the time the call is made, this function reads it and returns immediately. Otherwise, it waits until a character is received.

C macro format

```
int devnum;
long char;
   char = Bconin(devnum);
```

Machine language format

```
move.w      #devnum,-(sp)
move.w      #2,-(sp)
trap        #13
addq.l      #4,sp
```

Inputs

devnum	word	The device number of the input device from which to receive the character: 1 = AUX: (RS-232 port) 2 = CON: (Console keyboard) 3 = MIDI: (MIDI input)

Results

D0	char	long	The ASCII character received from the device is returned in the least significant byte of the long. For the console device, the least significant byte of the high word contains a key code that specifies the physical key that was pressed. See Appendix J for a complete list of scan codes.

See also

Kbshift( ), Bconstat( ), Bconout( ), Bcostat( )

Write a Character

Bconout( ) **Opcode = 3**

This function sends a single character to one of the output devices. It doesn't return until the character is actually sent. To avoid sending a character to a device that isn't ready to receive it, and thus "hanging" your program, you can first test the status of the output device with the Bcostat( ) function.

C macro format

```
int devnum, char;
   Bconout(devnum,char);
```

Machine language format

```
move.w      #char, -(sp)
move.w      #devnum, -(sp)
move.w      #3, -(sp)
trap        #13
addq.l      #6,sp
```

Inputs

devnum	word	The device number of the output device to which the character is sent: 0 = PRN: (Printer port) 1 = AUX: (RS-232 port) 2 = CON: (Console screen) 3 = MIDI: (MIDI output) 4 = IKDB: (intelligent keyboard controller)

Results

None

See also

Bcostat( ), Bconin( ), Bconstat( )

Read/Write Disk Sectors

Rwabs( ) **Opcode = 4**

This function allows you to read or write to the disk, a sector at a time.

C macro format

```
int mode, sectors, start, drivenum;
long buffer, status;
   status = Rwabs(mode, buffer, sectors, start, drivenum);
```

Machine language format

```
move.w      #drivenum, -(sp)
move.w      #start, -(sp)
move.w      #sectors, -(sp)
move.l      buffer, -(sp)
move.w      #mode, -(sp)
move.w      #4, -(sp)
trap        #13
addq.l      #14,sp
```

Inputs

drivenum	word	The number of the drive to use for the read or write operation (0 = drive A:, 1 = drive B:, and so on).
start	word	The starting sector number for the transfer
sectors	word	The number of sectors to be transferred
buffer	long	A pointer to the memory area used as a disk sector buffer. This buffer should start at an even address, and should have 512 bytes allocated to it for each sector to be read or written.
mode	word	A flag that indicates whether you wish to read or write sectors: 0 = Read sectors 1 = Write sectors 2 = Read sectors without affecting media change status 3 = Write sectors without affecting media change status

Results

D0	status	long	An error code that indicates whether the transfer was successful. A zero status means no error occurred. Otherwise, a negative GEMDOS error number is returned. See Appendix D for a list of GEMDOS error codes.

Read/Change Exception Vector

Setexec( ) **Opcode = 5**

This function allows you to read or change one of the 68000 exception vectors.

C macro format

```
int vecnum;
long vecaddr, oldaddr;
oldaddr = Setexec(vecnum, vecaddr);
```

Machine language format

```
move.l      vecaddr,-(sp)
move.w      vecnum,-(sp)
move.w      #5,-(sp)
trap        #13
addq.l      #8,sp
```

Inputs

vecaddr	long	The address of the new exception handler routine to use for this vector. A value of −1 indicates that you just wish to read the current vector address.
vecnum	word	The number of the vector to read or change. Since each vector is four bytes long, the address = 4 * vector number

Results

D0	oldaddr	long	The address stored in the vector before the call was made. This address should be saved, so your program can restore it before terminating.

Get Timer Calibration

Tickcal( ) **Opcode = 6**

This call returns the number of milliseconds between timer tick interrupts. For the current ST models, this value is 20 milliseconds.

C macro format

```
long ticklen;
   ticklen = Tickcal( );
```

Machine language format

```
move.w #6,-(sp)
trap #13
addq.l #2,sp
```

Inputs

None

Results

D0	ticklen	long	The number of milliseconds between system timer updates.

Get BIOS Parameter Block

Getbpb( ) **Opcode = 7**

This function returns a pointer to the BIOS Parameter Block, a data structure that contains information about a disk's size and layout.

C macro format

```
int drivenum;
long blockaddr;
    blockaddr = Getbpb(drivenum);
```

Machine language format

```
move.w      #drivenum,-(sp)
move.w      #7,-(sp)
trap        #13
addq.l      #4,sp
```

Inputs

drivenum	word	The number of the drive whose BPB you wish to read (0 = drive A:, 1 = drive B:, and so on).

Results

D0	blockaddr	long	The starting address of the BIO Parameter Block.

The Parameter Block is a data structure that contains nine words:

Word	*Name*	*Description*
0	recsiz	Number of bytes per sector (must be 512 under current GEMDOS)
1	clsiz	Number of sectors per cluster (must be 2 under current GEMDOS)
2	clsizb	Number of bytes per cluster (must be 1024 under current GEMDOS)
3	rdlen	Root directory length (in sectors)
4	fsiz	File Allocation Table (FAT) size (in sectors)
5	fatrec	Sector number of the start of second FAT
6	datrec	Sector number of the first data cluster
7	numcl	Number of data clusters on the disk
8	bflags	Bit flags. Currently only bit 0 is used. When set, it indicates 16-bit FAT entries instead of the usual 12-bit entries.

Get Output Device Status

Bcostat( ) **Opcode = 8**

Bcostat( ) tells you whether a particular output device is ready to accept a character. It can be used to avoid "hanging" your program up by sending a character to one of the output routines when the device is not ready.

C macro format

```
int devnum;
long status;
   status = Bcostat(devnum);
```

Machine language format

```
move.w      #devnum,-(sp)
move.w      #8,-(sp)
trap        #13
addq.l      #4,sp
```

Inputs

devnum	word	The device number of the input device whose output status is tested: 0 = PRN: (Printer port) 1 = AUX: (RS-232 port) 2 = CON: (Console screen) 3 = MIDI: (MIDI output) 5 = IKBD: (Intelligent keyboard controller)

Results

D0	status	long	The output status of the device: 0 = not ready to receive a character −1 = ready to receive a character

See also

Bconout( ), Bconin( ), Bconstat( )

Get Media Change Status

Mediach( ) **Opcode = 9**

This function is used by the disk operating system to determine whether a disk has been changed.

C macro format

```
int drivenum;
long status;
   status = Mediach(drivenum);
```

Machine language format

```
move.w      #drivenum, -(sp)
move.w      #9, -(sp)
trap        #13
addq.l      #4,sp
```

Inputs

drivenum	word	The number of the drive to check (0 = drive A:, 1 = drive B:, and so on).

Results

D0	status	long	The media change status of the drive. 0 = Disk was definitely changed. 1 = Disk might have been changed. 2 = Disk definitely was not changed.

Find Valid Drive Numbers

Drvmap( ) **Opcode = 10**

Drvmap( ) may be used to discover what disk drives are currently attached.

C macro format

```
long drives;
   drives = Drvmap( );
```

Machine language format

```
move.w      #10,-(sp)
trap        #13
addq.l      #2,sp
```

Inputs

None

Results

D0	drives	long	A bit flag that indicates which drives are connected. Each bit corresponds to a different drives, with bit 0 standing for drive A:, bit 1 for drive B:, and so on. A 1 in a bit position means the drive is connected, while a 0 means it isn't. If drive A: is present, drive B: is always present also, because if there is no physical B drive, the system uses drive A: as a logical drive B:.

Read/Change Keyboard Shift Status

Kbshift( ) **Opcode = 11**

This function returns information about the status of the special shift keys, including Control and Alternate. It can also be used to change the shift status.

C macro format

```
int shiftcode, mode;
   shiftcode = Kbshift(mode);
```

Machine language format

```
move.w      #mode,-(sp)
move.w      #11,-(sp)
trap        #13
addq.l      #4,sp
```

Inputs

mode	word	*Mode* is a flag that indicates whether you wish to read or set the shift status. A negative value in mode requests a status read. A positive value causes the function to set the status code to the value indicated, after reading the current code value.

Results

D0	shiftcode	long	A bit flag that indicates which of the shift keys are currently pressed. A bit that's set to 1 indicates that the key was pressed when the call was made.

Note that since keys read with Bconin( ) are buffered, the shift status at the time Kbshift( ) is called is not necessarily the same as when the last character was read with Bconin( ). The bit assignments are:

Bit Number	*Bit Value*	*Shift Key*
0	1	Right shift key
1	2	Left shift key
2	4	Control key
3	8	Alternate key
4	16	Caps lock on
5	32	Alternate-Clr/Home key combination (keyboard equivalent for right mouse button)
6	64	Alternate-Insert key combination (keyboard equivalent for left mouse button)
7	128	Reserved (currently 0)

See also

Bconin( )

Appendix B

XBIOS Functions

Like the BIOS functions, the XBIOS routines can be called from user mode. They use registers A0–A2 and D0–D2 as scratch registers, which means if you're programming in machine language and your program uses these registers, you must save their contents before making an XBIOS call and restore them after the XBIOS call terminates. Each of the XBIOS routines is associated with a command number and, optionally, command parameters that specify more precisely what it should do. For example, the XBIOS function to set one of the hardware color registers has a command number of 7. It requires two command parameters: One tells the function which register to set and the other specifies the new color value (from 0 to 0x777).

To call an XBIOS function from machine language, you must push the command parameters onto the stack, followed by the command number, and execute a TRAP #14 statement. The TRAP #14 instruction puts the program into supervisor mode, and begins executing the instructions found at the address stored in exception vector 46, whose address is 184 ($B8). This exception vector contains the address of the XBIOS handler that reads the command number on the top of the stack, and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register D0. When an XBIOS function call is completed, the calling program has the responsibility to adjust the stack in order to remove the command parameters and command number.

The following program fragment demonstrates how you would change the value of color register 0 (the background color) to yellow ($770) using BIOS command number 7:

```
move.w   #$770,-(sp)   * push color value on stack
 move.w  #0,-(sp)       * push color register number on stack
move.w   #7,-(sp)      * push XBIOS command number on
                         stack
trap     #14           * call XBIOS handler
addq.l   #6,sp         * pop parameters (6 bytes) off stack
```

Calling the XBIOS routines from C is much simpler. Most C compilers come with a library routine called xbios( ), that stacks the parameters and executes the TRAP #14 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement

```
xbios(7,0,0x770);
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OSBIND.H.H that defines macros for all of the XBIOS functions. For example, the macro definition for XBIOS command 7 is:

```
#define Setcolor (a,b)   xbios(7,a,b)
```

Therefore, after you #include OSBIND.H.H in your program, you can call the sample function like this:

```
Setcolor(0,0x777);
```

The C macro format is the one most commonly found in discussions of XBIOS routines and sample programs. To use XBIOS functions in your C programs, you must #include OSBIND.H if you use the macros, and you must link your program with the compiler library that contains the xbios( ) function.

Initialize Mouse

Initmous( ) **Opcode = 0**

This function lets you send the intelligent keyboard controller all of the commands required to initialize the mouse packet mode. This function is more for internal use by TOS than for the application programmer's benefit, since in most cases, the programmer will let the system control the mouse.

C macro format

```
int mode;
long params, vector;
   Initmous(mode, params, vector);
```

Machine language format

```
move.l      vector, -(sp)
move.l      params, -(sp)
move.w      #mode, -(sp)
move.w      #0, -(sp)
trap        #14
addq.l      #12,sp
```

Inputs

vector	long	Point to a new mouse packet interrupt handler to support the new mouse packet mode.
params	long	Pointer to a 12-byte data block containing the parameters needed for mouse packet initialization.

Contents of data block pointed to by *params:*

Byte Offset	*Label*	*Description*
0	topmode	Specifies origin of *y* position 0 = *y* origin (0 point) at bottom 1 = *y* origin at top
1	buttons	The parameter for the IKBD set mouse buttons command
2	xparam	In relative mode, *x* threshold In absolute mode, *x* scale In keycode mode, *x* delta
3	yparam	In relative mode, *y* threshold In absolute mode, *y* scale In keycode mode, *y* delta

The following are used only in mouse-absolute mode:

4	xmax	Maximum *x* position of mouse
6	ymax	Maximum *y* position of mouse
8	xinitial	Initial *x* position of mouse
10	yinitial	Initial *y* position of mouse

mode	word	A flag that specifies the type of mouse information packets the IKBD controller is to send. The various modes are

Mode Number	*Mouse Mode*
0	Mouse disabled
1	Mouse enabled in relative mode
2	Mouse enabled in absolute mode
3	Unused
4	Mouse enabled in keycode mode

Results

None

Get Screen RAM Physical Base Address

Physbase( ) **Opcode = 2**

This function returns the starting address of screen display memory.

C macro format

```
long scraddr;
   scraddr = Physbase( );
```

Machine language format

```
move.w      #2,-(sp)
trap        #14
addq.l      #2,sp
```

Inputs

None

Results

D0	scraddr	long	A pointer to the beginning of the 32K memory block that is currently displayed.

See also

Logbase( ), Setscreen( )

Get Screen RAM Logical Base Address

Logbase( ) **Opcode = 3**

This function returns the beginning address of logical screen display memory. The logical screen is the area in memory that TOS graphics functions writes to. This is usually the same area as the physical display, but can be changed to update a graphics display without letting the user see the update process.

C macro format

```
long logscraddr;
   logscraddr = Logbase( );
```

Machine language format

```
move.w        #3,-(sp)
trap          #14
addq.l        #2,sp
```

Inputs

None

Results

D0	logscraddr	long	The starting address of the logical screen (the 32K memory area to which graphics functions write).

See also

Physbase( ), Setscreen( )

Get Screen Resolution Mode

Getrez( ) **Opcode = 4**

Getrez( ) can be used to determine the current display mode. The three display modes currently supported by the ST are lo-res (320 × 200, 16 colors), medium-res (640 × 200, 4 colors) and hi-res (640 × 400, black and white).

C macro format

```
int rez;
    rez = Getrez( );
```

Machine language format

```
move.w          #4,-(sp)
trap            #14
addq.l          #2,sp
```

Inputs

None

Results

D0	rez	word	A flag that indicates the current display resolution: 0 = Low resolution 1 = Medium resolution 2 = High resolution

See also

Setscreen( )

Set Screen Parameters

Setscreen( ) **Opcode = 5**

The Setscreen( ) function allows the programmer to change the physical screen address, the logical screen address, and/or the display resolution mode. There are some constraints on these changes, however. Both the physical and logical screen addresses must begin on an even page (256-byte) boundary. Changing the resolution mode only works if the program is running on a color monitor, since TOS won't let you select a color mode on a monochrome screen, or vice versa. And on color monitors, resolution switching with Setscreen( ) only works for TOS programs, since there is no way of telling GEM to adjust to a new screen mode, other than by using the Set Preferences menu option of the Desktop. Note that when you change resolution modes, the screen is cleared and certain other screen parameters are reinitialized.

C macro format

```
int rez;
long logaddr, physaddr;
   Setscreen(logaddr, physaddr, rez);
```

Machine language format

```
move.l      #res, -(sp)
move.l      physaddr, -(sp)
move.l      logaddr, -(sp)
move.w      #5, -(sp)
trap        #14
addq.l      #12,sp
```

Inputs

rez	word	The new display resolution mode 0 = Low resolution 1 = Medium resolution 2 = High resolution Any negative number means keep current display resolution
physaddr	long	The new starting address for the physical screen. Any negative number means keep current physical screen.
logaddr	long	The new starting address for the logical screen. Any negative number means keep current logical screen.

Results

None

See also

Physbase( ), Logbase( ), Getrez( )

Set Color Palette

Setpalette( ) **Opcode = 6**

This function allows you to set an entire color palette of 16 hardware color registers at one time.

C macro format

```
int palette[16];
    Setpalette(palette);
```

Machine language format

```
move.l      palette, -(sp)
move.w      #6, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

palette	long	A pointer to an array that holds 16 words of color data, each of which contains the color settings for one of the color registers. Bits 0–3 are used for the blue component, bits 4–7 for green, and bits 8–11 for red.

Results

None

See also

Setcolor( )

Set Color Register

Setcolor( ) **Opcode = 7**

This function allows you to change the color in a single hardware color register.

C macro format

```
int oldcolor, register, newcolor;
    oldcolor = Setcolor(register, newcolor);
```

Machine language format

```
move.w      #newcolor, -(sp)
move.w      #register, -(sp)
move.w      #7, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

newcolor	word	A 16-bit word indicating the new color for the register. Bits 0–3 are used for the blue component, bits 4–7 for green, and bits 8–11 for red. A negative value indicates that you don't wish to change this register, but only to read its current contents.
register	word	The number of the hardware color register to change.

Results

D0	oldcolor	long	The value contained in the color register prior to the call.

See also

Setpalette( )

Read Floppy Disk Sector

Floprd( ) **Opcode = 8**

This function is used to read one or more sectors of information from a floppy disk.

C macro format

```
int status devnum, secnum
tracknum, sidenum, numsecs;
long buf, resvd;
   status = Floprd(buf, resvd, devnum, secnum,
            tracknum, sidenum, numsecs);
```

Machine language format

```
move.w      #numsecs, -(sp)
move.w      #sidenum, -(sp)
move.w      #track-
            num, -(sp)
move.w      #secnum , -(sp)
move.w      #devnum, -(sp)
clr.l       -(sp)
move.l      buf, -(sp)
move.w      #8, -(sp)
trap        #14
add.l       #20,sp
```

Inputs

numsecs	word	The number of contiguous sectors to read.
sidenum	word	The side of the disk to read from (0 or 1).
tracknum	word	The number of the disk track at which to begin reading.
secnum	word	The sector number at which to begin reading (sectors are usually numbered from 1 to 9).
devnum	word	The number of the drive to read (0 = drive A:, 1 = drive B:).
resvd	long	A longword whose value is ignored, but which must be present as a place holder.
buf	long	The address of a buffer where the data from one or more sequential sectors may be stored.

Results

D0	status	word	An error code for the function. A value of 0 means the operation was successful. Any negative value represents a system error.

See also

Flopwr( ), Flopver( )

Write Floppy Disk Sector

Flopwr( ) **Opcode = 9**

This function is used to write one or more sectors of information to a floppy disk.

C macro format

```
int status devnum, secnum,
tracknum, sidenum, numsecs;
long buf, resvd;
   status = Flopwr(buf, resvd, devnum, secnum,
            tracknum, sidenum, numsecs);
```

Machine language format

```
move.w      #numsecs, -(sp)
move.w      #sidenum, -(sp)
move.w      #track-
            num, -(sp)
move.w      #secnum , -(sp)
move.w      #devnum, -(sp)
clr.l       -(sp)
move.l      buf, -(sp)
move.w      #9, -(sp)
trap        #14
add.l       #20,sp
```

Inputs

numsecs	word	The number of contiguous sectors to write.
sidenum	word	The side of the disk to write to (0 or 1).
tracknum	word	The number of the disk track at which to begin writing.
secnum	word	The sector number at which to begin writing(sectors are usually numbered from 1 to 9).
devnum	word	The number of the drive to write to (0 = drive A:, 1 = drive B:).
resvd	long	A longword whose value is ignored, but which must be present as a place holder.
buf	long	The address of a buffer where the data for one or more sequential sectors is stored.

Results

D0	status	word	An error code for the function. A value of 0 means the operation was successful. Any negative value represents a system error.

See also

Floprd( ), Flopver( )

Format Floppy Disk Track

Flopfmt( ) **Opcode = 10**

Flopfmt( ) formats and verifies a single track of a floppy disk. To format and initialize a disk, you must create a boot sector and clear the first two tracks, in addition to formatting all the tracks.

C macro format

```
long buffer, skewtabl, magic;
int status, devnum, spt, tracknum,
sidenum, intrlev, magic, initial;
   status = Flopfmt(buffer, skewtabl, devnum, spt,
            tracknum, sidenum, intrlev, magic, initial)
```

Machine language format

```
move.w      #initial, -(sp)
move.l      #magic, -(sp)
move.w      #intrlev, -(sp)
move.w      #sidenum, -(sp)
move.w      #track-
            num, -(sp)
move.w      #spt, -(sp)
move.w      #devnum, -(sp)
move.l      skewtabl, -(sp)
move.l      buffer, -(sp)
move.w      #10, -(sp)
trap        #14
add.l       #26,sp
```

Inputs

initial	word	A 16-bit value to which all of the data bytes in a sector are initially set (usually $E5E5).
magic	long	Must be set to $87654321.
intrlev	word	Sector interleave flag 1 = normal, −1 = skewed.
sidenum	word	Side of the disk to format (0 or 1).
tracknum	word	Track number to be formatted (normally 0–79).
spt	word	Sectors per track. Normally set to 9.
devnum	word	The drive number (0 = drive A:, 1 = drive B:).

skewtabl	long	This parameter is ignored in the early (pre-blitter) version of TOS, but in the later versions of TOS, may be used to change the sector *interleave*. When *interlev* is set to -1, this parameter should point to an array that contains a 16-bit sector number for each sector, in the order in which sectors are to appear on successive tracks.
buffer	long	A pointer to a memory area used to hold the data to be written to the newly-formatted track. For a floppy using 9 sectors per track, an 8K buffer is recommended. The buffer must start on an even address.

Results

D0	status	word	Operation status code. Zero means no errors, and nonzero code represents a system error code. If the format fails due to nonverified sectors, a list of bad sectors is returned in the buffer.

See also

Protobt( ), Flopwr( )

Write String to MIDI Port

Midiws( ) **Opcode = 12**

This function sends a string of characters out the MIDI port.

C macro format

```
int bytes;
long buffer;
    Midiws(bytes, buffer);
```

Machine language format

```
move.l      buffer, -(sp)
move.w      #bytes, -(sp)
move.w      #12, -(sp)
trap        #14
addq.l      #8,sp
```

Inputs

buffer	long	The address of the memory buffer that contains the character string to write.
bytes	word	The length of the character string to write (minus one), in bytes.

Results

None

See also

Ikbdws( )

new page

Change MFP Interrupt Vector

Mfpint( ) **Opcode = 13**

This function is used to change an interrupt vector on the MFP chip.

C macro format

```
int number;
long vector;
   Mfpint(number, vector)
```

Machine language format

```
move.l      vector, -(sp)
move.l      #number, -(sp)
move.w      #13, -(sp)
trap        #14
addq.l      #8,sp
```

Inputs

vector	long	Pointer to the new interrupt handler routine to use.
number	word	Number of the MFP interrupt to change.

Results

None

See also

Jdisint( ), Jenabint( )

Get I/O Buffer Record

Iorec( ) **Opcode = 14**

Iorec( ) returns the address of a data structure known as the I/O buffer record, contains information about the input buffer used by a specified device.

C macro format

```
int dev;
long bufrec;
   bufrec = Iorec(dev);
```

Machine language format

```
move.w        #dev,-(sp)
move.w        #14,-(sp)
trap          #14
addq.l        #4,sp
```

Inputs

dev	word	The input device whose buffer record you wish to find: 0 = RS-232 serial port 1 = Console keyboard 2 = MIDI port

Results

D0	bufrec	long	The address of the device's buffer record.

This record contains 14 bytes of data, in the format.
The contents of the device's buffer record pointed to by *bufrec:*

Byte Number	*Element Name*	*Contents*
0–3	ibuf	Address of the input buffer
4–5	ibufsize	Size of the input buffer (in bytes)
6–7	ibufhd	Index to head (next write position)
8–9	ibuftl	Index to tail (next read position)
10–11	ibuflow	*Low water* mark
12–13	ibufhi	*High water* mark

For the RS-232 device only, an output buffer record follows immediately after the input buffer record.

Configure RS-232 Port

Rsconf( ) **Opcode = 15**

This function lets you change the RS-232 serial port parameters, such as communications speed, handshaking, parity, and so on.

C macro format

```
int speed, handshake, ucr, rsr, tsr, scr;
   Rsconf(speed, handshake, ucr, rsr, trs, scr);
```

Machine language format

```
move.w      #scr, -(sp)
move.w      #trs, -(sp)
move.w      #rsr, -(sp)
move.w      #ucr, -(sp)
move.w      #handshake, -(sp)
move.w      #speed, -(sp)
move.w      #15, -(sp)
trap        #14
add.l       #14,sp
```

Inputs

scr	word	Sets the MFP Synchronous Character Register (−1 = keep current contents)
tsr	word	Sets the MFP Transmit Status Register (−1 = keep current contents).
rsr	word	Sets the MFP Receive Status Register (−1 = keep current contents).
ucr	word	Sets the 8-bit MFP USART Control Register.

The 8-bit MFP USART Control Register:

Bit	*Function*
0	Not used
1	Parity type 0 = Odd 1 = Even
2	Parity enable 0 = Off 2 = On
3–4	Async start and stop bits

Bits 4	3	Number of start and stop bits
0	0	No start or stop bits (synchronous)
0	1	1 start bit, 1 stop bit
1	0	1 start bit, 1 1/2 stop bits
1	1	1 start bit, 2 stop bits

5–6 Data bits per word

Bits 6	5	Number of data bits
0	0	8 bits
0	1	7 bits
1	0	6 bits
1	1	5 bits

7 Clock

0 = Use clock directly for transfer frequency (synchronous transfer).

1 = Divide clock frequency by 16.

handshake	word	A flag that indicates the method of flow control of handshaking used.

The method of flow control of handshaking used:

Value	*Method*
0	No handshaking
1	XON/XOFF
2	RTS/CTS (not implemented in pre-blitter ROMs)

speed	word	The communications speed (baud rate).

Communication in bits per second for values in speed:

Speed Value	*Communication speed*
0	19200 bps
1	9600 bps
2	4800 bps
3	3600 bps
4	2400 bps
5	2000 bps
6	1800 bps
7	1200 bps
8	600 bps
9	300 bps
10	200 bps
11	150 bps
12	134 bps
13	110 bps
14	75 bps
15	50 bps

Results

None

Get/Set Keyboard Mapping Tables

Keytbl( ) **Opcode = 16**

This function allows you to find and change the tables that map keys to their ASCII values. This lets you change your keyboard layout to an alternate configuration, such as that used for Dvorak keyboards, or for foreign alphabets.

C macro format

```
char unshift[128], shift[128], capslock[128];
long vectable;
   vectable = Keytbl(unshift, shift, capslock);
```

Machine language format

```
move.l       capslock, -(sp)
move.l       shift, -(sp)
move.l       unshift, -(sp)
move.w       #16, -(sp)
trap         #14
add.l        #14,sp
```

Inputs

capslock	long	A pointer to your own 128-byte keyboard mapping table for Caps Lock characters. −1 = use current table
shift	long	A pointer to your own 128-byte keyboard mapping table for Shift characters. −1 = use current table
unshift	long	A pointer to your own 128-byte keyboard mapping table for unshifted characters. −1 = use current table

Results

D0	vectable	long	The address of a vector table that contains pointers to each of the three keyboard tables:

Byte Number	*Contents*
0–3	Address of unshifted table
4–7	Address of Shift table
8–11	Address of CapsLock table

See also

Bioskeys( )

Get Pseudo-Random Number

Random( ) **Opcode = 17**

This function returns the next 24-bit pseudo-random number in the series generated by the algorithm:

SEED = (SEED * 3141592621) + 1

The value returned is the new seed value, shifted eight bits to the right. Since the initial seed value is taken from the screen's vertical blank frame counter, the sequence should be different each time the machine is turned on.

C macro format

```
long rndnum;
   rndnum = Random( )
```

Machine language format

```
move.w        #17,-(sp)
trap          #14
addq.l        #2,sp
```

Inputs

None

Results

D0	rndnum	long	A 24-bit pseudo-random number (bits 24–31 are 0)

Produce Boot Sector Prototype

Protobt( ) **Opcode = 18**

This function is used to create a boot sector in memory. This is a specially formatted block of information that must be stored on the first sector of each floppy disk (side 0, track 0, sector 1). The finished boot sector should be written to the floppy disk, using the Flopwr( ) function, not the BIOS function Rwabs( ).

C macro format

```
int disktype, execflag;
long buffer, serialnum;
   Protobt(buffer, serialnum, disktype, execflag);
```

Machine language format

```
move.w      execflag, -(sp)
move.w      disktype, -(sp)
move.l      serialnum, -(sp)
move.l      buffer, -(sp)
move.w      #18, -(sp)
trap        #14
add.l       #14,sp
```

Inputs

execflag	word	A flag that indicates whether to execute some boot code at startup time. Boot code is up to 480 bytes of machine instruction, starting at byte 30 of the boot sector. 0 = no boot code 1 = execute boot code
disktype	word	A code word that specifies a disk's storage capacity and format: 0 = 40 tracks, single sided (180K) 1 = 40 tracks, double sided (360K) 2 = 80 tracks, single sided (360K) 3 = 80 tracks, double sided (720K)
serialnum	long	A unique 24-bit identifier code used by the file system to tell whether disks have been changed in a particular drive. To generate a random serial number, pass a value larger than ($1000000).
buffer	long	The address of a 512-byte memory buffer where the boot block information will be created.

Results

None

See also

Flopfmt( ), Flopwr( )

Verify Floppy Disk Sector

Flopver( ) **Opcode = 19**

This function is used to verify one or more sectors of information on a floppy disk.

C macro format

```
int status devnum, secnum, tracknum, sidenum, numsecs;
long buf, resvd;
   status = Flopver(buf, resvd, devnum, secnum, tracknum, sidenum,
            numsecs);
```

Machine language format

```
move.w     #numsecs, -(sp)
move.w     #sidenum, -(sp)
move.w     #tracknum, -(sp)
move.w     #secnum , -(sp)
move.w     #devnum, -(sp)
clr.l      -(sp)
move.l     buf, -(sp)
move.w     #19, -(sp)
trap       #14
add.l      #16,sp
```

Inputs

numsecs	word	The number of contiguous sectors to verify.
sidenum	word	The side of the disk to verify (0 or 1).
tracknum	word	The number of the disk track at which to begin verifying.
secnum	word	The sector number at which to begin verifying (sectors are usually numbered from 1 to 9).
devnum	word	The number of the drive to verify (0 = drive A:, 1 = drive B:).
resvd	long	A longword whose value is ignored, but which must be present as a place holder.
buf	long	The address of a buffer where the data from one or more sequential sectors is stored.

Results

D0	status	word	An error code for the function. A value of 0 means that the operation was successful. Any negative value represents a system error.

See also

Floprd( ), Flopver( )

Output Graphics Screen to Printer

Scrdmp( ) **Opcode = 20**

The Scrdmp( ) function prints a graphic representation of the screen display on an Atari or Epson-compatible printer.

C macro format

```
Scrdmp( );
```

Machine language format

```
move.w     #20,-(sp)
trap       #14
addq.l     #2,sp
```

Inputs

None

Results

None

See also

Prtblk( )

Configure Text Cursor

Cursconf( ) **Opcode = 21**

The Cursconf( ) function allows you to control the visibility and blink rate of the system's text cursor.

C macro format

```
int rate, mode, newrate;
   rate = Cursconf(mode,newrate);
```

Machine language format

```
move.w     #newrate,-(sp)
move.w     #mode,-(sp)
move.w     #21,-(sp)
trap       #14
addq.l     #6,sp
```

Inputs

newrate	word	If mode = 5, this value (0–255) determines the rate at which the cursor blinks, according to this formula: Duration of a single blink (in seconds) = 2 * rate / cycles. In this formula, cycles represents the monitor frequency (monochrome = 70, US color = 60, European color = 50).
mode	word	A flag that indicates the change in cursor function: 0 = Turn cursor off 1 = Turn cursor on 2 = Turn cursor blink on 3 = Turn cursor blink off 4 = Change blink rate to value contained in newrate 5 = Read cursor blink rate

Results

D0	rate	word	When mode is set to 5, the current blink rate value (0-255) is returned here.

Set System Time and Date

Settime( ) **Opcode = 22**

The Settime( ) function is used to set the intelligent keyboard's time and date clock.

C macro format

```
long datetime;
   Settime(datetime);
```

Machine language format

```
move.l     #datetime, -(sp)
move.w     #22, -(sp)
trap       #14
addq.l     #6,sp
```

Inputs

datetime	long	A 32-bit word indicating the time and date to which the clock should be set.

Values contained in datetime:

Bit Number	*Description*	*Range*
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23
16–20	Day	1–31
21–24	Month	1–12
25–31	Year – 1980	0–119)

Results

None

See also

Gettime( )

Get System Time and Date

Gettime( ) **Opcode = 23**

This function allows you to read the intelligent keyboards time and date clock.

C macro format

```
long datetime;
   datetime = Gettime( );
```

Machine language format

```
move.w      #23,-(sp)
trap        #14
addq.l      #2,sp
```

Inputs

None

Results

D0	datetime	long	A 32-bit value that indicates the current time and date settings.

Values for datetime:

Bit Number	*Description*	*Range*
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23
16–20	Day	1–31
21–24	Month	1–12
25–31	Year – 1980	0–119)

See also

Settime( )

Restore Default Keyboard Table

Bioskeys( ) **Opcode = 24**

This function replaces the keyboard mapping table you have installed using the Keytbl( ) function with the default system keymaps.

C macro format

Bioskeys( )

Machine language format

```
move.w          #24,-(sp)
trap            #14
addq.l          #2,sp
```

Inputs

None

Results

None

See also

Keytbl( )

Write String to Intelligent Keyboard

Ikbdws( ) **Opcode = 25**

This function sends a string of characters out to the Intelligent Keyboard controller.

C macro format

```
int bytes;
long buffer;
   Ikbdws(bytes, buffer);
```

Machine language format

```
move.l      buffer, -(sp)
move.w      #bytes, -(sp)
move.w      #25, -(sp)
trap        #14
addq.l      #8,sp
```

Inputs

buffer	long	The address of the memory buffer that contains the character string to write.
bytes	word	The length of the character string to write (in bytes) − 1.

Results

None

See also

Midiws( )

Disable an MFP Interrupt

Jdisint( ) **Opcode = 26**

Disables one of the 16 MFP interrupts.

C macro format

```
int intnum;
    Jdisint(intnum);
```

Machine language format

```
move.w      #intnum, -(sp)
move.w      #26, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

intnum	word	Which of the 16 MFP interrupts (0–15) to disable.

Results

None

See also

Jenabint( ), Xbtimer( ), Mfpint( )

Enable an MFP Interrupt

Jenabint( ) Opcode = 27

Enables one of the 16 MFP interrupts.

C macro format

```
int intnum;
   Jenabint(intnum);
```

Machine language format

```
move.w      #intnum, -(sp)
move.w      #27, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

intnum	word	Which of the 16 MFP interrupts (0–15) to enable.

Results

None

See also

Jdisint( ), Xbtimer( ), Mfpint( )

Read/Write Sound Chip

Giaccess( ) **Opcode = 28**

This function allows you to read or change any register in the Programmable Sound Generator (PSG) chip.

C macro format

```
char regvalue, value;
int regnum;
   regvalue = Giaccess(value, regnum);
```

Machine language format

```
move.w      #regnum, -(sp)
move.w      #value, -(sp)
move.w      #28, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

regnum	word	The number of the register to read or change. If you're reading the register, use the register number (0–15). To change the register, use the register number + 128 (128–143).
value	byte	The new 8-bit number to go into the register.

Results

D0	regvalue	byte	The number stored in the register at the end of the call.

See also

Ongibit( ), Offgibit( )

Clear a Bit on Sound Chip I/O Port

Offgibit( ) **Opcode = 29**

Atomically clears a single bit of the Port A I/O register on the PSG sound chip.

C macro format

```
int bitnum;
    Offgibit(bitnum);
```

Machine language format

```
move.w      #bitnum, -(sp)
move.w      #29, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

bitnum	word	The number of the bit (0–7) to change.

Results

None

See also

Ongibit( )

Set a Bit on Sound Chip I/O Port

Ongibit( ) **Opcode = 30**

Atomically sets a single bit of the Port A I/O register on the PSG sound chip.

C macro format

```
int bitnum;
    Ongibit(bitnum);
```

Machine language format

```
move.w      #bitnum, -(sp)
move.w      #30, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

bitnum	word	The number of the bit (0–7) to change.

Results

None

See also

Offgibit( )

Set an MFP Timer

Xbtimer( ) **Opcode = 31**

The Xbtimer( ) function allows you to set the MFP timer registers and assign an interrupt vector to a timer.

C macro format

```
int timernum, control, data;
long vector;
   Xbtimer(timernum, control, data, vector);
```

Machine language format

```
move.l      vector, -(sp)
move.w      data, -(sp)
move.w      control, -(sp)
move.w      timernum, -(sp)
move.w      #31, -(sp)
trap        #14
add.l       #12,sp
```

Inputs

vector	long	The address of the interrupt handler routine associated with this timer.
data	word	The value store in the timer data register.
control	word	The 8-bit value to place in the timer's control register.

For Timers A and B, the values are:

Control Value	*Timer Mode*
0	Timer off
1	Delay mode, clock divided by 4
2	Delay mode, clock divided by 10
3	Delay mode, clock divided by 16
4	Delay mode, clock divided by 50
5	Delay mode, clock divided by 64
6	Delay mode, clock divided by 100
7	Delay mode, clock divided by 200
8	Event Count Mode
9	Pulse Length mode, clock divided by 4
10	Pulse Length mode, clock divided by 10
11	Pulse Length mode, clock divided by 16
12	Pulse Length mode, clock divided by 50
13	Pulse Length mode, clock divided by 64
14	Pulse Length mode, clock divided by 100
15	Pulse Length mode, clock divided by 200

timernum	word	A number from 0–3 that represents the MFP timer to change. (0 = timer A, 1 = timer B, 2 = timer C, 3 = timer D).

Results
None

See also
Mfpint( ), Jdisint( ), Jenabint( )

Start Sound Interrupt Processing

Dosound( ) **Opcode = 32**

The Dosound( ) function enables an interrupt-driven routine that can play music or sound effects in the background.

C macro format

```
long commandlist;
   Dosound(commandlist);
```

Machine language format

```
move.l      commandlist, -(sp)
move.w      #32, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

commandlist	long	A pointer to a data storage area that contains a number of queued sound commands, one of which is executed each 1/50 of a second.

The command structure is:

(Command Number) *First Byte*	*Second Byte*	*Third Byte*	*Fourth Byte*
0–15 Load a byte value into the register specified by this command byte. (0–15)	The byte value to load into the register (0–255).		
128 Store a value in a temporary register for use by command number 129 (see below).	The byte value to store in the temporary register (0–255). This is the starting register value for command 129.		
129 Load a register with the value stored in the temporary register (by command 128). Increment the value in the register each timer tick, until an end value is reached.	The number of the register to use. (0–15)	Increment value (0–255). Added to register value each timer tick. Can be positive (0–127) or negative.	Final register value. Command ends when this value is reached.

First Byte	*(Command Number)* *Second Byte*	*Third Byte*	*Fourth Byte*
130–255 Pause for a specified number of timer ticks (1/50 second) before the next sound command is executed.	The number of timer ticks to pause (1–255). A value of 0 here will end the processing of sound commands.		

Results

None

See also

Giaccess( )

Set Printer Configuration

Setprt( ) **Opcode = 33**

This function allows you to set the printer configuration, a code number that contains information about the type of printer that is attached. This information is used by some system routines, like the screen printing function.

C macro format

```
int code, newcode;
   code = Setprt(newcode);
```

Machine language format

```
move.w      newcode, -(sp)
move.w      #33, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

newcode	word	The 16-bit printer configuration flag you want to set.

The meaning of each flag bit is shown in this table:

Bit Number	*Description*
0	Print type 0 = Dot-Matrix 1 = Daisywheel
1	Color type 0 = Monochrome 1 = Color print
2	Control code type 0 = Atari 1 = Epson
3	Print quality 0 = Draft 1 = Final quality
4	Printer port 0 = Parallel 1 = RS232 serial
5	Paper type 0 = Continuous 1 = Single Sheet
6	Reserved for future use
7	Reserved for future use
8	Reserved for future use
9	Reserved for future use
10	Reserved for future use
11	Reserved for future use
12	Reserved for future use

13	Reserved for future use
14	Reserved for future use
15	Must be 0

Results

D0	code	word	The old value of the printer configuration code. By setting newcode to -1, it's possible to read the current code value without changing it.

Get Keyboard Vector Table Base Address

Kbdvbase( ) **Opcode = 34**

This function returns pointers to several of the interrupt routines that are used to handle the input functions.

C macro format

```
long vecbase;
   vecbase = Kbdvbase( );
```

Machine language format

```
move.w        #34,-(sp)
trap          #14
addq.l        #2,sp
```

Inputs

None

Results

D0	vecbase	long	A pointer to a vector table.

Vector table structure:

Byte Offset	*Vector Name*	*Routine called*
0	midivec	MIDI input routine
4	vkbderr	IKBD ACIA over-run error routine
8	vmiderr	MIDI ACIA over-run error routine
12	statvec	IKBD status packet handler
16	mousevec	IKBD mouse packet handler
20	clockvec	IKBD clock packet handler
24	joyvec	IKBD joystick packet handler
28	midisys	System MIDI ACIA handler
32	ikbdsys	System IKBD ACIA handler

Set Keyboard Repeat Rate

Kbrate( ) **Opcode = 35**

The Kbrate( ) function is used to control the repeat rate of the console device keyboard.

C macro format

```
int oldvals, delay, rate;
   oldvals = Kbrate(delay, rate);
```

Machine language format

```
move.w      rate, -(sp)
move.w      delay, -(sp)
move.w      #35, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

rate	word	The amount of time that elapses between each repetition of a key. This time is measured in system clock ticks (1/50 of a second). A value of 0–255 is used. Zero represents the maximum delay of 256 ticks.
delay	word	The amount of time you must initially hold down a key before it starts to repeat.

Results

D0	oldvals	word	The previous rate and delay values are packed into a single 16-bit word. Bits 0–7 contain the rate value Bits 8–15 the delay value

Output Graphics Block to Printer

Prtblk( ) **Opcode = 36**

This function can be used to print the entire screen, or any portion of it, to a graphics printer.

C macro format

```
long prtable;
   Prtblk(prtable);
```

Machine language format

```
move.l     prtable, -(sp)
move.w     #36, -(sp)
trap       #14
addq.l     #6,sp
```

Inputs

prtable	long	The address of a 30-byte parameter table that determines how the screen block is printed.

Byte Number	*Element Name*	*Description*
0–3	long blkprt ($44E)	Starting address of screen RAM
4–5	word offset (0)	Offset from start address in bits (0–7)
6–7	word width (640)	Screen width (in bytes)
8–9	word height (400)	Screen height
10–11	word left (0)	Left margin for screen dump
12–13	word right (0)	Right margin for screen dump
14–15	word scrres (2)	Screen resolution
		0 = Low
		1 = Medium
		2 = High
16–17	word dstres (0)	Printer resolution
		0 = Draft: 960 dpi (dots per inch)
		1 = Final: 1280 dpi
18–21	long colpal ($FF8240)	Starting address of the color palette
22–23	word type (4)	Printer type
		0 = Atari monochrome dot-matrix
		1 = Atari monochrome daisywheel
		2 = Atari color dot-matrix
		4 = Epson monochrome dot-matrix
24–25	word port (0)	Printer port
		0 = Parallel
		1 = RS232 serial
26–29	word masks (0)	Starting address of halftone mask table (if 0, use default ROM table)

Results

None

See also

Scrdmp( )

Note Scrdmp() uses prtable = $992
Values in brackets above denote values used for
hi-res dump to Epson B/W dot-matrix

Wait for Vertical Blank

Vsync( ) **Opcode = 37**

This function simply waits until the vertical blank interrupt is finished, and returns. It can be used to synchronize screen drawing with the vblank, or as a simple delay mechanism.

C macro format

```
Vsync( );
```

Machine language format

```
move.w      #37, -(sp)
trap        #14
addq.l      #2,sp
```

Inputs

None

Results

None

Execute Supervisor Mode Function

Supexec( ) **Opcode = 38**

Runs a subroutine in the 68000 processor's supervisor mode.

C macro format

```
long sub;
   Supexec(sub);
```

Machine language format

```
move.l      sub, -(sp)
move.w      #38, -(sp)
trap        #14
addq.l      #6,sp
```

Inputs

sub	long	The address of the subroutine to run in supervisor mode.

Results

None

See also

Get/Set Blitter Configuration

Blitmode( ) **Opcode = 64**

This function is used to find out if a blitter chip is available, and whether it is being used for drawing routines. If the blitter chip is present, this function may also be used to choose between hardware blitting or software emulation mode.

C macro format

```
#define Blitmode(a) xbios(64,a)
int status, value;
   status = Blitmode(value);
```

Machine language format

```
move.w      value, -(sp)
move.w      #64, -(sp)
trap        #14
addq.l      #4,sp
```

Inputs

value	word	A bit flag used to set the blitter configuration. A value of −1 is used to read, not change, the configuration.

The bit values for the value flag are:

Bit Number	*Function*
0	Set blit mode 0 = Use software blit routines 1 = Use blitter hardware
1–14	Undefined, reserved
15	Must be 0

Results

D0	status	word	A bit flag that returns the blitter configuration as it stood prior to the set operation.

The status bit values are:

Bit Number	*Description*
0	Current blit mode 0 = Using software blit routines 1 = Using blitter hardware
1	Blitter chip availability 0 = Blitter chip not available 1 = Blitter chip is installed
2–14	Undefined, reserved
15	A 0 is always returned

Appendix C

GEMDOS Functions

GEMDOS routines can be called from user mode. They use registers A0–A2 and D0–D2 as scratch registers. If you are programming in machine language and your program uses these registers, you must save their contents before making a GEMDOS call and restore them after the call terminates. Each of the GEMDOS routines is associated with a command number, and some of the routines require command parameters that specify more precisely what they should do. For example, the GEMDOS function to write a character to the console screen has a command number of 2. It requires a single command parameter that tells the function which character to print.

To call a GEMDOS function from machine language, you must push the command parameters onto the stack, followed by the command number, and execute a TRAP #1 statement. The TRAP #1 instruction puts the program into supervisor mode, and begins executing the instructions found at the address stored in exception vector 33, whose address is 132 ($84). This exception vector contains the address of the GEMDOS handler, which reads the command number on the top of the stack, and directs program execution to the appropriate function. When the function terminates, the program returns to user mode, and the results, if any, are returned in register D0. In most cases, the value is returned as a longword, but there are exceptions. Some error codes are returned as words, so it's best to test only the low-order words when checking for errors. You should also be aware that sometimes a GEMDOS function will return a BIOS error number (between −1 and −31). When a GEMDOS function call is completed, the calling program has the responsibility to adjust the stack in order to remove the command parameters and command number.

The following program fragment demonstrates how you would print the character *A* on the console screen using GEMDOS command 2:

```
move.w   #'A',-(sp)  * push the character value on stack
move.w   #2,-(sp)    * push GEMDOS command number on
                     * stack
trap     #1          * call GEMDOS handler
addq.l   #4,sp       * pop parameters (4 bytes) off stack
```

Calling the GEMDOS routines from C is much simpler. Most C compilers come with a library routine called gemdos( ), which stacks the parameters and executes the TRAP #1 instruction. For example, the sample call illustrated above could be accomplished in C by the single statement:

```
gemdos(2, 'A');
```

Since it's easier to remember a command name than a command number, most C compilers include a header file called OSBIND.H that defines macros for all of the GEMDOS functions. For example, the macro definition for GEMDOS command 2 is

```
#define Cconout(a)   gemdos(0x2,a)
```

Therefore, after you #include OSBIND.H in your program, you can call the sample function like this:

```
Cconout('A');
```

Since this format is the most readable, most reference books and sample programs use the C macro notation. Remember, however, that in order to use GEMDOS functions in your C programs, you must link your program with the compiler library that contains the gemdos( ) function, and you must #include OSBIND.H if you use the macros.

Terminate Process

Pterm0( ) **Opcode = 0 ($00)**

This function terminates the current process, closes all open files, clears the memory space used by the process, and returns to the program that called it (normally the GEM Desktop). This function is mainly of interest to machine language programmers, since the startup modules supplied with C language compilers automatically call one of the terminate functions when the main( ) function exits.

C macro format

Pterm0( );

Machine language format

```
clr.w       -(sp)
trap        #1
```

Inputs

None

Results

None

See also

Pterm( ), Ptermres( )

Wait for Keyboard Character

Cconin( ) **Opcode = 1 ($01)**

This function waits until a character is available from the console keyboard, echoes it to the screen, and returns its ASCII value and scan code.

C macro format

```
long keycode;
   keycode = Cconin( );
```

Machine language format

```
move.w      #1,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	keycode	long	A longword that contains both the ASCII value of the key(s) pressed, and the scan code. The ASCII value is returned in the low byte of the low word of *keycode*, while the scan code is returned in the low byte of the high word. See Appendix J for a complete list of keycodes.

See also

Cconis( ), Crawcin( ), Cnecin( ), Crawcio( )

Send Character to Screen

Cconout( ) **Opcode = 2 ($02)**

This function sends a single character to the console screen device.

C macro format

```
char ch;
   Cconout(ch);
```

Machine language format

```
move.w      #char, -(sp)
move.w      #2, -(sp)
trap        #1
addq.l      #4,sp
```

Inputs

char	word	The low byte contains the ASCII value of the character to write to the screen, the high byte is 0.

Results

None

See also

Cconos( ), Cprnout( ), Cauxout( )

Wait for RS-232 Character

Cauxin( ) **Opcode = 3 ($03)**

This function waits until a character is available from the RS-232 serial device, and returns its ASCII value. It does not echo the character to the screen.

C macro format

```
char ch;
   ch = Cauxin( );
```

Machine language format

```
move.w          #3,-(sp)
trap            #1
addq.l          #2,sp
```

Inputs

None

Results

D0	ch	word	The lower eight bits contain the ASCII character received.

See also

Cauxis( ), Cconin( )

Send Character to RS-232 Port

Cauxout( ) **Opcode = 4 ($04)**

This function sends a single character to the RS-232 serial device. It does not return until the character has been sent.

C macro format

```
char ch;
   Cauxout(ch);
```

Machine language format

```
move.w      #ch, -(sp)
move.w      #4, -(sp)
trap        #1
addq.l      #4,sp
```

Inputs

ch	word	The low byte contains the ASCII value of the character to write to the serial port, the high byte is 0.

Results

None

See also

Cauxos( ), Cprnout( ), Cconout( )

Send Character to Printer

Cprnout( ) **Opcode = 5 ($05)**

This function sends a single character to the Centronics parallel printer device. It does not return until the character has been sent, or the printer times out.

C macro format

```
char ch;
int status;
   status = Cprnout(ch);
```

Machine language format

```
move.w      #ch, -(sp)
move.w      #5, -(sp)
trap        #1
addq.l      #4,sp
```

Inputs

ch	word	The low byte contains the ASCII value of the character to write to the printer port, the high byte is 0.

Results

status	word	A value of −1 is returned if the character has been sent correctly, a 0 if the character can't be sent within the *time out* period (time out period refers to the condition when the paper runs out, the printer is offline, and so on).

See also

Cprnos( ), Cprnout( ), Cconout( )

new page

Input/Output Console Character

Crawio( ) **Opcode = 6 ($06)**

This function allows you either to send characters to the console device, or to receive them. The receive character portion of the function does not wait for a character to become available, but returns immediately whether or not one was received.

C macro format

```
int chin, chout;
   chin = Crawio(chout);
```

Machine language format

```
move.w      chout,-(sp)
move.w      #6,-(sp)
trap        #1
addq.l      #4,sp
```

Inputs

chout	word	The low byte of this word either contains an ASCII character 0–254 to send to the screen, or a value of 255, which indicates that a character is to be received.

Results

D0	chin	word	If *chout* was set to 255, the value of the character read from the keyboard is returned here. If no character was available, a 0 is returned in both bytes. If a character can be read, the low byte contains its ASCII value, and the high byte contains its scan code.

Raw Keyboard Input Without Echo

Crawcin( ) **Opcode = 7 ($07)**

This function waits until a character is available from the console keyboard, and returns its ASCII value and scan code. Unlike Cconin( ), it does not echo the character to the screen.

C macro format

```
long keycode;
   keycode = Crawcin( );
```

Machine language format

```
move.w      #7,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	keycode	long	A longword that contains both the ASCII value of the key(s) pressed, and the scan code. The ASCII value is returned in the low byte of the low word of keycode, while the scan code is returned in the low byte of the high word. See Appendix J for a complete list of keycodes.

See also

Cconin( ), Cnecin( ), Crawcio( )

Keyboard Input Without Echo

Cnecin( ) **Opcode = 8 ($08)**

This function waits until a character is available from the console keyboard, does not echo the character to the screen, and returns its ASCII value and scan code. This function is identical to Crawcin( ), though the documentation states that it differs from that function in that Cnecin( ) acts on control codes like Control-S, Control-Q, and Control-C, instead of passing these codes on, like Crawcin( ). At least in current TOS versions, however, Cnecin( ) passes these codes on also.

C macro format

```
long keycode;
   keycode = Cnecin( );
```

Machine language format

```
move.w        #8,-(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None

Results

D0	keycode	long	A longword that contains both the ASCII value of the key(s) pressed, and the scan code. The ASCII value is returned in the low byte of the low word of *keycode*, while the scan code is returned in the low byte of the high word. See Appendix J for a complete list of keycodes.

See also

Cconin( ), Crawcin( ), Crawcio( )

Write String to Screen

Cconws( ) **Opcode = 9 ($09)**

Allows you to print an entire string of characters to the console device screen at once.

C macro format

```
int length;
char *string;
   length = Cconws(string);
```

Machine language format

```
move.l      string, -(sp)
move.w      #9, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

string	long	A pointer to a null-terminated string of text characters to be printed. Control character and escape sequences are interpreted as usual.

Results

D0	length	word	The number of characters that were printed.

See also

Cconout( )

Read String from Keyboard

Cconrs( ) **Opcode = 10 ($0A)**

This function reads an entire string of characters from the console keyboard, echoing each character to the screen as it is read. It provides some line-editing functions as well. The function will not return until the user signals that the entire string has been entered, by pressing the Return key, or by entering a string of the maximum length. Note that the entire program (not just this function) terminates if Control-C is struck during character entry.

C macro format

```
char buffer[MAXLEN];
int length;
   length = Cconrs(buffer);
```

Machine language format

```
move.l      buffer,-(sp)
move.w      #$0A,-(sp)
trap        #1
addq.l      #6,sp
```

Inputs

buffer	long	The address of a buffer into which the characters will be read. The first two bytes of this buffer are reserved. You should place the maximum number of characters that can be read (buffer length − 2) in the first byte. The function will terminate as soon as that many characters have been read. The length of the string that is read will be returned in the second byte of the buffer.

Results

D0	length	word	The length of the string read by the function.

Get Keyboard Input Status

Cconis( ) **Opcode = 11 ($0B)**

Allows you to determine whether there is a character waiting to be received from the console device keyboard.

C macro format

```
int status;
   status = Cconis( );
```

Machine language format

```
move.w      #$0B,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	status	long	A value of 0 means there are no characters waiting, while −1 means that at least one character is waiting to be received.

See also

Cconin( )

Set Default Drive Number

Dsetdrv( ) **Opcode = 14 ($0E)**

The default drive is the one GEMDOS assumes is referred to when only a filename is used. This function allows you to set the default drive, and returns information about the number of logical drives recognized by the system.

C macro format

```
long drives;
int default;
   drives = Dsetdrv(default);
```

Machine language format

```
move.w      #default,-(sp)
move.w      #$0E,-(sp)
trap        #1
addq.l      #4,sp
```

Inputs

default	word	The number of the drive to set as the default (drive A: = 0, drive B: = 1, and so on).

Results

D0	drives	long	A bit flag that indicates which drives are recognized by the system. Each bit that corresponds to a known drive is set to 1 (bit 0 = drive A:, bit 1 = drive B:, and so forth).

See also

Dgetdrv( ), Dsetpath( )

Get Screen Output Status

Cconos( ) **Opcode = 16 ($10)**

This function allows you to determine whether the console device screen is ready to accept a character. It is probably included for purposes of symmetry, since the screen is always ready to print a character.

C macro format

```
int status;
   status = Cconos( );
```

Machine language format

```
move.w        #$10, -(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None

Results

D0	status	long	A 0 means the device is not ready to accept a character, while a −1 means it is ready.

See also

Cconout( )

Get Printer Output Status

Cprnos( ) **Opcode = 17 ($11)**

This function allows you to determine whether the Centronics parallel printer device is ready to accept a character.

C macro format

```
int status;
   status = Cprnos( );
```

Machine language format

```
move.w        #$11, -(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None

Results

D0	status	long	A 0 means the device is not ready to accept a character, while a −1 means it is ready.

See also

Cprnout( )

Get RS-232 Input Status

Cauxis( ) **Opcode = 18 ($12)**

This function allows you to determine whether there is a character waiting to be received from the RS-232 serial device.

C macro format

```
int status;
   status = Cauxis( );
```

Machine language format

```
move.w      #$12,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	status	long	A value of 0 means there are no characters waiting, while −1 means at least one character is waiting to be received.

See also

Cauxin( )

Get RS-232 Output Status

Cauxos( ) **Opcode = 19 ($13)**

This function allows you to determine whether the RS-232 serial device is ready to accept a character.

C macro format

```
int status;
   status = Cauxos( );
```

Machine language format

```
move.w      #$13,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	status	long	A 0 means the device is not ready to accept a character, while a −1 means it is ready.

See also

Cauxout( )

Get Default Drive Number

Dgetdrv( ) **Opcode = 25 ($19)**

This function allows you to find the drive number of the current default drive.

C macro format

```
int default;
   default = Dgetdrv( );
```

Machine language format

```
move.w        #$19,-(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None

Results

D0	default	word	The drive number of the current default drive (drive A: = 0, drive B: = 1, and so on).

See also

Dsetdrv( )

Set Disk Transfer Address

Fsetdta( ) **Opcode = 26 ($1A)**

This function allows you to change the Disk Transfer Address, which points to the buffer used for disk read operations. This buffer is used as a scratch area for directory searches.

C macro format

```
char dta[44];
   Fsetdta(dta);
```

Machine language format

```
move.l      dta,-(sp)
move.w      #$1A,-(sp)
trap        #1
addq.l      #6,sp
```

Inputs

dta	long	A pointer to the DTA buffer that will be used from now on.

Results

None

See also

Fgetdta( ), Fsfirst( ), Fsnext( )

Set User/Supervisor Mode

Super( ) **Opcode = 32 ($20)**

This function is used to change between the 68000 processor's user and supervisor modes. Though most programs operate in user mode, certain operations on the ST can only be performed from supervisor mode. These include reading or writing to system variables stored in memory locations below 2048 ($800), and reading or writing to hardware registers located above 167444482 ($FF80000).

C macro format

```
long stack, oldstack;
  oldstack = Super(stack);
```

Machine language format

```
move.l      stack, -(sp)
move.w      #$20, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

stack	long	The address of the stack area to use after the mode change. 0 = Use same stack for supervisor and user modes. −1L = read (don't change) current privilege mode.

Results

D0	oldstack	long	If the mode was toggled, the address of the old supervisor stack is returned here. If stack was set to −1L, the value here is a flag indicating the current privilege mode: 0 = User 1 = Supervisor

Get GEMDOS Date

Tgetdate( ) **Opcode = 42 ($2A)**

Gets current date according to GEMDOS.

C macro format

```
int date;
   date = Tgetdate( );
```

Machine language format

```
move.w        #$2A,-(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None

Results

D0	date	word	A 16-bit code that specifies the GEMDOS date.

Interpretation of GEMDOS *date:*

Bit Number	*Description*	*Range*
0–4	Day	1–31
5–8	Month	1–12
9–15	Year – 1980	0–119

See also

Tsetdate( ), Tgettime( ), Tsettime( )

Set GEMDOS Date

Tsetdate( ) **Opcode = 43 ($2B)**

Sets current date for GEMDOS.

C macro format

```
int date;
   Tsetdate(date);
```

Machine language format

```
move.w      #date, -(sp)
move.w      #$2B, -(sp)
trap        #1
addq.l      #4,sp
```

Inputs

date	word	A 16-bit code that specifies the GEMDOS date.

Interpretation of GEMDOS *date:*

Bit Number	*Description*	*Range*
0–4	Day	1–31
5–8	Month	1–12
9–15	Year − 1980	0–119

Results

None

See also

Tgetdate( ), Tgettime( ), Tsettime( )

Get GEMDOS Time

Tgettime( ) **Opcode = 44 ($2C)**

This function is used to get the GEMDOS version of the current time.

C macro format

```
int time;
   time = Tgettime( );
```

Machine language format

```
move.w      #$2C, -(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	time	word	A 16-bit code that indicates the time.

Interpretation of GEMDOS *time:*

Bit Number	*Description*	*Range*
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hours	0–23

See also

Tsettime( ), Tgetdate( ), Tsetdate( )

Set GEMDOS Time

Tsettime( ) **Opcode = 45 ($2D)**

Allows you to set the GEMDOS time.

C macro format

```
int time;
   Tsettime(time);
```

Machine language format

```
move.w      #time, -(sp)
move.w      #$2D, -(sp)
trap        #1
addq.l      #4,sp
```

Inputs

time	word	A 16-bit code that indicates the time.

Interpretation of GEMDOS *time:*

Bit Number	*Description*	*Range*
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hours	0–23

Results

None

See also

Tsettime( ), Tgetdate( ), Tsetdate( )

Get Disk Transfer Address

Fgetdta( ) **Opcode = 47 ($2F)**

This function finds the current Disk Transfer Address that points to the buffer used for disk read operations. This buffer is used as a scratch area for directory searches.

C macro format

```
char *dta;
   dta = Fgetdta( );
```

Machine language format

```
move.w       #$2F,-(sp)
trap         #1
addq.l       #2,sp
```

Inputs

None

Results

D0	dta	long	A pointer to the current DTA.

See also

Fsetdta( ), Fsfirst( ), Fsnext( )

Get GEMDOS Version Number

Sversion( ) **Opcode = 48 ($30)**

This function returns the GEMDOS version number. This number refers only to the version of GEMDOS, not to the GEM or TOS version in general.

C macro format

```
int version;
   version = Sversion( );
```

Machine language format

```
move.w      #$30,-(sp)
trap        #1
addq.l      #2,sp
```

Inputs

None

Results

D0	version	word	A 16-bit code that indicates the GEMDOS version number. The major version number is stored in the low byte, and the minor revision number in the high byte.

Terminate and Stay Resident

Ptermres( ) **Opcode = 49 ($31)**

Like the Pterm( ) and Pterm0( ) functions, this one terminates the current process, closes all open files, and exits to the calling program, usually the GEM Desktop. It allows a return code to be passed to the calling program as well. Unlike the others, however, Ptermres( ) doesn't automatically clear the memory space used by the process. All or part of the program may remain loaded in memory even after the process terminates. TSR programs can "steal" the vectors used by exception handlers, such as the system timer, vertical blank interrupt, or the keyboard Alternate-HELP screen dump routine, to create hot-key applications. This function is mainly of interest to machine language programmers, since the startup modules supplied with C language compilers automatically call one of the terminate functions when the main( ) function exits.

C macro format

```
long keepsize;
int retcode;
   Ptermres(keepsize, retcode);
```

Machine language format

```
move.w      #retcode, -(sp)
move.l      #keepsize, -(sp)
move.w      #$31, -(sp)
trap        #1
```

Inputs

retcode	word	The exit code that is returned to the calling program.
keepsize	long	The number of bytes of memory to keep resident, starting at and including the 256-byte basepage.

Results

None

See also

Pterm0( ), Pterm( )

Get Disk Free Space

Dfree( ) **Opcode = 54 ($36)**

The Dfree( ) function is used to find the amount of free space left on a disk.

C macro format

```
long buffer[4];
int drivenum;
   Dfree(buffer, drivenum);
```

Machine language format

```
move.w      #drivenum,-(sp)
move.l      buffer,-(sp)
move.w      #$36,-(sp)
trap        #1
addq.l      #8,sp
```

Inputs

drivenum	word	The number of the drive to check (0 = drive A:, 1 = drive B:, and so on).
buffer	long	A pointer to a buffer that will store the four longwords of data returned by this call.

Contents of the buffer pointed to by *buffer:*

Longword Number	*Contents*
0	Number of free clusters
1	Total number of clusters
2	Sector size (in bytes)
3	Cluster size (in sectors)

Results

None

Create Directory

Dcreate( ) **Opcode = 57 ($39)**

This function is used to create a new subdirectory.

C macro format

```
int status;
char *pathname;
   status = Dcreate(pathname);
```

Machine language format

```
move.l      pathname, -(sp)
move.w      #$39, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

pathname	long	A pointer to a null-terminated ASCII directory path string (for instance, *C:\NEWDIR*).

Results

D0	status	word	A 0 means that the directory was created without problem, otherwise a negative GEMDOS error number is returned.

See also

Ddelete( )

Delete Directory

Ddelete( ) **Opcode = 58 ($3A)**

Used to delete a subdirectory, provided that it contains no files.

C macro format

```
int status;
char *pathname;
   status = Ddelete(pathname);
```

Machine language format

```
move.l      pathname, -(sp)
move.w      #$3A, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

pathname	long	A pointer to a null-terminated ASCII directory path string (for instance, *C:\NEWDIR*).

Results

D0	status	word	A 0 means that the directory was deleted without problem, otherwise a negative GEMDOS error number is returned.

See also

Dcreate( )

Set Default Directory Path

Dsetpath( ) **Opcode = 59 ($3B)**

This function is used to set a default directory on a drive, which is where GEMDOS will search first for a named file.

C macro format

```
int status;
char *path;
   status = Dsetpath(path)
```

Machine language format

```
move.l         path, -(sp)
move.w         #$3B, -(sp)
trap           #1
addq.l         #6,sp
```

Inputs

path	long	A pointer to a null-terminated ASCII directory path string (for instance, *WORDPRO\LETTERS\FRED*). If the path name begins with a drive letter and a colon, the path is set for that drive rather than the current default drive (for instance, *C:\DATABASE\CLIENTS*).

Results

D0	status	word	A 0 means that the directory was selected without problem, otherwise a negative GEMDOS error number is returned.

See also

Dgetpath( )

Create File

Fcreate( ) **Opcode = 60 ($3C)**

Creates a new file, or if the specified file already exists, truncates it to length 0. The file is opened, and a 16-bit handle is returned that can be used for further access to the file. (write only mode.)

C macro format

```
char *fname;
int handle, attr;
   handle = Fcreate(fname, attr);
```

Machine language format

```
move.w      attr, -(sp)
move.l      fname, -(sp)
move.w      #$3C, -(sp)
trap        #1
addq.l      #8,sp
```

Inputs

attr	word	A flag that specifies the file's attributes.

Interpretation of *attr* flag:

Bit Number	*Bit Value*	*Description*
0	1	Read-only file (can't be deleted or written to)
1	2	Hidden file (excluded from normal directory searches)
2	4	System file (excluded from normal directory searches)
3	8	Volume label (can only exist in root)

fname	long	A pointer to a null-terminated ASCII string containing the name of the file to create.

Results

D0	handle	word	If the function succeeds, a 16-bit handle value. If not, a negative GEMDOS error number.

See also

Fopen( )

Open File

Fopen( ) **Opcode = 61 ($3D)**

Opens a specified file, and returns a 16-bit handle that can be used for further access to the file.

C macro format

```
char *fname;
int handle, mode;
    handle = Fopen(fname, mode);
```

Machine language format

```
move.w      #mode,-(sp)
move.l      fname,-(sp)
move.w      #$3D,-(sp)
trap        #1
addq.l      #8,sp
```

Inputs

mode	word	A flag that specifies which operations will be available once the file has been opened. 0 = Read only 2 = Write only 3 = Read or Write
fname	long	A pointer to a null-terminated ASCII string containing the name of the file to open.

Results

D0	handle	word	If the function succeeds, a 16-bit handle value. If not, a negative GEMDOS error number.

See also

Fcreate( ), Fclose( )

Close File

Fclose( ) **Opcode = 62 ($3E)**

When a program is finished with a file, it must use this function to close the file.

C macro format

```
int handle, status;
   status = Fclose(handle);
```

Machine language format

```
move.w      handle,-(sp)
move.w      #$3E,-(sp)
trap        #1
addq.l      #4,sp
```

Inputs

handle	word	The file handle of the file to close.

Results

D0	status	word	A 0 means success, failure is indicated by the appropriate GEMDOS error number.

See also

Fcreate( ), Fopen( )

Read File

Fread( ) **Opcode = 63 ($3F)**

This function reads a specified number of bytes from an open file.

C macro format

```
long status, count;
int handle;
char *buffer;
   status = Fread(handle, count, buffer);
```

Machine language format

```
move.l          buffer, -(sp)
move.l          #count, -(sp)
move.l          #handle, -(sp)
move.w          #$3F, -(sp)
trap            #1
add.l           #12,sp
```

Inputs

buffer	long	A pointer to the buffer where the data that is read in will be stored.
count	long	The number of bytes to read. Note: This must be a long value.
handle	word	The file handle of the file to read.

Results

D0	status	word	If the function succeeds, the actual number of bytes read is returned. If the function attempts to read past the end of the file, 0 is returned. For any other error, a negative GEMDOS error number is returned

See also

Fopen( ), Fclose( )

Write File

Fwrite( ) **Opcode = 64 ($40)**

This function writes a specified number of bytes to an open file.

C macro format

```
long status, bytes;
int handle;
char *buffer;
   status = Fwrite(handle, bytes, buffer);
```

Machine language format

```
move.l      buffer, -(sp)
move.l      #bytes, -(sp)
move.l      #handle, -(sp)
move.w      #$40, -(sp)
trap        #1
add.l       #12,sp
```

Inputs

buffer	long	A pointer to the buffer where the data to be written is stored.
bytes	long	The number of bytes to write. Note: This must be a long value.
handle	word	The file handle of the file to write to.

Results

D0	status	word	If the function succeeds, the actual number of bytes written is returned. If not, a negative GEMDOS error number is returned

See also

Fopen( ), Fclose( )

Delete File

Fdelete( ) **Opcode = 65 ($41)**

Deletes a specified file from its directory.

C macro format

```
int status;
char *filename;
   status = Fdelete(filename);
```

Machine language format

```
move.l      filename,-(sp)
move.w      #$41,-(sp)
trap        #1
addq.l      #6,sp
```

Inputs

filename	long	A pointer to a null-terminated ASCII string that contains the name of the file to be deleted

Results

D0	status	word	If successful, 0 is returned, if not, a negative GEMDOS error number.

See also

Fcreate( )

Seek File

Fseek( ) **Opcode = 66 ($42)**

Allows you to move the file pointer to change the position in the file at which data will be read or written.

C macro format

```
long position, offset;
int handle, seekmode;
   position = Fseek(offset, handle, seekmode);
```

Machine language format

```
move.w      #seekmode, - (sp)
move.w      #handle, - (sp)
move.l      #offset, - (sp)
move.w      #$42, - (sp)
trap        #1
add.l       #10,sp
```

Inputs

seekmode	word	A flag that indicates the position relative to which the file pointer will move: 0 = Beginning of file 1 = Current position 2 = End of file
handle	word	The handle for the file whose pointer will be moved.
offset	long	The number of bytes to move the file pointer. A positive number moves toward the end of the file, a negative number moves toward the beginning.

Results

D0	position	long	The absolute current position of the file pointer after the Fseek( ) call, expressed as an offset from the start of the file.

Get/Set File Attributes

Fattrib( ) **Opcode = 67 ($43)**

This function can be used to read or change file's attributes.

C macro format

```
int attributes, mode, newattr;
char *filename;
   attributes = Fattrib(filename, mode, newattr);
```

Machine language format

```
move.w      newattr, -(sp)
move.w      mode, -(sp)
move.l      filename, -(sp)
move.w      #0, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

newattr	word	A bit flag that indicates the file's new attributes.

Interpretation of *newattr* flag:

Bit Number	*Attribute*
0	Read-only file (can't be deleted or written to)
1	Hidden file (excluded from normal directory searches)
2	System file (excluded from normal directory searches)
3	Volume label (can only exist in root)
4	Subdirectory
5	Archive bit

mode	word	A flag that determines whether the attributes are read or changed: 0 = Read 1 = Change
filename	long	A pointer to a null-terminated ASCII string that contains the name of the files whose attributes will be read or changed.

Results

D0	attributes	word	If *mode* is set to 0, the current file attributes are returned here, in the same format as described for *newattr*, above.

See also

Fcreate( )

Duplicate Standard File Handle

Fdup( ) **Opcode = 69 ($45)**

This function is used as part of the file redirection process. It creates a user-designated file handle that duplicates the function of one of the standard file handles. This allows you to use Fforce( ) (that only works with user-designated file handles) with standard devices.

C macro format

```
int newhandle, handle;
   newhandle = Fdup(handle);
```

Machine language format

```
move.w      handle, -(sp)
move.w      #$45-(sp)
trap        #1
addq.l      #4,sp
```

Inputs

handle	word	One of the standard device handles (0–5).

Results

D0	newhandle	word	If successful, a user-designated file handle (greater than 5). If not, a GEMDOS error number.

See also

Fforce( )

Replace Standard File Handle

Fforce( ) **Opcode = 70 ($46)**

Permits you to redirect I/O from a standard device handle to a user-designated one.

C macro format

```
word status, standard, user;
   status = Fforce(standard, user);
```

Machine language format

```
move.w      user, -(sp)
move.w      standard, -(sp)
move.w      #$46, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

user	word	The handle of the user-designated file that replaces the standard file.
standard	word	The handle of the standard file to be replaced.

Results

D0	status	word	A 0 indicates success, otherwise a negative GEMDOS error number is returned.

See also

Fdup( )

Get Default Directory Path

Dgetpath( ) **Opcode = 71 ($47)**

This function is used to find the default directory on a drive, which is where GEMDOS searches first for a named file.

C macro format

```
word drivenum;
char *buffer;
   Dgetpath(buffer, drivenum);
```

Machine language format

```
move.w      #drivenum, -(sp)
move.l      buffer, -(sp)
move.w      #0, -(sp)
trap        #1
addq.l      #8,sp
```

Inputs

drivenum	word	The number of the drive (0–15) whose default path you want to find (0 = drive A:, 1 = drive B:, and so forth).
buffer	long	A pointer to the buffer where the function returns the path name (allocate at least 128 bytes).

Results

None

See also

Dsetpath( )

Allocate Memory Block

Malloc( ) **Opcode = 72 ($48)**

Used to allocate some of the system's free memory, and reserve it for use by the program. After the program is finished using the memory, it should return it with Mfree( ). Malloc( ) can also be used to determine the size of the largest block of memory in the free memory pool. Because of a bug in Malloc( ), programs should try to get all the memory they need with one big Malloc( ) call, instead of several little ones.

C macro format

```
long address, bytes;
   address = Malloc(bytes);
```

Machine language format

```
move.l      #bytes,-(sp)
move.w      #$48,-(sp)
trap        #1
addq.l      #6,sp
```

Inputs

bytes	long	The number of bytes of memory to allocate. If set to −1, the function returns the size of the largest block of free memory in *address*.

Results

D0	address	long	If the function is able to allocate *bytes* amount of memory, the starting address of the memory block is returned here. If there isn't sufficient free memory to allocate the block that was requested, a value of 0 is returned.

See also

Mfree( )

Free Memory Block

Mfree( ) **Opcode = 73 ($49)**

This function is used to return memory that was allocated with Malloc( ) back to the system.

C macro format

```
int status;
long address;
   status = Mfree(address);
```

Machine language format

```
move.l      address, -(sp)
move.w      #$49, -(sp)
trap        #1
addq.l      #6,sp
```

Inputs

address	long	The address of the memory block to be returned.

Results

D0	status	word	A 0 indicates successful return of the memory. A negative number indicates an error.

See also

Malloc( )

Shrink Memory Block

Mshrink( ) **Opcode = 74 ($4A)**

When a program is run, it is allocated all the system's free memory. This function is used to return any excess memory back to the system.

C macro format

```
int status;
long address, size;
   status = Mshrink(0, address, size);
```

Machine language format

```
move.l     size,-(sp)
move.l     address,-(sp)
move.w     #$4A,-(sp)
trap       #1
add.l      #10,sp
```

Inputs

size	long	The number of bytes to retain.
address	long	The starting address of the memory block to retain.

Note that a zero word must also be passed, as a place holder.

Results

D0	status	word	A 0 signals success, failure results in a negative GEMDOS error number.

See also

Malloc( )

Execute Process

Pexec( ) **Opcode = 75 ($4B)**

This function is used to load a program file and execute it.

C macro format

```
*char file, command, env;
int mode;
long status;
    status = Pexec(mode, file, command, env);
```

Machine language format

```
move.l      env, -(sp)
move.l      command, -(sp)
move.l      file, -(sp)
move.w      #mode, -(sp)
move.w      #$4B, -(sp)
trap        #1
add.l       #14,sp
```

Inputs

The meaning of the input parameters vary according to the value of *mode*. There are four different modes of operation for the Pexec( ) function. These are:

Mode Number	*Function*	*file*	*command*	*env*
0	Load and execute	Pointer to filename string	Pointer to command string	Pointer to environment string
3	Just load, don't execute	Pointer to filename string	Pointer to command string	Pointer to environment string
4	Just execute	Unused	Basepage address	Unused
5	Create basepage	Unused	Pointer to command string	Pointer to environment string

Results

D0	status	long	The address of the basepage, or an error number.

Terminate Process with Return Code

Pterm( ) **Opcode = 76 ($4C)**

Like Pterm0( ), Pterm( ) terminates the current process, closes all open files, clears the memory space used by the process, and exits to the calling program, usually the GEM Desktop. In addition, it allows a return code to be passed to the calling program. This function is mainly of interest to machine language programmers, since the startup modules supplied with C language compilers automatically call one of the terminate functions when the main( ) function exits.

C macro format

```
int retcode;
    Pterm(retcode);
```

Machine language format

```
move.w      #retcode, -(sp)
move.w      #$4C, -(sp)
trap        #1
```

Inputs

retcode	word	The exit code that is returned to the calling program.

Results

None

See also

Pterm0( ), Ptermres( )

Find First File in Directory Chain

Fsfirst( ) **Opcode = 78 ($4E)**

This function is used to find the first file in a directory that matches a particular description. It is used in obtaining a directory listing.

C macro format

```
int status, attributes;
char *filespec;
   status = Fsfirst(filespec, attribs);
```

Machine language format

```
move.w     attribs, -(sp)
move.l     filespec, -(sp)
move.w     #$4E, -(sp)
trap       #1
addq.l     #8,sp
```

Inputs

attribs	word	The attributes of the file to search for.

Interpretation of value in *attribs:*

Bit Number	*Attribute*
0	Read-only file (can't be deleted or written to)
1	Hidden file (excluded from normal directory searches)
2	System file (excluded from normal directory searches)
3	Volume label (can only exist in root)
4	Subdirectory
5	Archive bit

filespec	long	Pointer to a null-terminated ASCII string containing the file specification to be searched for. If wildcards are used, it may match more than one file.

Results

D0 status word If a match is found, a value of 0 is returned here, and a 44-byte data structure is written to the address pointed to by the DTA. If a match isn't found, the appropriate negative GEMDOS error number is returned.

Information in data structure returned if match is successful:

Byte Number	*Contents*
0–20	Reserved for internal use (must not be altered)
21	File attributes
22–23	Time stamp
24–25	Date stamp
26–29	File size
30–43	Filename and extension

See also

Fsnext( )

Find Next File in Directory Chain

Fsnext( ) **Opcode = 79 ($4F)**

This function is used to find additional files in a directory chain that meet the criteria set forth in a previous Fsfirst( ) call. This function will only work if the file specification used for Fsfirst( ) included a wildcard character.

C macro format

```
int status;
   status = Fsnext( );
```

Machine language format

```
move.w        #$4F,-(sp)
trap          #1
addq.l        #2,sp
```

Inputs

None—search criteria are set by previous Fsfirst( ) call.

Results

D0	status	word	If a match is found, a value of 0 is returned here, and a 44-byte data structure is written to the address pointed to by the DTA. If no match is found, the appropriate negative GEMDOS error number is returned.

Information in data structure returned if match is successful:

Byte Number	*Contents*
0–20	Reserved for internal use (must not be altered)
21	File attributes
22–23	Time stamp
24–25	Date stamp
26–29	File size
30–43	Filename and extension

See also

Fsfirst( )

Rename File

Frename( ) **Opcode = 86 ($56)**

Changes the name of a file.

C macro format

```
int status;
char *oldname;
char *newname;
   status = Frename (0, oldname, newname);
```

Machine language format

```
move.l      newname,-(sp)
move.l      oldname,-(sp)
clr.w       -(sp)
move.w      #$56,-(sp)
trap        #1
add.l       #12,sp
```

Inputs

newname	long	A pointer to the string containing the filename to change to. This filename may be in a new directory path.
oldname	long	A pointer to a string containing the name of the file to be changed.

Note that a zero word must also be passed as a place holder.

Results

D0	status	word	If successful, a 0 is returned. If not, a negative GEMDOS error code.

Get/Set File Date/Time Stamp

Fdatime( ) **Opcode = 87 ($57)**

This function is used to read or change the time and date stamp of an open file.

C macro format

```
int handle, mode;
long *timeptr;
   Fdatime(timeptr, handle, mode);
```

Machine language format

```
move.w      mode, -(sp)
move.w      handle, -(sp)
move.l      timeptr, -(sp)
move.w      #$57, -(sp)
trap        #1
addq.       #10,sp
```

Inputs

mode	word	Read or change flag: 0 = Read 1 = Change
handle	word	The handle of the file
timeptr	long	The address of a 32-bit buffer that holds the time and date stamp information. The date information is stored in the first word.

Contents of first word of buffer pointed to by *timeptr*:

Bit Number	*Description*	*Range*
0–4	Day	1–31
5–8	Month	1–12
9–15	Year − 1980	0–119

The second word holds the time, in the following format:

Bit Number	*Description*	*Range*
0–4	Seconds divided by 2	0–29
5–10	Minutes	0–59
11–15	Hour	0–23

Results

None

See also

Tgettime( ), Tsettime( )

Appendix D

Error Codes

BIOS Errors

Error Number	Macro Name	Description
0	OK	Action successful (no error)
−1	ERROR	General error
−2	DRIVE_NOT_READY	Device not ready, not connected, or busy
−3	UNKNOWN_CMD	The device doesn't know how to respond to this command
−4	CRC_ERROR	A soft error occured while reading a sector
−5	BAD_REQUEST	Device can't handle this command, possibly because of bad parameters
−6	SEEK_ERROR	Drive could not seek to that track
−7	UNKNOWN_MEDIA	The boot sector of this disk doesn't follow the ST format
−8	SECTOR_NOT_FOUND	Sector couldn't be located
−9	NO_PAPER	Printer out of paper
−10	WRITE_FAULT	Write operation failed
−11	READ_FAULT	Read operation failed
−12	GENERAL_MISHAP	Reserved for future use
−13	WRITE_PROTECT	Write operation failed because media was write-only or write-protected
−14	MEDIA_CHANGE	Write operation failed because disk was changed since last write operation
−15	UNKNOWN_DEVICE	Operation requested for a device that is unkown to the BIOS

−16	BAD_SECTORS	Format operation resulted in one or more bad sectors
−17	INSERT_DISK	Ask user to insert disk

GEMDOS Errors

MS-DOS Error Number	GEMDOS Error Number	Macro Name	Description
1	−32	EINVFN	Invalid function number
2	−33	EFILNF	File not found
3	−34	EPTHNF	Path not found
4	−35	ENHNDL	No file handles available
5	−36	EACCON	Access denied
6	−37	EIHNDL	Invalid handle
8	−39	ENSMEM	Insufficient memory
9	−40	EIMBA	Invalid memory block address
15	−46	EDRIVE	Invalid drive specification
18	−47	ENMFIL	No more files
	−64	ERANGE	Range error
	−65	EINTRN	GEMDOS internal error
	−66	EPLFMT	Not an executable file
	−67	EGSBF	Memory block growth failure

Appendix E

VT-52 Console Escape Sequences

Unlike GEM graphics text functions, which output any character for which there is image data, the console device screen emulates a DEC VT-52 display terminal, and treats the ASCII characters from 0 to 31 as nonprinting control characters. This means that it interprets the ASCII character 13 as a carriage return, an instruction to move the cursor to the beginning of the line, rather than as a character that should be printed. There are a number of VT-52 escape codes to which the console device responds. These escape sequences are strings of characters beginning with the ASCII character 27 (Esc), followed by one or more text characters. The VT-52 codes to which Bconout() responds are:

Esc A	Cursor Up
Esc B	Cursor Down
Esc C	Cursor Right
Esc D	Cursor Left
Esc E	Clear Screen and Home Cursor
Esc H	Home Cursor
Esc I	Cursor Up (scrolls screen down if at top line)
Esc J	Clear to End of Screen
Esc K	Clear to End of Line
Esc L	Insert Line
Esc M	Delete Line
Esc Y (row + 32) (column + 32)	Position Cursor at Row, Col (starts with 0)
Esc b (register)	Select Foreground (Character) Color
Esc c (register)	Select Background Color
Esc d	Clear to Beginning of Screen
Esc e	Cursor On
Esc f	Cursor Off
Esc j	Save Cursor Position
Esc k	Move Cursor to Saved Position
Esc l	Clear line

Esc o	Clear from Beginning of Line
Esc p	Reverse Video On
Esc q	Reverse Video Off
Esc v	Line Wrap On
Esc w	Line Wrap Off

In addition to the escape codes, the ST terminal emulation also responds to the following ASCII control codes:

08	Backspace
09	Tab
10–12	Linefeed
13	Carriage Return

Appendix F

The MFP Chip

The 68901

Multi-Function Peripheral (MFP) chip handles a variety of input/output chores on the ST. It contains an 8-bit parallel I/O port, each bit of which is used for a different purpose. The port is used for monochrome monitor detection, RS-232 carrier detect, ring detect, and clear to send (CTS), parallel port and blitter-busy detection a unit, and data requests from the floppy drives, DMA port, intelligent keyboard controller, and MIDI ports. The MFP chip's onboard Universal Synchronous/Asynchronous Receiver/Transmitter (USART) provides the hardware interface for the ST's serial port. The chip also has four general-purpose timers. They're used for the RS-232 baud rate generator, the 200 Hz system clock, and as a horizontal blanking counter.

The serial port, the timers, and each bit of the parallel I/O port are capable of generating an interrupt. The MFP chip interrupts have an Interrupt Priority Level (IPL) of six, but they are not autovectored. This means that when an MFP interrupt occurs, the IPL 6 interrupt handler is not called. Instead, the MFP chip directs exception processing through one of its 16 exception vectors. These are vectors 64–80, which are located at addresses 256 – 323 ($100 – $143). Although all MFP interrupts have an overall 68000 IPL of 6, there are subpriorities that are handled by the MFP chip.

MFP Registers

The ST system communicates with the MFP chip through its 24 8-bit registers. On the current ST models, these registers are found at the 24 odd addresses, starting at address $FFFFFA01. Since this may change with future models, programmers should, whenever possible, communicate with the MFP chip through XBIOS routines like Xbtimer( ), Rsconf( ), Jdisint( ), Jenabint( ) and Mfpint( ). The MFP registers, and their functions, are:

Register 1. General Purpose I/O Interrupt Port (GPIP). This is the data register for the 8-bit parallel I/O port, where the data is read and written.

Register 2. Active Edge Register (AER). For parallel port input bits. This register specifies whether the interrupt will occur on low-to-high transitions (1), or high-to-low transitions (0).

Register 3. Data Direction Register (DDR). This register specifies whether each bit of the parallel I/O port will be used for input (0), or output (1).

Register 4. Interrupt Enable Register A (IERA). Each bit of this register is used to determine whether the corresponding interrupt will be enabled (1), or disabled (0). These are detailed in Table F-1. The interrupts that have an asterisk next to their priority level are initially disabled.

Table F-1. Interrupt Disable Register A (IERA)

Bit Number	Internal Priority Level	Interrupt Source	Function
7	15*	I/O Port Bit 7	Monochrome monitor detect
6	14*	I/O Port Bit 6	RS-232 Ring Indicator (RI)
5	13*	Timer A	User-defined timer interrupt
4	12	USART	RS-232 receive buffer full
3	11	USART	RS-232 receive error
2	10	USART	RS-232 transmit buffer empty
1	9	USART	RS-232 transmit error
0	8*	Timer B	Horizontal blank counter

Register 5. Interrupt Enable Register B (IERB). Each bit of this register is used to determine whether the corresponding interrupt will be enabled (1), or disabled (0). The interrupts that have an asterisk next to their priority level are initially disabled.

Table F-2. Interrupt Enable Register B (IERB)

Bit Number	Internal Priority Level	Interrupt Source	Function
7	7*	I/O Port Bit 5	Floppy drive/DMA port data request
6	6	I/O Port Bit 4	Keyboard and MIDI ACIA data request

Bit Number	Internal Priority Level	Interrupt Source	Function
5	5	Timer C	System Clock (200 Hz)
4	4*	Timer D	RS-232 baud rate generator
3	3*	I/O Port Bit 3	Graphics blitter chip done
2	2	I/O Port Bit 2	RS-232 Clear To Send (CTS)
1	1*	I/O Port Bit 1	RS-232 Data Carrier Detect (DCD)
0	0*	I/O Port Bit 0	Parallel port busy

Registers 6 and 7. Interrupt Pending Register A (IPRA) and Interrupt Pending Register B (IPRB), respectively. When an interrupt is triggered by one of the A or B group of events, the appropriate bit in the IPRA or IPRB is set to 1. In AEI mode, this bit is cleared automatically when execution is vectored to the interrupt handler. In SEI mode (used on the ST), it must be cleared by the software.

Registers 8 and 9. Interrupt In-Service Register A (ISRA) and Interrupt In-Service Register B (ISRB), respectively. When an interrupt has been triggered by one of the A or B group of events, the corresponding bit in the ISRA or ISRA is set to one. This prevents a second, lower-priority interrupt from occurring while the first is still being processed. In AEI mode, this bit is cleared automatically when execution is vectored to the interrupt handler. In SEI mode (used on the ST), it must be cleared by the software.

Registers 10 and 11. Interrupt Mask Register A (IMRA) and Interrupt Mask Register B (IMRB), respectively. These mask registers can be used to stop one of the A or B groups of events from triggering automatic exception processing. A masked interrupt will still cause the appropriate bit of IPRA or IPRB to be set, however.

Register 12. Vector Register (VR). The vector register helps specify the vectors used for the 16 interrupts. The top four bits of this register hold the high nibble for the vector number used by each of the interrupts. Of the low nibble, only bit 3 is used. This bit determines whether the MFP operates in Automatic End-of-Interrupt mode (AEI), or Software End-of-Interrupt mode (SEI), like the ST.

Registers 13 and 14. Timer A Control Register (TACR) and Timer B Control Register (TBCR), respectively. Timers A and B function identically. Each has a data register, a control register, and an 8-bit interval counter, whose value decrements by one at every impulse. When the interval counter timer reaches 0, the value of the data register is loaded into the counter, and an interrupt will be generated if enabled.

The source of the impulse that causes the counter to decrement may be a clock pulse (Delay mode), or an external signal (Event Count mode). There is also a combined mode (Pulse Length mode), in which the clock is turned on and off by the external signal.

The control registers use bits 0–3 to determine the mode in which the timers will function. The possible combinations are detailed in Table F-3.

Table F-3. Timer A and B Register Values and Timer Modes

Register Value	Timer Mode
0	Timer off
1	Delay mode, clock divided by 4
2	Delay mode, clock divided by 10
3	Delay mode, clock divided by 16
4	Delay mode, clock divided by 50
5	Delay mode, clock divided by 64
6	Delay mode, clock divided by 100
7	Delay mode, clock divided by 200
8	Event Count Mode
9	Pulse Length mode, clock divided by 4
10	Pulse Length mode, clock divided by 10
11	Pulse Length mode, clock divided by 16
12	Pulse Length mode, clock divided by 50
13	Pulse Length mode, clock divided by 64
14	Pulse Length mode, clock divided by 100
15	Pulse Length mode, clock divided by 200

Bit four (value of 16) can be used to force a timer reset.

Register 15. Timers C and D Control Register (TCDCR). Since Timers C and D only run in Delay mode, only one byte is needed for their control register. The upper nibble controls Timers C, while the lower nibble controls timer D, as shown in Table F-4.

Table F-4. Control Register Values for Timers C and D

Timer C Value	Timer D Value	Timer Mode
0	0	Timer off
16	1	Delay mode, clock divided by 4
32	2	Delay mode, clock divided by 10

Table F-4. Control Register Values for Timers C and D

Timer C Value	Timer D Value	Timer Mode
48	3	Delay mode, clock divided by 16
64	4	Delay mode, clock divided by 50
80	5	Delay mode, clock divided by 64
96	6	Delay mode, clock divided by 100
112	7	Delay mode, clock divided by 200

Registers 16–19. Timers A–D Data Registers (TADR, TBDR, TCDR, and TDDR). The four timer data registers are used to store the countdown value for the interval counter.

Register 20. Synchronous Character Register (SCR). In synchronous transfer mode, all data characters received are stored in the SCR as well as the receive buffer, which signals the application when a character has been received.

Register 21. USART Control Register (UCR). This register is used to set the serial interface communications parameters as shown in Table F-5.

Table F-5. USART Control Register (UCR)

Bit	Function
0	Not used.
1	Parity type. 0 = Odd. 1 = Even.
2	Parity enable. 0 = Off. 2 = On.
3–4	Async start and stop bits.

Bits

4	*3*	*Number of Start and Stop Bits*
0	0	No start or stop bits (synchronous).
0	1	1 start bit, 1 stop bit.
1	0	1 start bit, 1½ stop bits.
1	1	1 start bit, 2 stop bits.

Bit	Function
5–6	Data bits per word.

Bits

5	*6*	*Number of data bits*
0	0	8 bits.
0	1	7 bits.
1	0	6 bits.
1	1	5 bits.

Table F-5. USART Control Register (UCR)

Bit	Function
7	Clock. 0 = Use clock directly for transfer frequency (synchronous transfer). 1 = Divide clock frequency by 16.

Register 22. Receiver Status Register (RSR). This register contains information about serial port reception as shown in Table F-6.

Table F-6. Receiver Status Register (RSR)

Bit	Function
0	Receiver Enable Bit. 0 = Receipt disabled. 1 = Receipt enabled.
1	Synchronous Strip Enable. In synchronous mode, this bit enables checking of whether the character in the SCR is identical to a character in the receive buffer.
2	Match/Character in Progress. In synchronous mode, this bit signals that a character matching the SCR byte would be received. In asynchronous mode, this bit is set when the start bit is detected, and cleared when the stop bit is detected.
3	Search/Break Detected. In synchronous mode, this bit signals that a character was received that matches the SCR byte. In asynchronous mode, this bit is set when a BREAK is received.
4	Frame Error. This bit is set when the byte received is not a null, but the stop bit is a null.
5	Parity Error. This bit is set when the parity bit of the last received character was incorrect.
6	Overrun Error. This bit is set when a character can't be read into the receive buffer.
7	Buffer Full. This bit is set when a character is transferred into the receive buffer, and cleared when it's read.

Register 23. Transmitter Status Register (TSR). This register contains information about serial port transmission (Table F-7).

Table F-7. Transmitter Status Register (TSR)

Bit	Function
0	Transmit Enable. 0 = Transmission disabled. 1 = Transmission enabled.

Table F-7. Transmitter Status Register (TSR)

Bit	Function
1–2	High/Low Output. 0 = High output. 1 = Low output. 2 = High output. 3 = Loop-back mode.
3	Break. In asynchronous mode, a break is sent when this bit is set.
4	End of Transmission.
5	Auto Turnaround.
6	Underrun Error.
7	Buffer Empty. This bit is set when a character is transferred from the transmit buffer, and cleared when new data is written to it.

Register 24. USART Date Register (UDR). Serial data is sent and received through this register. Writing a value places data in the send buffer, while reading this register returns the next character in the receive buffer.

Appendix G

System Characters

This appendix includes all the system font. The font supports all characters from 0 through 255.

0		13	CR	26	ə	39	'
1	⇧	14	/|	27	ES	40	(
2	⇩	15	|\	28	ɛ	41	)
3	⇨	16	0	29	⤵	42	*
4	⇦	17	1	30	✗	43	+
5	⌛	18	2	31	✓	44	,
6	◩	19	3	32	Space	45	-
7	◈	20	4	33	!	46	.
8	√	21	5	34	"	47	/
9	◷	22	6	35	#	48	0
10	🔔	23	7	36	$	49	1
11	♪	24	8	37	%	50	2
12	FF	25	9	38	&	51	3

52	4	66	B	80	P	94	^
53	5	67	C	81	Q	95	_
54	6	68	D	82	R	96	`
55	7	69	E	83	S	97	a
56	8	70	F	84	T	98	b
57	9	71	G	85	U	99	c
58	:	72	H	86	V	100	d
59	;	73	I	87	W	101	e
60	<	74	J	88	X	102	f
61	=	75	K	89	Y	103	g
62	>	76	L	90	Z	104	h
63	?	77	M	91	[	105	i
64	@	78	N	92	\	106	j
65	A	79	O	93	]	107	k

108	l	122	z	136	ê	150	û
109	m	123	{	137	ë	151	ù
110	n	124	|	138	è	152	ÿ
111	o	125	}	139	ï	153	Ö
112	p	126	~	140	î	154	Ü
113	q	127	∆	141	ì	155	¢
114	r	128	Ç	142	Ä	156	£
115	s	129	ü	143	Å	157	¥
116	t	130	é	144	É	158	ß
117	u	131	â	145	æ	159	ƒ
118	v	132	ä	146	Æ	160	á
119	w	133	à	147	ô	161	í
120	x	134	å	148	ö	162	ó
121	y	135	ç	149	ò	163	ú

164	ñ	178	Ø	192	ĳ	206	מ
165	Ñ	179	ø	193	Ĳ	207	נ
166	ª	180	œ	194	א	208	ס
167	º	181	Œ	195	ב	209	ע
168	¿	182	À	196	ג	210	פ
169	⌐	183	Ã	197	ד	211	צ
170	¬	184	Õ	198	ה	212	ק
171	½	185	¨	199	ו	213	ר
172	¼	186	´	200	ז	214	ש
173	¡	187	†	201	ח	215	ת
174	«	188	¶	202	ט	216	ן
175	»	189	©	203	י	217	ך
176	ã	190	®	204	כ	218	ם
177	õ	191	™	205	ל	219	ף

220	ץ	234	Ω	248	°
221	§	235	δ	249	•
222	∧	236	∮	250	·
223	∞	237	φ	251	√
224	α	238	∈	252	ⁿ
225	β	239	∩	253	²
226	Γ	240	≡	254	³
227	π	241	±	255	¯
228	Σ	242	≥		
229	σ	243	≤		
230	µ	244	⌠		
231	τ	245	⌡		
232	Φ	246	÷		
233	Θ	247	≈		

Appendix H

The Line A Variable Table

The line A routines use about 1K of variable space, in which much of the system's graphics data is stored. The location of this variable table may vary from machine to machine, and from one version of TOS to another, but it can always be found by calling the Line A Initialization function (see Chapter 7 for more details). When this function is called, the base address of the line A variable table is returned both in register A0 and D0. Starting at this base address, you can locate the variables in the table by using their offset from the base address. The following chart shows the offset of each line A variable in decimal and hexadecimal format, its name, and a description of its contents. Those variables whose names start with *V_* are used by the ST BIOS character output routines.

−910 to −905 **(−$38E to −$389)**

RESERVED

This area is reserved for internal use.

−906 to −901 **(−$388 to −$387)**

CUR_FONT

A pointer to the current font header is stored here.

−902 to −855 **(−$386 to −$355)**

RESERVED

This area is reserved for internal use.

The Following 37 Words Hold the Sprite Definition Block for the Mouse Pointer

−856 to −855 **(−$356 to −$355)**

M_POS_HX

The *x* offset of the mouse hot spot within the 16 × 16 image area.

−854 to −851 **(−$354 to −$353)**

M_POS_HY

The *y* offset of the mouse hot spot within the 16 × 16 image area.

−852 to −849 **(−$352 to −$351)**

M_PLANES

The writing mode for the mouse pointer (usually 1). 1 = VDI mode, −1 = XOR mode. The different colors produced by various combinations of image and mask bits for both modes are shown in the table below:

Image Mask	*VDI mode color*	*XOR mode color*
0 0	Transparent	Transparent
0 1	Background	Background
1 0	Foreground	NOT destination
1 1	Foreground	Foreground

−850 to −847 **(−$350 to −$34F)**

M_CDB_BG

Mouse pointer background color.

−848 to −845 **(−$34E to −$34D)**

M_CDB_FG

Mouse pointer foreground color.

−846 to −782 **(−$34C to −$30F)**

MASK_FORM

Mask and image data for the mouse pointer sprite. This data is stored in interleaved sprite format. The first two words contain the mask data and the image data for line 0 of the sprite, the next two words the mask data and image data for line 1, and so on.

−782 to −693 **(−$30E to −$2B5)**

INQ_TAB

This table, 45 words long, contains the information returned by the VDI function vq_extnd( ). See *COMPUTE!'s Technical*

Reference Guide, Atari ST Volume One: The VDI for more information.

−692 to −603 **(−$2B4 to −$25B)**
DEV_TAB
This table contains the first 45 words of information returned by the VDI function v_opnwk( ). See *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* for more information.

−602 to −601 **(−$25A to −$259)**
GCURX
The current mouse pointer *x* position.

−600 to −599 **(−$258 to −$257)**
GCURY
The current mouse pointer *y* position.

−598 to −597 **(−$256 to −$255)**
M_HID_CT
Number of times the mouse pointer has been hidden. The application must use the Show Mouse function this many times to actually display the mouse pointer (or it may just force display using that option the of Show Mouse function). If this value is set to 0, the pointer is currently being displayed.

−596 to −595 **(−$254 to −$253)**
MOUSE_BT
Current mouse button status. Bit 0 gives the left button status, bit 1 the right button status. A bit value of 0 means that the button is up, while a bit value of 1 means that it is currently pressed. See also CUR_MS_STAT, −348.

−594 to −499 **(−$252 to −$1F3)**
REQ_COL
This table contains 48 words of RGB color values for the 16 VDI color indices, as returned by the VDI function vq_color( ). See *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* for more information.

−498 to −469 **(−$1F2 to −$1D5)**

SIZ_TAB

This table contains the final 12 words of information returned by the VDI function v_opnwk( ). Three words of storage are reserved at the end of the table, making it 15 words in length. See *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* to read more about the information returned by v_opnwk( ).

−468 to −467 **(−$1D4 to −$1D3)**

RESERVED

This area is reserved for internal use.

−466 to −466 **(−$1D2 to −$1D1)**

RESERVED

This area is reserved for internal use.

−464 to −461 **(−$1D0 to −$1CD)**

CUR_WORK

This variable contains the address of the current VDI virtual workstation attribute table.

−460 to −457 **(−$1CC to −$1C9)**

DEF_FONT

A pointer to the default font header is stored here.

−456 to −441 **(−$1C8 to −$1B9)**

FONT_RING

This is a longword array containing four pointers to font headers, the last is which is a null (0) entry. Each of these headers is a actually a linked list, since the last field in a font header is a pointer to the next font header in the list. When the VDI searches for a font, it starts with the header pointed to by FONT_RING[0], and searches through that linked list. When it comes to the end of that list, it continues with the linked list pointed to by FONT_RING[1], and then the linked list pointed to by FONT_RING[2]. The first two entries in FONT_RING are reserved for pointers to system font headers, while FONT_RING[2] is used for a linked list of disk-loaded GDOS fonts.

−440 to −437 **(−$1B8 to −$1B7)**

FONT_COUNT

The number of fonts in the FONT_RING lists.

−438 to −349 **(−$1B6 to −$15D)**

RESERVED

This area is reserved for internal use.

−348 **(−$15C)**

CUR_MS_STAT

Current Mouse Status. This byte contains flag bits indicating whether the mouse buttons are up or down, and whether the mouse has moved or buttons have changed since the last mouse interrupt. The meaning of each bit in the flag is as follows:

Bit Number	*Description*
0	Left mouse button status (0 = up, 1 = down)
1	Right mouse button status (0 = up, 1 = down)
2	Reserved
3	Reserved
4	Reserved
5	Mouse move flag (0 = didn't move, 1 = did move)
6	Right mouse button status change flag 0 = status didn't change 1 = status changed
7	Left mouse button status change flag 0 = status didn't change 1 = status changed

−347 **(−$15B)**

RESERVED

This area is reserved for internal use.

−346 to −345 **(−$15A to −$159)**

V_HID_CNT

The depth at which the text cursor is hidden.

−344 to −343 **(−$158 to −$157)**

CUR_X

Horizontal position at which mouse pointer is to be drawn. CUR_X, CUR_Y, and CUR_FLAG are used by the vertical blank interrupt handler, which draws the mouse pointer, to determine where and whether to redraw the mouse pointer during the next vertical blank.

−342 to −341 **(−$156 to −$155)**

UR_Y

Vertical position at which the mouse pointer is to be drawn.

−340 **(−$154)**

CUR_FLAG

Mouse pointer draw flag. If nonzero, the mouse pointer will be redrawn during the vertical blanking interval.

−339 **(−$153)**

MOUSE_FLAG

Mouse interrupt processing flag. A zero means mouse interrupt processing is disabled, while a nonzero value means it is enabled.

−338 to −335 **(−$152 to −$14F)**

RESERVED

This area is reserved for internal use.

−334 to −331 **(−$14E to −$14B)**

V_SAV_XY

The first word contains the horizontal position of the saved text cursor, while the second word contains its vertical position.

−330 to −329 **(−$14A to −$149)**

SAVE_LEN

The height (in vertical lines) of the form saved in SAVE_AREA. This value, along with SAVE_ADDR, SAVE_STAT, and SAVE_AREA, are used by the system to save the portion of the screen covered by the mouse pointer.

−328 to −325 **(−$148 to −$145)**

SAVE_ADDR

The address of the first word of screen data saved in SAVE_AREA.

−324 to −321 **(−$144 to −$143)**

SAVE_STAT

Save area status flag.

Bit Number	*Description*
0	Information in save buffer valid? (1 = yes, 0 = no)
1	Width of area save (0 = 16 bits, 1 = 32 bits)
2–15	Reserved

−322 to −067 **(−$142 to −$043)**

SAVE_AREA

Save buffer for the mouse pointer sprite.

−066 to −063 **(−$042 to −$03F)**

USER_TIM

This vector can be used to install a routine that executes during each system timer-tick interrupt. The user's routine should end by jumping to the address stored in NEXT_TIM, below. For more information, see the description of the VDI function vex_timv( ) in *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

−062 to −059 **(−$03E to −$03B)**

NEXT_TIM

See above.

−058 to −055 **(−$03A to −$037)**

USER_BUT

The Button Change vector. For more information, see the description of the VDI function vex_butv( ) in *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

−054 to −050 **(−$036 to −$033)**

USER_CUR

The Cursor Change vector. For more information, see the description of the VDI function vex_curv( ) in *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

−050 to −047 **(−$032 to −$02F)**

USER_MOT

The Mouse Movement vector. For more information, see the description of the VDI function vex_motv( ) in *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI.*

−046 to −045 **(−$02E to −$02D)**
V_CEL_HT
Text cell height (in pixels).

−044 to −043 **(−$02C to −$02B)**
V_CEL_MX
Maximum horizontal text position (in characters). This equals the maximum number of characters on a line, minus one.

−042 to −041 **(−$02A to −$029)**
V_CEL_MY
Maximum vertical text position (in characters). This equals the maximum number of screen rows, minus one.

−040 to −039 **(−$028 to −$027)**
V_CEL_WR
Number of bytes to the next vertical character cell.

−038 to −037 **(−$026 to −$025)**
V_COL_BG
Color register of text background color.

−036 to −035 **(−$024 to −$023)**
V_COL_FG
Color register of text foreground color.

−034 to −031 **(−$022 to −$01F)**
V_CUR_AD
Current text cursor address.

−030 to −029 **(−$01E to −$01D)**
V_CUR_OF
Byte offset from screen base address to the top of the first text character.

−028 to −025 **(−$01C to −$019)**
V_CUR_XY
The character position of the text cursor. The first word contains the column number, while the second word contains the row number.

−024 **(−$018)**
V_PERIOD
Text cursor blink rate (in vertical blanking intervals).

−023 **(−$017)**
V_CUR_CT
Text cursor countdown timer to next blink toggle.

−022 to −019 **(−$016 to −$013)**
V_FNT_AD
Address of monospaced font data.

−018 to −017 **(−$012 to −$011)**
V_FNT_ND
Last ASCII character in font.

−016 to −015 **(−$010 to −$00F)**
V_FNT_ST
First ASCII character in font.

−014 to −013 **(−$00E to −$00D)**
V_FNT_WD
Width of font form in bytes (sum of the widths of all of the characters in the font, divided by 8).

−012 to −011 **(−$00C to −$00B)**
V_REZ_HZ
Horizontal pixel resolution (width in pixels).

−010 to −007 **(−$00A to −$007)**
V_OFF_AD
Address of font offset table (from font header).

−006 to −005 **(−$006 to −$005)**
RESERVED
This area reserved for internal use.

−004 to −003 **(−$004 to −$003)**
V_REZ_VT
Vertical pixel resolution (height in pixels).

−002 to −001 **(−$002 to −$001)**

BYTES_LIN

Width of the destination memory form; set to the save value as WIDTH.

+000 to +001 **(+$000 to +$001)**

PLANES

Number of color bit planes for current screen resolution.

+003 to +003 **(+$002 to +$003)**

WIDTH

The width of the destination memory form in bytes. When the destination is the screen (as is usually the case), the value stored here should be 160 ($A0) for the low- and medium-resolution modes, and 80 ($50) for high resolution.

+004 to +007 **(+$004 to +$007)**

CONTRL

Pointer to the CONTRL array.

+008 to +011 **(+$008 to +$00B)**

INTIN

Pointer to the INTIN array.

+012 to +015 **(+$00C to +$00F)**

PTSIN

Pointer to the PTSIN array.

+016 to +019 **(+$010 to +$013)**

INTOUT

Pointer to the INTOUT array.

+020 to +023 **(+$014 to +$017)**

PTSOUT

Pointer to the PTSOUT array.

+024 to +025 **(+$018 to +$019)**

COLBIT0

The color bit plane value for plane 0 of the display. This variable, and the three that follow, are used to determine the color drawn by several of the line A functions.

+026 to +027 **(+$01A to +$01B)**

COLBIT1

The color bit plane value for plane 1 of the display.

+028 to +029 **(+$01C to +$01D)**

COLBIT2

The color bit plane value for plane 2 of the display.

+030 to +031 **(+$01E to +$01F)**

COLBIT3

The color bit plane value for plane 3 of the display.

+032 to +033 **(+$020 to +$021)**

LSTLIN

This is a flag which indicates whether the last pixel of a line should be drawn. A zero value means that the pixel is drawn, while a nonzero value means that it is not drawn. This flag is used for drawing a series of connected lines using the XOR writing mode, so that the common endpoint doesn't disappear when the second line is drawn.

+034 to +035 **(+$022 to +$023)**

LNMASK

The line-draw pattern mask.

+036 to +037 **(+$024 to +$025)**

WMODE

The VDI Writing mode to use for drawing.

Mode	*Meaning*
0	Replace
1	Transparent,
2	XOR
3	Reverse transparent

For TextBlt, the BitBlt logic modes may be stored here as well:

Mode	*Logic Operation*	*Description*
4	D1 = 0	Clear destination block
5	D1 = S AND D	
6	D1 = S AND (NOT D)	
7	D1 = S	Replace mode
8	D1 = (NOT S) AND D	Erase mode

Mode	*Logic Operation*	*Description*
9	D1 = D	Destination unchanged
10	D1 = S XOR D	XOR mode
11	D1 = S OR D	Transparent mode
12	D1 = NOT (S OR D)	
13	D1 = NOT (S XOR D)	
14	D1 = NOT D	
15	D1 = S OR (NOT D)	
16	D1 = NOT S	
17	D1 = (NOT S) OR D	Reverse transparent mode
18	D1 = NOR (S AND D)	
19	D1 = 1	Fill destination block

+038 to +039 **(+$026 to +$027)**

X1

This location, and the three following, are often used to hold *x* and *y* coordinates for drawing.

+040 to +041 **(+$028 to +$029)**

Y1

See above.

+042 to +043 **(+$02A to +$02B)**

X2

See above.

+044 to +045 **(+02C to +$02D)**

Y2

See above.

+046 to +049 **(+$02E to +$031)**

PATPTR

Pointer to the current fill pattern.

+050 to +051 **(+$032 to +$033)**

PATMSK

This value is ANDed with Y1, and used as an index into the fill pattern. In most cases, the correct value will be the length of the pattern (in lines) minus one.

+052 to +053 **(+034 to +$035)**

MFILL

Multiplane fill pattern flag. A zero indicates a single plane fill pattern, while a nonzero value indicates a multiplane fill pattern.

+054 to +055 (+$036 to +$037)

CLIP

Clipping flag. A 0 here turns clipping off, while any other value turns it on.

+056 to +057 (+$038 to +$039)

XINCL

Left edge of clip rectangle.

+058 to +059 (+$03A to +$03B)

XMAXCL

Right edge of clip rectangle.

+060 to +061 (+$03C to +$03D)

YMINCL

Top of clip rectangle.

+062 to +063 (+$03E to +$03F)

YMAXCL

Bottom of clip rectangle.

+064 to +065 (+$040 to +$041)

XDDA

Accumulator for text scaling. Should be set to $8000 before each TextBlt that requires scaling.

+066 to +067 (+$042 to +$043)

DDAINC

Scaling increment. For scaling up, DDAINC = 256 * (Size2 − Size1) / Size1. For scaling down, DDAINC = 256 * (Size2) / Size1, where Size1 is the actual character point size, and Size2 is the scaled character size.

+068 to +069 (+$044 to +$045)

SCALDIR

Text scaling direction (0 = down, 1 = up)

+070 to +071 (+$046 to +$047)

MONO

Monospaced font flag.

Value	*Meaning*
0	Font is not monospaced, or size may vary due to special effects.
1	Font is monospaced, and uses no special effects other than thickening (boldface).

+072 to +073 **(+$048 to +$049)**

SOURCEX

The *x* coordinate of character to be printed in font form. SOURCEX can usually be computed from information in the font header:

ch = character to be printed − first_ade
SOURCEX = off_table[ch]

+074 to +075 **(+$04A to +$04B)**

SOURCEY

The *y* coordinate of the character to be printed (usually 0).

+076 to +077 **(+$04C to +$04D)**

DESTX

Horizontal screen position where character will be printed.

+078 to +079 **(+$04E to +$04F)**

DESTY

Vertical screen position where character will be printed.

+080 to +081 **(+$050 to +$051)**

DELX

Width of character. Can be computed from information in the font header as follows:
ch = character to be printed − first_ade
SOURCEX = off_table[ch]
DELX = off_table[ch+1] − SOURCEX

+082 to +083 **(+$052 to +$053)**

DELY

Height of the character. Can be taken from the variable form_height in the font header.

+084 to +087 **(+$054 to +$057)**

FBASE

The address of the font's character image data block. This address may be at an offset of 76 bytes from the beginning of the font header, in the dat__table variable.

+088 to +089 **(+$058 to +$059)**

FWDITH

Width the font form in bytes (the sum of the pixel width of all of the characters in the font, divided by 8).

+090 to +091 **(+$05A to +$05B)**

STYLE

TextBlt special effects flag:

Bit Number	*Effect**
0	Thickening (boldface)
1	Lightening
2	Skewing (italics)
3	Underline (not performed by TextBlt)
4	Outline

* Note that it is up to the application itself (or the VDI) to perform underlining.

+092 to +093 **(+$05C to +$05D)**

LITEMASK

Mask used to lighten text (usually $5555). Can be obtained from the skewmask variable of the font header.

+094 to +095 **(+$05E to +$05F)**

SKEWMASK

Mask used to italicize (usually $5555). Can be obtained from the skewmask variable of the font header.

+096 to +097 **(+$060 to +$061)**

WEIGHT

Width by which to thicken text for boldface. Can be obtained from the skewmask variable of the font header.

+098 to +099 **(+$062 to +$063)**

ROFF

Offset above baseline for italicizing (offset of 0, if skewing is not used). Can be obtained from the skewmask variable of the font header.

+100 to +101 **(+$064 to +$065)**

LOFF

Offset below baseline for italicizing (0 if skewing is not used). Can be obtained from the skewmask variable of the font header.

+102 to +103 **(+$066 to +$067)**

SCALE

Text scaling flag (0 = no scaling used).

+104 to +105 **(+$068 to +$069)**

CHUP

Character rotation angle.

Value	*Rotation*
0	No rotation
900	90 degree clockwise rotation
1800	180 degree clockwise rotation
2700	270 degree clockwise rotation

+106 to +107 **(+$06A to +$06B)**

TEXTFG

Text foreground color.

+108 to +111 **(+$06C to +$06D)**

SCRTCHP

Pointer to two contiguous scratch buffers used for special text effects. Each buffer must be large enough to contain the widest character created by text effects. Calculate the number of bytes required for such a character, and double it. Declare a buffer of that size, place its address in SCRTCHP, and its length divided by 2 in SCRPT2

+112 to +113 **(+$070 to +$071)**

SCRPT2

Offset from first text effects scratch buffer to second.

+114 to +115 **(+$072 to +$073)**

TEXTBG

Text background color

+116 to +117 **(+$074 to +$075)**

COPYTRAN

line A Copy Raster mode:

Value	*Meaning*
Zero	Copy Raster Opaque
Nonzero	Copy Raster Transparent

+118 to +121 **(+$076 to +$079)**

SEEDABORT

Pointer to a routine called by the SeedFill function after each horizontal line is filled. If this routine returns a value of zero in register D0, Seed Fill continues filling the next scan line. If this routine returns a nonzero value in D0, Seed Fill aborts. At minimum, this vector should point to the routine:

```
seedabort:  sub.l  d0,d0  rts
```

If this vector is not set before calling Seed Fill, it will probably point to an illegal address, and Seed Fill will bomb.

Appendix I

The Intelligent Keyboard Controller

On the ST, there are two ways of receiving input from the keyboard. The normal method is to use the GEMDOS console device. You can, however, communicate directly with the keyboard itself. That's because on the ST, the keyboard isn't just a dumb, passive hardware device. It's a separate controller, with its own microprocessor, memory, and I/O port. This level of sophistication allows the controller to be used for tasks other than just fetching keystrokes. The ST Intelligent Keyboard device (IKBD) is responsible for tracking input from the mouse and joysticks, and it maintains a time-of-day clock with one-second accuracy.

Keyboard Functions

The IKBD sends the ST a keycode each time a key is pressed or released. A one-byte code is sent when the key is pressed (make), which is the same as the first byte of the extended keyboard codes shown in Appendix J. The one-byte code that is sent when the key is released (break) is the same as the make code plus 128. Thus, if the *a* key has an extended code of $1E61, the make code sent by the IKBD when the *a* is pressed is $1E, and the break code sent when it is released is $9E.

Although there are 128 possible make codes, and 128 break codes, the ST does not use all of these codes for keys. Codes $74 and $75 are sometimes used to translate mouse button presses into keycodes. Codes $F6 through $FF are used for the packet headers which signal that the next few bytes of information will not be keycodes, but rather will contain information about the mouse, joystick, clock, or IKBD status. These packet headers are:

Header	Information Packet Type
$F6	IKBD status record
$F7	Absolute mouse position record
$F8–$FB	Relative mouse position record (bits 0 and 1 indicate button status)
$FC	Time-of-day record
$FD	Joystick report (both sticks)
$FE	Joystick 0 event record
$FF	Joystick 1 event record

Mouse Functions

The IKBD enables the ST mouse to operate in one of three modes. In relative position mode (the default mode used by TOS), the IKBD sends mouse position packets whenever a mouse button is pressed or released, or moves more than a prescribed threshold distance (which the user may set) in any direction. The movement report is sent at full resolution, not scaled to even multiples of the threshold distance. You may designate whether you want to place the *y* origin at the bottom (the ST default) or the top of the coordinate scheme. If you place it at the top, downward motion is shown as positive, and upward motion as negative. If you place it at the bottom, downward movement is negative, and upward movement is positive.

The relative position may change by a value much greater than the threshold distance if mouse reports have been paused, or if the motion occurs while the IKBD is busy reporting other events. If the accumulated motion exceeds the +127 to −128 range of a single report, multiple reports are sent.

The second mouse mode is absolute position mode. In this mode, position reports are not sent until the user asks for them, via an interrogate command. If the user so designates, position reports may also be sent when one of the mouse buttons is pressed or released. The position reports that are sent give the mouse position as absolute *x,y* coordinates. These coordinates are tracked internally by the IKBD, according to a user-designated scale factor which determines how far the mouse has to move before an *x* or *y* value changes. There is also a command for resetting the *x,y* position.

The mode can also operate in keycode mode. In this mode, mouse position changes are translated into the equivalent cursor key strokes (up arrow, right arrow, and so on). The user may set a scale factor that determines how far the mouse must move before a cursor keycode is generated. A right mouse button press is reported as keycode $74, while a left mouse button press is reported as keycode $75. For both mouse movement and button events, break codes are sent immediately after the make codes.

Joystick Functions

The IKBD provides support for four separate modes of operating the joystick. TOS, however, does not provide any support for reading joysticks. One possible reason for this is a partial conflict between use of the joysticks and the mouse. The mouse and joystick 0 share a single port, and the right mouse button and the fire button on joystick 1 have to share one hardware line, making it impossible to read both at once.

After any joystick command, the IKBD reads both ports as if they have joysticks attached, with one of the buttons assigned to each. After any mouse command (except mouse disable, command $12), the IKBD scans port 0 as if a mouse were attached, and "steals" the joystick 1 fire button line for use in reading the right mouse button. In this condition, the only way to read the fire button on joystick 1 is to temporarily disable the mouse with command $12. Then, the fire button on joystick 1 can be read until the mouse is enabled again.

Without direct TOS support, using joysticks in your program can be a little complicated. In order to enable the joysticks, you must give the IKBD the proper command to set the mode. In order to read the data packets that the IKBD returns, you must install your own interrupt handler routine. Details on this procedure are supplied in the section on communications between the IKBD and ST, below.

Three of the joystick modes that the IKBD supports correspond roughly to the three mouse modes. The first is event reporting mode, in which the IKBD sends a joystick information packet whenever the stick position or button status changes. Information is only sent for the stick that changes. Another mode of joystick operation is interrogation mode.

When this mode is in effect, joystick information is only sent after a joystick interrogation command ($16).

In keycode mode, stick position changes are translated into the equivalent cursor key strokes (up arrow, right arrow, and so on). The user may set a time factor that determines how long the stick must be held in one position before the cursor keycode is repeated. A *velocity stick* feature is implemented, which means that if the user holds the stick in one position for a certain amount of time, the cursor keycodes are repeated faster. In this mode, a button press on joystick 0 is reported as keycode $74, while a button press on joystick 1 is reported as keycode $75. For both mouse movement and button events, break codes are sent immediately after the make codes.

The IKBD also supports joystick monitoring mode, and fire button monitoring mode. In these modes, the IKBD does nothing but constantly monitor the joysticks and/or the fire buttons, and send out status reports as fast as possible. This mode is of questionable value for ST programmers. It disables the keyboard, returns information at a rate that forces the system to do nothing but watch the information coming in from the IKBD, and is hard to use under TOS. These conditions make this mode effectively useless.

Clock Functions

The IKBD maintains a time-of-day clock that keeps track of the time and date, to a resolution of one second. This hardware clock is used for the XBIOS functions Settime( ) and Gettime( ), but not for the GEMDOS time and date functions. In the new (blitter) ROMs, however, the GEMDOS clock is set from the IKBD clock at the end of every process.

Status Functions

The IKBD provides a number of functions that allow you to query the joystick and mouse modes, along with other parameter settings. These functions even allow you to read and set the 6301 processor's memory, as well as execute subroutines within that memory space.

Communicating with the IKBD on the ST

As you have seen, the IKBD supports several different functions, each with several modes of operation. To select a function and mode, merely send a message to the controller, using the BIOS function Bconout( ), or the XBIOS functions Ikbdws( ) or Initmous( ). Receiving input from the device is a little more complex than sending output, however. That's because the IKBD device connects to the system via an ACIA serial interface chip, and the system must be ready to receive information from this device at any time.

When the ACIA chip receives information from the IKBD, it causes an interrupt to occur on the 68901 MFP chip. If the interrupt handler determines that the source of the interrupt was the IKBD ACIA, it calls the main IKBD interrupt routine, ikbdsys. The location of the ikbdsys routine can be found via the XBIOS function Kbdvbase( ), which returns a pointer to all of the IKBD interrupt routines, including ikbdsys.

The ikbdsys routine checks to see if the interrupt was caused by the receipt of data, or by an error. If it was an error, a routine called vkbderr is called to handle it. If a data byte was received, the byte is checked to see if it was a keycode, or an IKBD packet header. If it's a keycode, ikdbsys processes the code, and places the character information into an input buffer, the location of which can be found with the XBIOS Iorec( ) call. If the byte was a header from a mouse, clock, status, or joystick packet, however, the main interrupt routine routes execution through one of the four vectors set up to handle these packet types (mousevec, clockvec, statvec, or joyvec).

Of these four, the mousevec and clockvec vectors are used by the system, and should generally be left alone (particularly if you want your mouse and clock functions to continue). The statvec and joyvec vectors are not used by the system, however, and you may want to install you own handlers for these functions. For example, in order to use joysticks with your program, you must send a command to the IKBD to return joystick information packets, then install your joystick packet handler to process these packets. If you do install your own handler, remember at the point that it is entered, the address of the packet buffer will be on the stack and in register A0.

Your routine should begin by saving all registers that you will use, and restoring those registers before ending. It should not spend more than one millisecond handling the interrupt (most of the time, just moving the packet information to your own buffer), and should end with an RTS instruction. Remember also that if you replace one of the vectors used by the system (like mousevec, for instance), you must either duplicate its actions in your own handler, or lose system-level functions (like Line A and GEM mouse support). And always save the default vectors, so you can replace them before your program ends.

IKBD Mode and Parameter Setting Commands

The IKBD supports a number of commands for using the mouse, clock, and joystick in various modes. These settings can changed by sending command strings to the IKBD with the Bconin( ), Ikbdws( ), or Initmous( ) functions. A summary of the command strings and their functions can be found below.

$07 Set Mouse Button Action

Determines whether the IKBD treats the mouse buttons as keyboard keys. In absolute positioning mode, this function can also be used to cause a mouse button press or release to return the absolute mouse position.

Byte Number

Byte Number	*Description*
1	The lower three bits of this byte are used as a flag. Possible values are: 1 = In absolute positioning mode, mouse button press causes mouse position report (see mouse interrogate command $0D). 2 = In absolute positioning mode, mouse button release causes mouse position report. 4 = Mouse buttons are treated like keyboard keys (left button returns keycode $74, right button returns keycode $75). Presumed to be the case when in mouse keycode mode.

$08 Set Relative Mouse Position Reporting

This command puts the mouse into relative position mode. In this mode, the IKBD sends mouse position packets whenever the mouse is moved more than the threshold distance in the *x* or *y* direction. The threshold distance is set with command $0B.

Input parameters

None

Packet returned

Byte Number	*Description*
1	$F8–$FB (mouse relative position packet header). Bits 0 and 1 record the right and left mouse button state: $F8 = neither button pressed $F9 = left button only pressed $FA = right button only pressed $FB = both buttons pressed
2	Change in horizontal position expressed as signed integer
3	Change in vertical position expressed as signed integer

$09 Set Absolute Mouse Positioning

This command puts the mouse into absolute position mode. In this mode, a mouse position packet that returns the current mouse position as an absolute *x,y* coordinate is generated whenever command $0D is issued. Depending on the mouse button action setting (command $07), this packet may also be generated by a press or release of either mouse button. In absolute position mode, set a maximum *x* and *y* position. Movement beyond the maximum position, or below 0, is ignored.

Setting absolute position mode resets the current *x,y* mouse position to 0,0. Another *x,y* position can be set using command $0E. In this mode, you may set a scaling factor that determines how many mouse units must be traversed before the *x* or *y* position changes. This scaling factor is set with command $0C.

Input parameters

Byte Number	*Description*
1	High byte, maximum *x* position (in scaled units).
2	Low byte, maximum *x* position.
3	High byte, maximum *y* position (in scaled units).
4	Low byte, maximum *y* position.

$0A Set Mouse Keycode Mode

This command puts the mouse into keycode mode. In this mode, the IKBD sends cursor arrow keycodes whenever the mouse is moved more than the threshold distance in the *x* or *y* direction. The make code is followed immediately by the break code. The mouse buttons are also treated like keyboard keys. The left button generates a keycode of $74, and the right button generates a keycode of $75.

Input parameters

Byte Number	*Description*
1	Horizontal distance (in mouse units) that must be traveled before the cursor left or cursor right code is sent.
2	Vertical distance (in mouse units) that must be traveled before the cursor up or cursor down code is sent.

$0B Set Mouse Threshold

This command sets the movement threshold necessary before a mouse position report is sent in relative position mode. Note that this command only affects relative position mode, and that the position data will not be rounded to the nearest multiple of the threshold.

Input parameters

Byte Number	*Description*
1	Horizontal distance (in mouse units) that must be traveled before the mouse position packet is sent.
2	Vertical distance (in mouse units) that must be traveled before the mouse position packet is sent.

$0C Set Mouse Scale

This command sets the scale factor for absolute position mode. This is the number of mouse units you must move before the internal *x* or *y* position settings change.

Input parameters

Byte Number	*Description*
1	Horizontal distance (in mouse units) that must be traveled before the mouse *x* position changes.
2	Vertical distance (in mouse units) that must be traveled before the mouse *y* position changes.

$0D Interrogate Mouse Position

This function is used to read the mouse position when in absolute position mode. The only other way to get the mouse position from that mode is to set the mouse buttons to report on a press or release, using command $07.

Input parameters
None

Packet returned

Byte Number	*Description*
1	$F7 (mouse absolute position packet header)
2	Button change status: Bit 0 = 1 Right button pressed since last read Bit 1 = 1 Right button released since last read Bit 2 = 1 Left button pressed since last read Bit 3 = 1 Left button released since last read
3	High byte of horizontal position
4	Low byte of horizontal position
5	High byte of vertical position
7	Low byte of vertical position

$0E Load Mouse Position

In absolute position mode, this command is used to reset the absolute *x* and *y* position values.

Input parameters

Byte Number	*Description*
1	A filler byte, must be 0.
2	High byte, *x* position (in scaled coordinate units).
3	Low byte, *x* position.
4	High byte, *y* position (in scaled coordinate units).
5	Low byte, y position.

$0F Set *y* Origin at Bottom

This command sets the origin of the *y* coordinate system at the bottom, which means that downward movement (towards the user) is treated as negative, and upward movement (away from the user) is treated as positive.

Input parameters
None

$10 Set *y* Origin at Top

This command sets the origin of the *y* coordinate system at the bottom, which means that downward movement (towards the user) is treated as positive, and upward movement (away from the user) is treated as negative.

Input parameters

None

$11 Resume

This command causes the IKBD to resume sending data packets after it has been paused with command $13. Since every other command also causes the IKBD to resume, and since this command is ignored if the IKBD isn't paused, it can be regarded as a No Operation (NOP) command.

Input parameters
None

$12 Disable Mouse

This command disables all mouse movement and mouse button scanning and reporting. The mouse may be reenabled by setting the mouse reporting mode with command $08, $09, or $0A.

Input parameters

None

$13 Pause Output

This command stops all transfer of data to the main processor. Scanning continues internally, and keystrokes, mouse movements, and joystick events (if enabled) will be queued up to capacity of the 6301 microprocessor's small buffer. In practical terms, if reporting is paused for more than a very small time period, these input events will be lost.

Input parameters

None

$14 Set Joystick Event Reporting

This command puts the joystick ports into event reporting mode. In this mode, each movement of the stick or button causes a joystick event packet to be sent.

Input parameters

None

Packet returned

Byte Number	*Description*
1	$FE–$FF (Joystick event packet header) $FE = Joystick 0 $FF = Joystick 1
2	Stick and button status byte (BxxxRLDU) Bit 0 = 1 Joystick pressed in up direction Bit 1 = 1 Joystick pressed in down direction Bit 2 = 1 Joystick pressed in left direction Bit 3 = 1 Joystick pressed in right direction Bit 7 = 1 Fire button pressed

$15 Set Joystick Interrogation Mode

This command puts the joystick ports into interrogation mode. In this mode, joystick events will only be reported when the interrogate joystick command ($16) is sent to the IKBD.

Input parameters

None

$16 Interrogate Joystick

This function can be used to read the current status of the joysticks when in event reporting mode or interrogation mode. It is the only way to read the joysticks when in interrogation mode.

Input parameters

None

Packet returned

Byte Number	*Description*	
1	$FD (Joystick report packet header)	
2	Joystick 0 stick and button status byte (BxxxRLDU)	
	Bit 0 = 1	Joystick pressed in up direction
	Bit 1 = 1	Joystick pressed in down direction
	Bit 2 = 1	Joystick pressed in left direction
	Bit 3 = 1	Joystick pressed in right direction
	Bit 7 = 1	Fire button pressed
3	Joystick 1 stick and button status byte (BxxxRLDU)	
	Bit 0 = 1	Joystick pressed in up direction
	Bit 1 = 1	Joystick pressed in down direction
	Bit 2 = 1	Joystick pressed in left direction
	Bit 3 = 1	Joystick pressed in right direction
	Bit 7 = 1	Fire button pressed

$17 Set Joystick Monitoring

This command sets the IKBD into joystick monitoring mode. When in this mode, the IKBD does nothing but monitor the joystick, communicate over the serial line, and maintain the time-of-day clock. The rate at which the joystick is sampled may be controlled down to 1/100 of a second.

Input parameters

Byte Number	*Description*
1	Time between joystick samples (in units of 1/100 of a second).

Packet returned

Byte Number	*Description*	
1	Fire button status for joystick 0 and 1(xxxxxxBB)	
	Bit 0 = 1	Fire button pressed on joystick 1
	Bit 1 = 1	Fire button pressed on joystick 0
3	Joystick status for joystick 0 and 1 (RLDURLDU)	
	Bit 0 = 1	Joystick 1 pressed in up direction
	Bit 1 = 1	Joystick 1 pressed in down direction
	Bit 2 = 1	Joystick 1 pressed in left direction
	Bit 3 = 1	Joystick 1 pressed in right direction
	Bit 4 = 1	Joystick 0 pressed in up direction
	Bit 5 = 1	Joystick 0 pressed in down direction
	Bit 6 = 1	Joystick 0 pressed in left direction
	Bit 7 = 1	Joystick 0 pressed in right direction

$18 Set Fire Button Monitoring

This command sets the IKBD into fire button monitoring mode. When in this mode, the IKBD does nothing but monitor the fire button on joystick 1, communicate over the serial line, and maintain the time-of-day clock. Eight samples are taken in the time it takes to send a single byte of data over the serial line. The results of the eight samples are packed into a single byte, with bit 7 holding the result of the most recent sample.

Input parameters

None

Packet returned

Byte Number	*Description*	
1	Fire button status for last eight reads of joystick 1	
	Bit 0 = 1	Fire button pressed during first sample
	Bit 1 = 1	Fire button pressed during second sample
	Bit 2 = 1	Fire button pressed during third sample
	Bit 3 = 1	Fire button pressed during fourth sample
	Bit 4 = 1	Fire button pressed during fifth sample
	Bit 5 = 1	Fire button pressed during sixth sample
	Bit 6 = 1	Fire button pressed during seventh sample
	Bit 7 = 1	Fire button pressed during last sample

$19 Set Joystick Keycode Mode

This command puts the joystick ports into keycode mode. In this mode, the IKBD sends cursor arrow keycodes whenever joystick 0 is moved from the center position. The make code is followed immediately by the break code. This mode has a velocity feature that allows you send the keycodes much faster if the stick has been held in one direction for longer than the breakpoint time duration. Until the breakpoint is reached, the keycodes repeat at a slower rate, but after the breakpoint is reached, they repeat at a faster rate.

Input parameters

Byte Number	*Description*
1	RX: The length of time until the horizontal velocity breakpoint is reached, and the key repeats become faster (in tenths of a second). By setting this to 0, the horizontal velocity feature is disabled, and VX is always used as the time between repeats.
2	RY: The length of time until the vertical velocity breakpoint is reached, and the key repeats become faster (in tenths of a second). By setting this to 0, the vertical velocity feature is disabled, and VY is always used as the time between repeats.
3	TX: The length of time between horizontal key repeats before the horizontal breakpoint is reached (in tenths of a second).
4	TY: The length of time between vertical key repeats before the vertical breakpoint is reached (in tenths of a second).
3	VX: The length of time between horizontal key repeats after the horizontal breakpoint is reached (in tenths of a second).
3	VX: The length of time between vertical key repeats after the horizontal breakpoint is reached (in tenths of a second).

$1A Disable Joysticks

This command disables all joystick movement and joystick button scanning and reporting. The mouse may be reenabled by setting the joystick reporting mode with command $14, $15, $17, $18, or $19.

Input parameters

None

$1B Set Time-of-Day Clock

This command sets the date and time for the internal time-of-day clock. The values are passed in packed Binary Coded Decimal (BCD) format. This means that each nibble contains a number from 0–9, so that each byte yields a two-digit decimal number. Any digit that isn't in BCD format is treated as a flag indicating that the current value shouldn't be changed.

Input parameters

Byte Number	*Description*
1	Year (last two digits only)
2	Month
3	Day
4	Hours (0–24)
5	Minutes
6	Seconds

$1C Interrogate Time-of-Day Clock

This command causes the IKBD to send a packet which gives information about the current time-of-day clock settings. The information that's returned is stored in packed BCD format.

Input parameters

None

Packet returned

Byte Number	*Description*
1	$FC (time of day packet header)
2	Year (last two digits only)
3	Month
4	Day
5	Hours (0–24)
6	Minutes
7	Seconds

$20 Memory Load

This command allows you to load arbitrary values into the 128 bytes of processor RAM which is found at locations $80–$FF on the 6301 processor chip. There's not much you can accomplish by doing it, however, since the processor uses all of the RAM for its own functions.

Input parameters

Byte Number	*Description*
1	High byte of address in controller memory to load (0)
2	Low byte of address in controller memory to load
3	Number of bytes to load (0–128)
4–*n*	Data bytes to load

$21 Memory Read

This command allows you to read the 6301 processor ROM or RAM, six bytes at a time.

Input parameters

Byte Number	*Description*
1	High byte of address in controller memory to read
2	Low byte of address in controller memory to read

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	$20 The number of the corresponding set command
3–8	Six bytes of 6301 memory, starting at the address specified in the command

$22 Controller Execute

This command allows you to execute a 6301 processor subroutine from a certain point in processor memory.

Input parameters

Byte Number	*Description*
1	High byte of subroutine address in controller memory
2	Low byte of subroutine address in controller memory

$80 Reset

This command performs a simple self-test, and returns the keyboard controller to its default mode and parameter settings. It does not affect the clock settings.

Input parameters

Byte Number	*Description*
1	A value of 1. Any other value causes the reset command to be ignored.

IKBD Status Commands

Many of the IKBD setting commands have corresponding status inquiry commands. These commands cause the IKBD to send an information packet that tells about the current setting, rather than changing it. Their command numbers are formed by adding $80 to the original setting command number. The returned responses are designed to imitate the format of the original setting commands, so if the status packet header byte ($F6) at the beginning is stripped away, the rest of the packet may be sent back to the IKBD as a command which restores the original setting. All status reports are padded to eight bytes in length, but the 0 value that is used for padding is ignored by the IKBD when sent as a command. In order to receive IKBD status packets on the ST, you must replace the statvec vector with a routine of your own that transfers the packets to your own buffer. A summary of the IKBD status commands are shown below.

$87 Inquire Mouse Button Action

This command causes the IKBD to send a packet which gives information about whether the mouse buttons are treated like keyboard keys, and if, in absolute position mode, a button press or release will cause a mouse event packet to be sent.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header).
2	$07 The number of the corresponding set command.
3	The lower three bits of this byte are used as a flag. Possible values are: 1 = In absolute positioning mode, mouse button press causes mouse position report. 2 = In absolute positioning mode, mouse button release causes mouse position report. 4 = Mouse buttons are treated like keyboard keys (left button returns keycode $74, right button returns keycode $75). Presumed to be the case when in mouse keycode mode.
4–8	Pad bytes (all zeros).

$88, $89, $8A Inquire Mouse Mode

This command causes the IKBD to send a packet which contains information about the mouse mode, and settings relevant to that mode.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	Mode number (corresponds to mode set command) $08 = Relative mode $09 = Absolute mode $0A = Keycode mode
3	Relative mode = 0 Absolute mode = High byte, maximum *x* position (in scaled units) Keycode mode = Horizontal distance (in mouse units) that must be traveled before the cursor left or cursor right code is sent.
4	Relative mode = 0 Absolute mode = Low byte, maximum *x* position Keycode mode = Vertical distance (in mouse units) that must be traveled before the cursor up or cursor down code is sent.
5	Relative mode = 0 Absolute mode = High byte, maximum *y* position (in scaled units) Keycode mode = 0
6	Relative mode = 0 Absolute mode = Low byte, maximum *y* position Keycode mode = 0
7–8	Pad bytes (0)

$8B Inquire Mouse Threshold

This command causes the IKBD to send a packet which contains information about the threshold values used for relative position mode.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	$0B The number of the corresponding set command
3	Horizontal distance (in mouse units) that must be traveled before the mouse position packet is sent
4	Vertical distance (in mouse units) that must be traveled before the mouse position packet is sent.
5–8	Pad bytes (0)

$8C Inquire Mouse Scale

This command causes the IKBD to send a packet which contains information about the scale factor used for absolution position mode.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	$0C The number of the corresponding set command
3	Horizontal distance (in mouse units) that must be traveled before the mouse *x* position changes.
4	Vertical distance (in mouse units) that must be traveled before the mouse *y* position changes.
5–8	Pad bytes (0)

$8F, $90 Inquire Mouse *y* Origin

This command causes the IKBD to send a packet which contains information about whether the *y* origin is set at the bottom position (in which the mouse is closest to the user) or the top position (in which the mouse is farthest away from the user).

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	The number of the corresponding set *y* origin command $0F = Origin at bottom $10 = Origin at top
3–8	Pad bytes (0)

$92 Inquire Mouse Enabled/Disabled

This command causes the IKBD to send a packet which contains information about whether mouse scanning and reporting is enabled or disabled.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	Number of the corresponding mouse status command $00 = Enabled $12 = Disabled
3–8	Pad bytes (0)

$94, $95, $99 Inquire Joystick Mode

This command causes the IKBD to send a packet that contains information about the current joystick mode. If keycode mode is being used, it also returns information about the velocity cursor settings.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	The number of the corresponding set mode command $14 = Event reporting mode $15 = Interrogation mode $19 = Keycode mode
3–8	For event reporting and interrogation modes these are pad bytes (all 0s). For keycode mode, they are:
3	RX: The length of time until the horizontal velocity breakpoint is reached, and the key repeats become faster (in tenths of a second).
4	RY: The length of time until the vertical velocity breakpoint is reached, and the key repeats become faster (in tenths of a second).
5	TX: The length of time between horizontal key repeats before the horizontal breakpoint is reached (in tenths of a second).
6	TY: The length of time between vertical key repeats before the vertical breakpoint is reached (in tenths of a second).
7	VX: The length of time between horizontal key repeats after the horizontal breakpoint is reached (in tenths of a second).
8	VX: The length of time between vertical key repeats after the horizontal breakpoint is reached (in tenths of a second).

$9A Inquire Joystick Enabled/Disabled

Input parameters

This command causes the IKBD to send a packet that contains information about whether mouse scanning and reporting is enabled or disabled.

Packet returned

Byte Number	*Description*
1	$F6 (status packet header)
2	Number of the corresponding joystick enable command $00 = Enabled $1A = Disabled
3–8	Pad bytes (0)

Appendix J

Keycodes

The GEMDOS function Cconin( ), and the BIOS function Bconin( ) (when used to read the console device keyboard), both return a two-byte keycode, which is extended into a longword. The first byte of the keycode, which is found in the second byte of the longword, is usually a unique key identifier that refers to the physical key that was struck, regardless of shift-key combinations. The second byte, which is found in the last byte of the longword, is usually the ASCII value of the key combination, which does depend on the state of the shift keys (Shift, Control, and Alternate). The following table shows the keycodes, as four-digit hexadecimal numbers, for all key and shift combinations.

	Unshifted		Shift	CTRL	ALT
a	1E61	A	1E41	1E01	1E00
b	3062	B	3042	3002	3000
c	2E63	C	2E43	2E03	2E00
d	2064	D	2044	2004	2000
e	1265	E	1245	1205	1200
f	2166	F	2146	2106	2100
g	2267	G	2247	2207	2200
h	2368	H	2348	2308	2300
i	1769	I	1749	1709	1700
j	246A	J	244A	240A	2400
k	256B	K	254B	250B	2500
l	266C	L	264C	260C	2600
m	326D	M	324D	320D	3200
n	316E	N	314E	310E	3100
o	186F	O	184F	180F	1800
p	1970	P	1950	1910	1900
q	1071	Q	1051	1011	1000
r	1372	R	1352	1312	1300
s	1F73	S	1F53	1F13	1F00
t	1474	T	1454	1414	1400

Unshifted		Shift		CTRL	ALT
u	1675	U	1655	1615	1600
v	2F76	V	2F56	2F16	2F00
w	1177	W	1157	1117	1100
x	2D78	X	2D58	2D18	2D00
y	1579	Y	1559	1519	1500
z	2C7A	Z	2C5A	2C1A	2C00
1	0231	!	0221	0211	7800
2	0332	@	0340	0300	7900
3	0433	#	0423	0413	7A00
4	0534	$	0524	0514	7B00
5	0635	%	0625	0615	7C00
6	0736	^	075E	071E	7D00
7	0837	&	0826	0817	7E00
8	0938	*	092A	0918	7F00
9	0A39	(	0A28	0A19	8000
0	0B30	)	0B29	0B10	8100
-	0C2D	_	0C5F	0C1F	8200
=	0D3D	+	0D2B	0D1D	8300
‘	2960	ˊ	297E	2900	2960
\	2B5C		2B7C	2B1C	2B5C
[	1A5B	{	1A7B	1A1B	1A5B
]	1B5D	}	1B7D	1B1D	1B5D
;	273B	:	273A	271B	273B
'	2827	"	2822	2807	2827
,	332C	<	333C	330C	332C
.	342E	>	343E	340E	342E
/	352F	?	353F	350F	352F
Space	3920		3920	3900	3920
Esc	011B		011B	011B	011B
Backspace	0E08		0E08	0E08	0E08
Delete	537F		537F	531F	537F
Return	1C0D		1C0D	1C0A	1C0D
Tab	0F09		0F09	0F09	0F09

Cursor Pad

Unshifted	Shift	CTRL	ALT	
Help	6200	6200	6200	(screen print)
Undo	6100	6100	6100	6100
Insert	5200	5230	5200	(left mouse button)
Clr/Home	4700	4737	7700	(right mouse button)

Unshifted	Shift	CTRL	ALT	
Up-Arrow	4800	4838	4800	(move mouse up)
Dn-Arrow	5000	5032	5000	(move mouse down)
Lft-Arrow	4B00	4B34	7300	(move mouse left)
Rt-Arrow	4D00	4D36	7400	(move mouse right)

Numeric Pad

Unshifted	Shift	CTRL	ALT	
(	6328	6328	6308	6328
)	6429	6429	6409	6429
/	652F	652F	650F	652F
*	662A	662A	660A	662A
–	4A2D	4A2D	4A1F	4A2D
+	4E2B	4E2B	4E0B	4E2B
.	712E	712E	710E	712E
Enter	720D	720D	720A	720D
0	7030	7030	7010	7030
1	6D31	6D31	6D11	6D31
2	6E32	6E32	6E00	6E32
3	6F33	6F33	6F13	6F33
4	6A34	6A34	6A14	6A34
5	6B35	6B35	6B15	6B35
6	6C36	6C36	6C1E	6C36
7	6737	6737	6717	6737
8	6838	6838	6818	6838
9	6939	6939	6919	6939

Function Keys

Unshifted	Shift	CTRL	ALT	
F1	3B00	5400	3B00	3B00
F2	3C00	5500	3C00	3C00
F3	3D00	5600	3D00	3D00
F4	3E00	5700	3E00	3E00
F5	3F00	5800	3F00	3F00
F6	4000	5900	4000	4000
F7	4100	5A00	4100	4100
F8	4200	5B00	4200	4200
F9	4300	5C00	4300	4300
F10	4400	5D00	4400	4400

Appendix K

ST Memory Map

In order to maintain upward compatibility, programmers are cautioned to use only documented system variables, and to avoid writing directly to the hardware. All RAM below location 2048 ($800) is reserved for use by the system. This is *protected memory,* and may only be accessed from supervisor mode. Attempts to read or write to these locations from user mode will cause a bus error (two mushrooms).

Table K-1. 68000 Exception Vectors

Vector Number	Address	Vector
0	$000	Initial Supervisor Stack Pointer on Reset
1	$004	Initial Program Counter on Reset
2	$008	Bus Error
3	$00C	Address Error
4	$010	Illegal Instruction Error
5	$014	Divide by 0
6	$018	CHK Instruction
7	$01C	TRAPV Instruction
8	$020	Privilege Violation
9	$024	Trace
10	$028	Opcode 1010 emulation (Line A routines
11	$02C	Opcode 1111 emulation (Line F--used by AES)
12–23	$030–$05C	Reserved by Motorola
24	$060	Spurious Interrupt

Table K-2. Auto-Vector Interrupts

Vector Number	Address	Vector
25	$064	Level 1 Interrupt: (used if Hblank enabled)
26	$068	Level 2 Interrupt: Horizontal blank sync (Hblank)
27	$06C	Level 3 Interrupt: Normal processor interrupt level

Table K-2. Auto-Vector Interrupts

Vector Number	Address	Vector
28	$070	Level 4 Interrupt: Vertical blanking sync (Vblank)
29	$074	Level 5 Interrupt
30	$078	Level 6 Interrupt: MK68901 MFP chip interrupts
31	$07C	Level 7 Interrupt: Nonmaskable interrupt

Table K-3. TRAP Instruction Vectors

Vector Number	Address	Vector
32	$080	TRAP #0 (unused by system)
33	$084	TRAP #1 (GEMDOS calls)
34	$088	TRAP #2 (used by GEMDOS)
35–44	$08C–$0B0	TRAPs #3–12 (unused by system)
45	$0B4	TRAP #13 (BIOS calls)
46	$0B8	TRAP #14 (XBIOS calls)
47	$0BC	TRAP #15 (unused by system)
48–63	$0C0–$0FC	Reserved by Motorola

Table K-4. User Interrupt Vectors (MFP 68901)

Vector Number	Address	Vector
64	$100	Parallel Port (int 0)
65	$104	RS-232 Carrier Detect (int 1)
66	$108	RS-232 Clear to Send (int 2)
67	$10C	Graphics blit done (int 3)
68	$110	RS-232 baud rate generator (Timer D)
69	$114	200 Hz system clock (Timer C)
70	$118	Keyboard/MIDI (6850 processor) (int 4)
71	$11C	Polled Floppy Disk Controller/Hard Disk Controller (int 5)
72	$120	Horizontal Blank counter (Timer B)
73	$124	RS-232 transmit error interrupt
74	$128	RS-232 transmit buffer empty interrupt
75	$12C	RS-232 receive error interrupt
76	$130	RS-232 receive buffer full interrupt
77	$134	User/application (Timer A)
78	$138	RS-232 ring indicator (int 6)
79	$13C	Polled monochrome monitor detect (int 7)

Processor State Save Area

Exception vectors 2, 3, 4, 6, 7, and 8 all point to a handler that saves information about the processor state, and displays a number of mushroom clouds equal to the exception number. TOS uses the following portion of the User Interrupt Vector space to save the processor state information when a crash occurs. This memory area is not cleared by a system reset, but may be overwritten by subsequent crashes.

proc_lives

896–899 **($380–$383)**
If the system was able to save the processor state, it sets this variable to $12345678 as a flag that the following information is valid.

proc_dregs

900–931 **($384–$3A3)**
The contents of the eight data registers are saved here, starting with D0 and ending with D7.

proc_aregs

932–963 **($3A4–$3C3)**
The contents of the eight address registers are saved here, starting with A0 and ending with A7. Note that A7 represents the Supervisor Stack Pointer, not the User Stack Pointer, which is saved in proc_usp, below.

proc_pc

964–967 **($3C4–$3C7)**
The first byte of this longword is the exception number that caused the crash.

proc_usp

968–967 **($3C8–$3CB)**
The User Stack Pointer is saved here.

proc_stk

972–1003 **($3CC–$3DB)**
The top 16 words of the supervisor stack are saved here.

Logical Vectors

These vectors are not guaranteed to appear at this location, so to change them, use the BIOS function Setexec( ), with an exception number of $100–$107.

etv_timer

1024–1027 **($400–$403)**

Timer tick handoff vector (logical vector $100). The system timer interrupt handler (called every 20 milliseconds) is used to maintain the GEMDOS time-of-day clock, drive the XBIOS background sound routine, handle key repeat, and perform other housekeeping chores. At the end of every timer tick, the interrupt handler also performs a JSR through this vector. To install a new handler routine, use Setexec( ), and save the old vector address. When the handler is entered, the timer tick rate (in milliseconds) is on the stack. The handler should execute its own code first, and exit by jumping through the old vector. Since the TOS portion of the interrupt handler saves all registers before calling this routine, and restores them afterwards, there is no need for the user-installed routine to save them as well.

etv_critic

1028–1031 **($404–$407)**

Critical error handler (logical vector $101). This vector is called by the BIOS when certain errors (such as Rwabs( ) disk errors or media changes) occur. This allows an application to take care of its own error handling. When execution passes to the handler at this address, the first word on the stack, 4(sp), is an error number, and other parameters may follow it, depending on the error. The handler should preserve registers D3–D7/A3–A6, and when it returns, it should place a longword error code in D0:

Error Code	Action
0x00010000	Retry
0x00000000	Ignore the error
0xFFFFFFxx	Abort with error

The default handler merely returns a −1L.

etv_term

1032–1035 **($408–$40B)**

Process terminate handler (logical vector $102). GEMDOS calls this vector right before it terminates a process. Normally, this vector points to an RTS instruction, which allows the process to terminate. If the application doesn't want to be terminated (because of an accidental CTRL-C during Cconrs( ), for example), it can regain control here. See also the criticret vector (1162, $48A).

etv_xtra

1036–1055 **($40C–$41F)**

Reserved for up to five future logical vectors ($103–$107).

System Variables

The following system variable locations are the only ones that Atari guarantees not to change between TOS versions. Any undocumented variables, ROM routines or exception vectors are almost certain to change, and should not be relied upon.

memvalid

1056–1059 **($420–$423)**

If this location contains the magic number $752019F3, and the magic number in memval2 (1082, $43A) is also correct, then the coldstart was successful, and the memory configuration in memcntlr (below) is correct.

memcntlr

1060–1061 **($424–$425)**

The second nybble of this word contains the configuration value for the memory controller. Some possible values include:

Value of Nybble	**Memory Configuration**
0	128K (or 256K, 2 banks)
4	512K
5	1Mb (2 banks)

resvalid

1062–1065 **($426–$429)**

If this location contains the magic number $31415926, then on system reset, execution will be directed through resvector below.

resvector

1066–1069 **($42A–$42D)**

The system will jump through this vector very early in the reset process if resvalid is set to its magic number.

phystop

1070–1073 **($42E–$431)**

This location point to the physical top of RAM, the first unusable byte at the high end of the memory space (for instance, $80000 for a 512K machine).

_membot

1074–1077 **($432–$435)**
A pointer to the lowest available memory location. The BIOS Getmpb( ) function uses this value as the start of the GEMDOS Transient Program Area (TPA).

_memtop

1078–1081 **($436–$439)**
A pointer to the highest available memory location. The BIOS Getmpb( ) function uses this value as the end of the GEMDOS Transient Program Area (TPA).

memval2

1082–1085 **($43A–$43D)**
If this location contains the magic number $237698AA, and the magic number in memvalid (1056, $420) is also correct, then the coldstart was successful, and the memory configuration in memcntlr (below) is correct.

flock

1086–1087 **($43E–$43F)**
Floppy lock variable. If it contains a value other than zero, the vertical blank disk routine is disabled, which guarantees that it doesn't touch the DMA chip registers. This variable must be nonzero when the DMA bus is being used.

seekrate

1088–1089 **($440–$441)**
Bits 0 and 1 of this word contain the default seek rate (the time it takes for the head to move to the next track) for both floppy drives:

Value in Bits 0 and 1	Seek Rate (in Milliseconds)
0	6
1	12
2	2
3	3 (default)

_timr_ms

1090–1091 **($442–$443)**
The time between system timer ticks in milliseconds. For current STs, this value is 20 milliseconds, which corresponds to 50 timer updates per sec-

ond. This is the value that is returned by the BIOS function Tickcal( ). This value is also placed on the stack before the timer interrupt handler calls the timer handoff vector.

_fverify

1092–1093 **($444–$445)**

Floppy disk write verify flag. When the value stored here is nonzero (the default), all writes are verified. When zero, write verify is turned off.

_bootdev

1094–1095 **($446–$447)**

This location contains the number of the device from which the system was booted.

almode

1096–1097 **($448–$449)**

A 0 indicates that NTSC (60 Hz, U.S.) video output is used. A nonzero value indicates that the video output is PAL (50 Hz, European).

defshftmd

1098–1099 **($44A–$44B)**

The default video resolution. If the system is shifted from monochrome to color, the value here determines whether the color screen is brought up in low resolution (0), or medium resolution (1).

sshiftmd

1100–1101 **($44C–$44D)**

Contains a shadow copy of the shiftmd hardware register, which determines the screen display mode:

Value	Screen Display Mode
0	320 × 200, 16 colors (low resolution)
1	640 × 200, 4 colors (medium resolution)
2	640 × 400, 2 colors (high resolution)

_v_bas_ad

1102–1105 **($44E–$451)**

This location contains the starting address of screen RAM, a 32K memory area that always starts on a 512-byte boundary.

vblsem

1106–1107 **($452–$453)**
A semaphore for vertical blank interrupt processing. A value of 0 here disables vblank processing, while a value of 1 enables processing.

nvbls

1108–1109 **($454–$455)**
The number of deferred vertical blank interrupt handler vectors in the table pointed to by _vblqueue (see below). These are slots where applications can tack on their own vertical blank interrupt handlers. The default number of slots that TOS allocates is 8.

_vblqueue

1110–1113 **($456–$459)**
The vertical blank interrupt (VBI) queue. VBI processing starts at the point when the video display's electron beam has finished updating the last line of the screen display, and finishes before it starts updating the first line of the next screen frame. Depending on what kind of screen is used, this interrupt will occur every 1/50 second (European color monitor), every 1/60 second (U.S. color monitor), or every 1/70 second (monochrome monitor). A number of routine system functions are carried out by the default vertical blank interrupt handler. These include blinking the cursor, checking for a shift between monochrome and color monitors, and other video-related tasks which are best performed when the screen is not being updated.

Because the vertical blanking interval is the perfect time for making graphics changes, it is often desirable for applications to insert their handlers into the vertical blank interrupt handler chain. To this end, TOS supports the installation of *deferred* VBI handlers. TOS initially allocates a table that can hold the addresses of up to eight deferred VBI slots (the number is stored in nvbls, above). The address of this table is stored in this location, vblqueue. Just before the system interrupt handler routine returns, it checks the contents of each of these eight addresses in turn. If it finds a nonzero value in the first one, it assumes it to be the address of a deferred VBI handler, and performs a JSR to that address. When the first subroutine is completed, the system interrupt handler checks the next slot, and so on, until it encounters a value of zero in the vblqueue table.

In order to install your own interrupt handler, you must examine the entries in the table pointed to by vblqueue. You may install your VBI handler at the first table entry that contains a 0. Your handler may use any registers except the user stack pointer. It should return with an RTS instruction, not an RTE. Be sure to clear the address of your handler from the vblqueue table before your application closes.

The ST system is not limited to eight deferred VBI handlers. If the table pointed to by vblqueue is full, you may wish to allocate a larger table, copy the contents of the vblqueue table to it, and then update nvbls and vblqueue to reflect the new table size and location. Be sure to restore the old table before your application exits.

colorptr

1114–1117 **($45A–$45D)**

If this value is nonzero, then at the next vertical blank interrupt, the table of 16 color values that start at the address stored here are loaded into the 16 color registers. If this value is zero, the color palette is not changed.

screenpt

1118–1121 **($45E–$461)**

If this value is nonzero, it is assumed to be the starting address for a new 32K block of display memory, and the physical screen base address will be changed accordingly, during the next vertical blank interrupt.

_vbclock

1122–1125 **($462–$465)**

Counter for the number of vertical blank interrupts that have actually been processed (not blocked by vblsem).

_frclock

1126–1129 **($466–$469)**

Total count of vertical blank interrupts. This count is updated regardless of the status of vblsem.

hdv_init

1130–1133 **($46A–$46D)**

Vector to the hard disk initialization routine. A 0 indicates that no hard disk is installed.

swv_vec

1134–1137 **($46E–$471)**

Vector to the routine that the system uses when it detects a monitor change (from monochrome to color, or vice versa). This vector initially points to the reset handler, so the system resets if the user changes the monitor.

hdv_bpb

1138–1141 **($472–$475)**

The vector to the routine to use when the Getbpb( ) routine requests information about a hard disk. This routine is installed when the hard disk is initialized, and should follow the calling sequence of the normal BIOS Getbpb( ) function. A value of 0 is stored here if no hard disk is attached.

hdv_rw

1142–1145 **($476–$479)**

The vector to the routine to use when TOS functions wish to read or write to the hard disk. This routine is installed when the hard disk is initialized, and should follow the calling sequence of the normal BIOS Rwabs( ) function. A value of 0 is stored here if no hard disk is attached.

hdv_boot

1146–1149 **($47A–$47D)**

Vector to the routine used to boot from the hard disk. A value of 0 is stored here if no hard disk is attached.

hdv_mediach

1150–1153 **($47E–$481)**

The vector to the routine to use when the Mediachb( ) routine requests information about a hard disk's media change status. This routine is installed when the hard disk is initialized, and should follow the calling sequence of the normal BIOS Mediach( ) function. A value of 0 is stored here if no hard disk is attached.

_cmdload

1154–1155 **($482–$483)**

When this location contains a nonzero value, the system attempts to load and execute a program called COMMAND.PRG from the boot device. This makes it possible to load an application other than the GEM Desktop at boot time. Store a nonzero value here from the disk's boot code.

conterm

1156 **($484)**

This byte contains a number of flag bits that control various aspects of the console device's functioning:

Bit Set	Action
0	Enable key-click sound when key is pressed
1	Enable key repeat
2	Enable bell sound when a CTRL-G character (ASCII 7) is written to CON:
3	Cause the BIOS Bconin( ) function to return information about the shift key status from kbshift in bits 24–31

reserved

1157 **($485)**

trpl4ret

1158–1161 **($486–$489)**
Saved trap 14 return address.

criticret

1162–1165 **($48A–$48D)**
Saved return address for etv_critic vector.

themd

1166–1181 **($48E–$49D)**
The the MD (memory descriptor) structure that is initialized by the Getmpb( ) call. This structure may not be changed once GEMDOS is initialized.

_md

1182–1185 **($49E–$4A1)**
Pointer to additional memory descriptors.

savptr

1186–1189 **($4A2–$4A5)**
Pointer to the buffer the BIOS uses to save register values.

_nflops

1190–1191 **($4A6–$4A7)**
Number of floppy drives connected to the system (0, 1, or 2).

con_state

1192–1195 **($4A8–$4AB)**
Vector for console output routines, which can be set to point to various Esc functions.

save_row

1196–1197 **($4AC–$4AD)**
The row number is saved temporarily in this buffer when positioning the cursor with the VT-52 Esc-Y command.

sav_contxt

1198–1201 **($4AE–$4B1)**
Pointer to a temporary buffer where the processor context is saved.

_bufl

1202–1209 **($4B2–$4B9)**
Two pointers to GEMDOS buffer list headers. Each points to a Buffer Control Block (BCB), which is laid out as follows:

```
struct BCB
{
  BCB       *b_link;      /* ptr to next BCB */
  int       b_bufdrv;     /* drive number, or -1 */
  int       b_buftyp;     /* buffer type */
  int       b_bufrec;     /* record number in this buffer */
  int       b_dirty       /* dirty (buffer changed) flag */
  DMD       *b_dm         /* ptr to Drive Media Descriptor */
  char      *b_bufr       /* ptr to buffer */
```

The first buffer is used to store data sectors, the second is used to store FAT and directory sectors.

_hz_200

1210–1213 **($4BA–$4BD)**
Counter for 200 Hz timer. Divided by 4 for the 50 Hz system clock. This value is used as the starting seed the first time Random( ) is called.

the_env

1214–1217 **($4BE–$4C1)**
The default environment string, which consists of four 0 bytes.

_drvbits

1218–1221 **($4C2–$4C5)**
Indicates which drives are connected. Each of the 32 bits of the _drvbits variable corresponds to a different drive. Bit 0 is assigned to drive A:, bit 1 to drive B:, and so on up to bit 15, which corresponds to drive P: (the current version of the ST operating system only recognizes 16 drives). If the bit which corresponds to a drive is set to 1, that drive is connected, otherwise, it is unavailable. The value stored here is the same one returned by the BIOS function Drvmap( ). Note that if even one floppy is connected, bits 0 and 1 are both always set to one. That's because if drive A: is connected, the system will use it as a logical drive B: if no physical drive B: is present.

_dskbufp

1222–1225 **($4C6–$4C9)**

Pointer to a 1K disk buffer. This buffer is also used by some graphics operations, and should not be used by interrupt routines.

_autopath

1226–1229 **($4CA–$4CD)**

Pointer to autoexec path (or null).

_vbl_list

1230–1261 **($4CE–$4ED)**

This area is used for the default deferred VBI table, which is pointed to by vblqueue (1110, $456).

_prt_cnt

1262–1263 **($4EE–$4EF)**

This flag is used to let the system know when to start a screen dump, and when to abort it. It's initially set to −1, and incremented when the Alternate-Help key is pressed. The screen dump code starts sending information to the printer when it sees a value of zero here, and aborts the screen print when it sees a nonzero value.

_prtabt

1264–1265 **($4F0–$4F1)**

Flag to abort print operation due to time-out.

_sysbase

1266–1269 **($4F2–$4F5)**

Pointer to the start of the TOS ROMs. This value is needed because there is no guarantee that the current starting address will not change (for instance, the ROMs might get bigger).

_shell_p

1270–1273 **($4F6–$4F9)**

Pointer to shell-specific context.

end_os

1274–1275 **($4FA–$4FD)**
Points to the first byte past the low RAM area used by TOS. This is used for the starting address of the TPA (end_os is copied into _membot).

exec_os

1278–1281 **($4FE–$501)**
This points to the operating system command shell which is executed when system initialization is complete. Normally, this points to the first byte of the AES text segment.

scr_dump

1282–1285 **($502–$505)**
The Scrdump( ) routine is vectored through this location, so that when Scrdump( ) is called, or the Alternate-Help keys are pressed, program execution is directed to the routine whose address is found here. To install a printer driver for another printer, therefore, all you have to do is to load the new screen print program as a terminate-and-stay-resident program (see Ptermres( ), Chapter 5), and store its address here.

This vector can also be diverted for other purposes. Some "snapshot" programs, for example, use this vector to install a routine that saves the screen picture to a disk file when the Alternate-Help keys are pressed, rather than sending it to a printer. It's also possible to install a short routine that tests for shift keys when Alternate-Help is pressed, thus allowing additional hot-key programs to be installed, rather than just replacing the screen print function.

prv_lsto

1286–1289 **($506–$509)**
This vector is used by Prtblk( ) to call the PRN: device output status routine.

prv_lst

1290–1293 **($50A–$50D)**
This vector is used by Prtblk( ) to call the PRN: device output routine.

prv_auxo

1294–1297 **($50E–$511)**
This vector is used by Prtblk( ) to call the AUX: device output status routine.

prv_aux

1298–1301 **($512–$515)**
This vector is used by Prtblk( ) to call the AUX: device output routine.

Index by Function Name

Index by Function Name

Index by Function Name

Index

Index

$24.95
Higher in Canada

The Complete TOS Reference

COMPUTE!'s Technical Reference Guide, Atari ST—Volume Three: TOS includes all the information you'll need to work with GEMDOS, the BIOS, and the XBIOS. In addition, this book features a complete Atari ST memory map detailing hardware registers and important operating system variables.

With this book, noted author and programmer Sheldon Leemon completes COMPUTE!'s series of comprehensive reference guides to the inner workings of the Atari ST.

Here's a list of just some of the topics covered inside:

- The Basic Input/Output System (BIOS)
- The eXtended Basic Input/Output System (XBIOS)
- The GEM Disk Operating System (GEMDOS)
- Low-level graphics (Line A)
- Reference section explaining each TOS function
- Program examples in C and machine language
- Complete memory map

If you are an ST programmer, this is the reference you've been looking for. COMPUTE! Books remains the leading publisher of programs and information for the Atari ST. *COMPUTE!'s Technical Reference Guide—Atari ST, Volume Three: TOS* is yet another example of the high quality you've come to expect in any guide to personal computing from COMPUTE!.

COMPUTE! Books
Greensboro, North Carolina
Radnor, Pennsylvania

ISBN 0-87455-149-8
90000
9 780874 551495