– CHAPTER 4 –

XBIOS

Overview

The e**X**tended **B**asic **I**nput/**O**utput **S**ystem (**XBIOS**) is a software sub-system of **TOS** which contains functions used to interact with and control Atari computer hardware. The availability of many of these functions is dependent on hardware whose presence can be determined by the current **TOS** version or by interrogating the system 'cookie jar' (see *Chapter 3:* ***BIOS*** for more details).

Some functions (notably video hardware and storage device related functions) should only be used by device drivers and system level software as they represent a non-portable method of hardware interaction which may be unsupported in future Atari computers.

As a general rule, **GEMDOS** and **VDI** functions should be used, when possible, rather than **XBIOS** calls. The **GEMDOS** and **VDI** provide a software abstraction layer which will make software applications much more compatible across new computer releases.

Video Control

The video capabilities of Atari computer systems have varied greatly since their introduction. Applications which use the **VDI** for their video displays will require little if any modifications to run on new systems. The **XBIOS** is mostly required for device drivers and other applications which require more direct control over the video hardware. When present, the '_VDO' entry in the system cookie jar will reveal information about the video hardware present.

The Physical/Logical Screen

Two separate video display pointers are maintained by the **XBIOS** at any time. The physical screen address points to the memory location that the video shifter uses to update the display. This memory must *not* be in fast RAM and must be **WORD**-aligned (original ST computers expect screen memory to be aligned to a 256-byte boundary).

A second video memory pointer points to the 'logical' screen. This memory area is used by the **VDI** to output graphics. Normally, the physical screen address is equal to the logical screen address meaning that **VDI** output is shown immediately on screen. Software (most commonly games) can allocate an additional memory block and use these two pointers to page-flip for smooth animations.

Physbase() and **Logbase()** return these two addresses. **Setscreen()** can be used to reset these addresses and change screen modes. As of **TOS** 4.0, **Setscreen()** reinitializes the **VDI** screen driver (you must still call **vq_extnd()** to update your workstations) but will *not* reinitialize the **AES**. This means that if you change resolution using **Setscreen()**, do not use the **AES** until the screen is restored to its original resolution. On **TOS** versions prior to 4.0, you should not use any **GEM** calls while the screen mode is altered.

The Falcon030 function **VgetSize()** is a utility function that will return the number of bytes that must be allocated for the specified video mode. When not running on a Falcon030, you will have to calculate this yourself.

Setting/Determining Screen Resolution

Getrez() was *originally* a safe method for determining the current video hardware configuration. As new video modes became available, though, **Getrez()** became less and less useful. Currently, **Getrez()** should be used for *only* one purpose. The formula **Getrez()** + 2 should be used to select the **VDI** physical device ID for the screen so that the proper screen fonts can be selected. See the description of **v_opnvwk()** for more details.

In order to provide true screen independence, you should use the values returned by the **VDI** call **v_opnvwk()** to determining the screen resolution your application is using. The **XBIOS** provides calls that will determine the current video mode but they are hardware dependent and will probably stop working as expected as new video hardware is released.

The **Getrez()** call can reliably determine the video mode of an ST, STe or Mega ST/e. Three calls have since been added to determine the video mode of the TT030 and Falcon030 computers.

EgetShift() and **EsetShift()** can be used to interrogate and set the TT030 video mode. **VsetMode()** can similarly be used to interrogate and set the Falcon030 video mode. The Falcon030 call **VgetMonitor()** can be used to determine the type of attached monitor and, therefore, the available video modes.

TT030 **TOS** also provides the calls **EsetGray()** and **EsetSmear()**. Together, these calls duplicate some of the functionally contained in **EsetShift()** but can be used individually as desired to configure the special gray-scale and smear modes present in the TT030.

EsetShift() and **VsetMode()** are designed to change the video modes of the TT030 and Falcon030 respectively, however, they do not reinitialize the **AES** or **VDI**. It is also possible to change TT030 and Falcon030 video modes using **Setscreen()**. TT030 modes are set by supplying the appropriate resolution code (see **Getrez()** for a list of resolution codes). Falcon030 modes are set by adding an extra parameter to the call with a special resolution code of 3. See the explanation for **Setscreen()** later in this chapter for details.

Manipulating the Palette

Prior to the introduction of the TT, **Setcolor()** and **Setpalette()** were used to set the 16 available palette entries. **Setpalette()** sets the entire palette at once whereas **Setcolor()** sets colors at an individual level and can also be used to interrogate palette entries.

The ST has 16 palette entries, each supporting any of 512 available colors. The ST specifies color in components of red, green, and blue. Intensity settings of 0–7 are valid for each color component. The following list contains the red, green, and blue values for the ST's default 16 color palette.

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

You might have noticed that these registers are not mapped the same as **VDI** color indexes. The **VDI** re-maps color requests to its own needs. For a list of these re-mappings, see the entry for **vr_trnfm()**. It is also possible to build a remapping table on the fly by plotting one pixel for each **VDI** pen on the screen and using the **VDI v_get_pixel()** call on each to return the **VDI** and hardware register index.

Each of the sixteen color registers is bitmapped into a **WORD** as follows (The first row indicates color, the second is bit significance):

```
xxxx xRRR xGGG xBBB
xxxx x321 x321 x321
```

The STe series expanded the color depth to four bits instead of three which expanded the number of available colors from 512 to 4096. This changed the layout of these color **WORD**s as follows:

```
xxxx RRRR GGGG BBBB
xxxx 1432 1432 1432
```

This odd bit layout allowed for backward compatibility to the ST series.

The TT030 supports an expanded palette of 256 entries in 16 banks containing any of 4096 colors. The first bank of colors is still supported by **Setcolor()** and **Setpalette()**, however to access the additional 240 colors, 4 additional palette support calls were added.

Esetpalette(), **Egetpalette()**, and **Esetcolor()** provide access to these colors in a similar manner to **Setpalette()** and **Setcolor()**. **Esetbank()** switches between the 16 available banks of colors in color modes that support less than 16 colors. You should note that the TT030 color calls returned the color **WORD**s to normal bit ordering as follows:

```
xxxx RRRR GGGG BBBB
xxxx 4321 4321 4321
```

When using the TT's special gray mode, the lower eight bits of each hardware register is used as a gray value from 0–255.

The Falcon030 computer gives up the TT030 calls in favor of a more portable method of setting the hardware palette (ST calls will remain as compatible as possible). **VsetRGB()** and **VgetRGB()** set color palette entries based on 24-bit true color values. The **XBIOS** will scale these values as appropriate for the screen mode.

Advanced Video

Vsync() halts all further processing by the application until a vertical blank interrupt occurs. This interrupt signals that the video display gun has reached the bottom of the display and is returning to the top. At this time, a brief period occurs where updates to the screen will not be immediately apparent to the user. This time is usually used to present flicker-free animation and redraws.

VsetSync() is used to enable external hardware video synchronization for devices such as GENLOCK's. Both the vertical and horizontal syncronizations may be set independent of each other with this call.

VsetMask() provides easy access to the Falcon030's overlay mode. This call allows you to specify bits which will be added or removed to future color definitions created with the **VDI** call **vs_color()**. When a GENLOCK hardware device is connected, pixels with their overlay bit cleared will be replaceable by the device with external video.

The Falcon030 Sound System

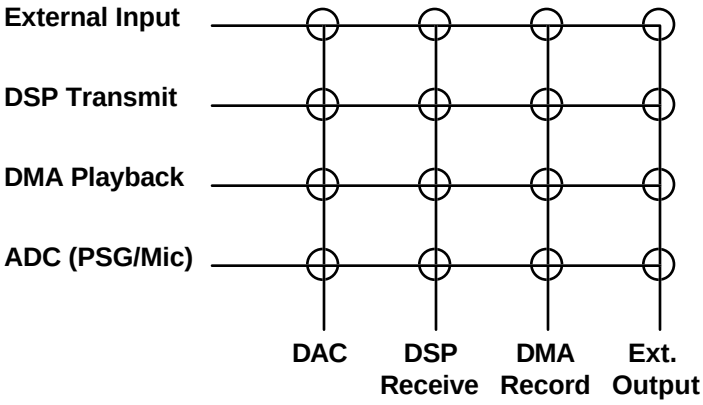
XBIOS sound system calls are only present as of the Falcon030 computer (though their presence should always be verified by the '_SND' cookie). If you want to program digitized audio that plays on an STe, TT, and Falcon030, see *Chapter 5: **Hardware***.

The Falcon030 sound system consists of four stereo 16-bit DMA playback and record channels[1], an onboard ADC (microphone jack), DAC (speaker and headphone jack), connection matrix, and digital signal processor.

When your application uses the sound system you should first lock it with **Locksnd()**. This ensures that other system processes don't try to access the sound system simultaneously. **Unlocksnd()** should be used as soon as the sound system is free.

[1]Only one output track may be monitored at a time, though the DSP may be programmed as a mixer to combine more tracks while sound is being output.

Each of four possible source devices can be connected to any or all of the four possible destination devices using the connection matrix as follows:



The external input and output are accessible with a specially designed hardware device connected to the DSP connector.

The Connection Matrix

The sound system call **Devconnect()** connects sound system components together. You must specify the source device, destination device(s), source clock, prescaler setting, and handshaking protocol.

The source clock can be set to either of two internal clocks (25.175 MHz and 32 MHz) or an external clock. The internal DMA sound routines are only compatible with the 25.175 MHz clock. Other clock sources are used in conjunction with external hardware devices.

The prescaler sets the actual sample playback and recording rate. A value of 0 will cause the sound system to use a STe/ TT030 compatible prescaler for outputting sound recorded at STe/TT030 frequencies. One STe/TT030 frequency, 6.258 kHz, is not supported on the Falcon030. You can set the STe/TT030 prescaler with the **Soundcmd()** call. Using values other than 0 will set the Falcon030 prescaler as documented under the **Devconnect()** call.

The last parameter you must pass to **Devconnect()** specifies whether to enable or disable hardware handshaking. Enabling handshaking will produce data that is 100% error free but will result in a variable transfer rate which may negatively affect digital sound. Handshaking is generally only enabled when the data being transferred must be transferred without errors (usually compressed audio or video data).

Recording/Playing Digital Audio

To record or playback an audio sample, use **Setbuffer()** to identify the location and length of your playback/recording buffer. Also, any **Devconnect()**, **Setmode()**, and **Soundcmd()** calls should be made prior to starting your playback/recording to set the sound hardware to the proper frequency and mode.

The Falcon030 *only* supports the recording of 16-bit stereo audio. To generate 8-bit samples you must scale the values in the buffer from **WORD**s to **BYTE**s after recording.

When processing either recording or playback through the DSP, the command **Dsptristate()** must be used to connect the DSP to the matrix.

You may use the function **Setinterrupt()**, as desired, to cause a MFP or Timer A interrupt at the end of every frame. This is most useful when you are playing or recording in repeat mode and you wish to use multiple buffers.

Buffptr() may be used to determine the current playback or record buffer pointer as sounds are being played/recorded.

Setmontracks() is used to define which track which will be output over the computer speaker/headphones. **Settracks()** controls which tracks will be used to record/playback data.

Configuring Levels

The function **Soundcmd()** has four modes which allow the setting and interrogation of the current levels of attenuation and gain. Gain affects input levels. The higher the value for gain, the louder the microphone input will be. Attenuation affects output levels. The higher the attenuation setting, the softer sounds will be output from the computer speaker/headphone jack.

Other Calls

Sndstatus() can be used to tell if a source clock rate was correctly set or if hardware clipping has occurred on either channel.

Gpio() is used to communicate data over the three general purpose pins of the DSP connector.

The DSP

The Falcon030 comes standard with a Motorola 56001 digital signal processor (DSP). Digital signal processors are useful for many different purposes such as audio/video compression, filtering, encryption, modulation, and math functions.

The DSP is able to support both programs and subroutines. Both must be written in 56001 assembly language (or a language which outputs 56001 object code). A full treatment of 56001 assembly language is beyond the scope of this document. Consult the *DSP56000/56001 Digital Signal Processor's User Manual* published by Motorola, Inc. for more information.

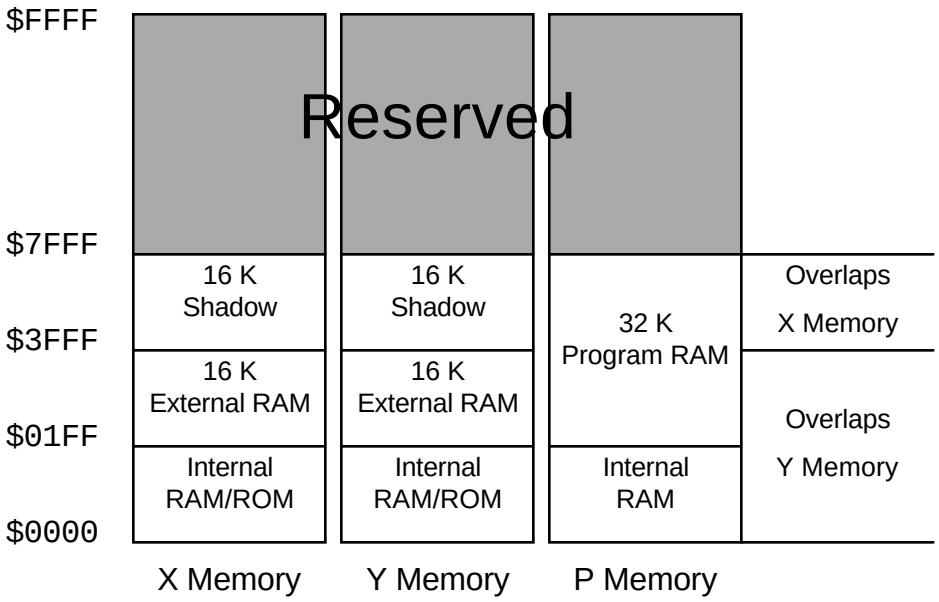
The DSP is capable of having many subroutines resident in memory, however, only one program may be loaded at any time.

When using the DSP you should call **Dsp_Lock()** to prevent other processes from modifying your setup and to ensure that you do not modify the work of other processes. Call **Dsp_Unlock()**

when done (the DSP's MR and IPR registers should have been returned to their original state) to release the DSP semaphore.

DSP Memory

The Falcon030's DSP contains 96K bytes of RAM for system programs, user programs, and subroutines. The DSP uses three distinct address spaces, X, Y, and P. Program memory (P) overlaps both X and Y memory spaces. Because of this, DSP programs should be careful when referencing memory. The following is a memory map of the DSP:



DSP Word Size

The 56001 uses a 24-bit **WORD**. Future Atari computers may use different DSP's with different **WORD** sizes. Use the **Dsp_GetWordSize()** call prior to using the DSP to determine the proper DSP **WORD** size.

DSP Subroutines

Subroutines are usually short programs (no longer than 1024 DSP **WORD**s) which transform incoming data. Each subroutine must be written to be fully relocatable. When writing subroutines, start instructions at location $0. All addresses in the subroutine must be relocatable based on the original PC of $0 in order to function. An alternative to this is to include a stub program at the start of your subroutine that performs a relocation based upon the start address assigned by the **XBIOS** (which is available in X:HRX at subroutine start).

Subroutines should store initialized data within its program space. The memory area from $3f00–$3fff is reserved for use as the BSS of subroutines. Subroutines must not rely on the BSS's data to remain constant between subroutine calls.

Each subroutine must be assigned a unique ability code either by using one predefined by Atari (none have been published yet) or by using the **Dsp_RequestUniqueAbility()** call. Since subroutines are only flushed from the DSP when necessary, an application may be able to use an existing subroutine with the same ability left by another application by using the **Dsp_InqrSubrAbility()** call.

Here is a sample of how to load a DSP subroutine with a non-unique ability code:

```
if(!DSP_Lock())
{
        ability = DSP_RequestUniqueAbility();
        handle = DSP_LoadSubroutine( subptr, length, ability );
        if(!handle)
        {
              DSP_FlushSubroutines();
              handle = DSP_LoadSubroutine( subptr, length, ability );
              if(!handle)
                   error(“Unable to load DSP subroutine”);
        }

        if(handle)
        {
              if(!Dsp_RunSubroutine( handle ))
                   DSP_DoBlock( data_in, size_in, data_out, size_out);
              else
                   error(“Unable to run DSP subroutine!”);
        }
}
```

DSP Programs

Only one DSP program may be resident in memory at once. Prior to loading a DSP program you should ensure enough memory is available for your program by calling **Dsp_Available()**. If not enough memory is available, you may have to flush resident subroutines to free enough memory.

After you have found that enough memory is available, you must reserve it with **Dsp_Reserve()**. This memory will be reserved until the next **Dsp_Reserve()** call so you should ensure that you have called **Dsp_Lock()** to block other processes from writing over your program.

Programs can be stored in either binary or ASCII (‘.LOD’) format. The function **Dsp_LodToBinary()** can be used to convert this data. DSP programs in binary form load much faster than those in the ‘.LOD’ format.

Dsp_LoadProg() is used to execute programs stored on disk in the ‘.LOD’ format. **Dsp_ExecProg()** is used to execute programs stored in memory in binary format.

As with subroutines, programs are assigned a unique ability code that can be determined with **Dsp_GetProgAbility()**.

Sending Data to the DSP

Several functions transfer data to and from DSP programs and subroutines as follows:

- **Dsp_DoBlock()**
- **Dsp_BlkHandshake()**
- **Dsp_BlkUnpacked()**
- **Dsp_BlkWords()**
- **Dsp_BlkBytes()**
- **Dsp_MultBlocks()**
- **Dsp_InStream()**
- **Dsp_OutStream()**

You should read the description of each in the function reference and decide which is best suited for your needs.

Dsp_SetVectors() installs special purpose routines that are called when the DSP sends an interrupt indicating it is ready to send or receive data. **Dsp_RemoveInterrupts()** removes these routines from the vector table in memory.

DSP State

The HFx bits of the HSR register can be read atomically with the four calls **Dsp_Hf0()**, **Dsp_Hf1()**, **Dsp_Hf2()**, and **Dsp_Hf3()**. The current value of the ISR register may be read with **Dsp_Hstat()**.

DSP programs may also define special host commands at DSP vectors $13 and $14 to be triggered by the command **DSP_TriggerHC()**.

DSP Debugging

When full control over the DSP is necessary (such is the case for specialized debuggers), the command **Dsp_ExecBoot()** can be used to download up to 512 DSP **WORD**s of bootstrap code. The DSP will be reset before this happens. This call should only be used by advanced applications as it will cause other DSP functions to stop working unless those functions are properly supported.

User/Supervisor Mode

The **XBIOS** call **Supexec()** provides access to a special mode of the 680x0 processor called supervisor mode. Normal programs always execute in user mode. Programs operating in user mode, however, have less memory access privileges than those operating in supervisor mode.

Some special instructions of the 680x0 may only be executed in supervisor mode. In addition, any memory reads or writes to locations $0–$7FF or memory-mapped I/O must be made in supervisor mode.

To use **Supexec()**, simply pass it the address of a function to be called. When writing the function in 'C', you should be careful to define the function in a way that is safe for your compiler (see your compiler documentation for details).

While in supervisor mode, the **AES** should never be called.

MetaDOS

One special **XBIOS** opcode, **Metainit()** was reserved for a **TOS** extension called **MetaDOS**. **MetaDOS** was designed to supplement the OS to allow for more than 16 drives and to provide the extra support needed for CD-ROM drives. **MetaDOS** is no longer officially supported by Atari because of the increased functionality of **MultiTOS**.

MultiTOS allows the use of all 26 drive letters as well as providing loadable device drivers and file systems. See *Chapter 2: **GEMDOS*** for more information.

Keyboard and Mouse Control

The **XBIOS** has several functions that provide extended control over the keyboard and mouse. These functions should be used with care, however, as the keyboard and mouse are 'global' devices shared by other processes.

Initmous() is used to change the way the keyboard controller reports mouse movements to the system. Changing this mode will cause the **AES** and **VDI** to be unable to recognize mouse input.

Keytbl() allows you to read and manipulate the tables which translate IKBD scan codes into ASCII codes. This is essential when you want your application to run on Atari machines with foreign keyboards. Use **Keytbl()** to return a pointer to the internal table structure and then convert keycodes into ASCII by looking codes up in the appropriate table.

Loadable XBIOS Keyboard Tables

TOS versions 5.0 and greater support the loading of external keyboard tables when the '_AKP' cookie is present. In this case, if a file called 'KEYTBL.TBL' is found in the '\MULTITOS' directory of the boot drive, it will be loaded upon bootup to provide keyboard mapping changes. The format of the file is as follows:

Magic Table Identifier Word This should be a **WORD** value of 0x2771.
Unshifted Keyboard Table This is a 128 byte table of ASCII codes that are generated when no keyboard shift keys are being held down. There is one entry for each possible scan code.
Shifted Keyboard Table This is a 128 byte table of ASCII codes that are generated when the SHIFT key is being held down. There is one entry for each possible scan code.

CAPS-LOCK Keyboard Table
This is a 128 byte table of ASCII codes that are generated when CAPS-LOCK is engaged and no shift keys are being held. There is one entry for each possible scan code.
Alternate-Unshifted Keyboard Table
This is a variable length table consisting of two-byte entries. Each entry consists of a scan code and the ASCII code generated when that scan code occurs while the ALTERNATE key (and no other) keyboard shift keys are being held. The list is terminated by a single **NULL** byte.
Alternate-Shifted Keyboard Table
This is a variable length table consisting of two-byte entries. Each entry consists of a scan code and the ASCII code generated when that scan code occurs while the ALTERNATE key and the SHIFT key is being held. The list is terminated by a single **NULL** byte.
Alternate CAPS-LOCK Keyboard Table
This is a variable length table consisting of two-byte entries. Each entry consists of a scan code and the ASCII code generated when that scan code occurs while the ALTERNATE key is being held with the CAPS-LOCK mode in effect. The list is terminated by a single **NULL** byte.

Bioskeys() returns any mapping changes made by **Keytbl()** to their original state.

The configuration functions **Cursconf()** and **Kbrate()** set the cursor blink rate and keyboard repeat rates respectively. These settings should only be changed by a CPX or other configuration utility at the user's request as they are global and affect all applications.

IKBD Intelligent Keyboard Controller

The IKBD Controller is an intelligent hardware device that handles communications between the computer and the keyboard matrix. The **XBIOS** function **Ikbdws()** can be used to transmit command strings to the IKBD controller. For further information about the IKBD, consult *Chapter 5: **Hardware***.

Disk Functions

Boot Sectors

Both floppy disks and hard disks share a similar format for boot sectors as follows:

Name	Offset	Contents
BRA	0x0000	This **WORD** contains a 680x0 BRA.S instruction to the boot code in this sector if the disk is executable, otherwise it is unused.
OEM	0x0002	These six bytes are reserved for use as any necessary filler information. The disk-based **TOS** loader program places the string 'Loader' here.
SERIAL	0x0008	The low 24-bits of this **LONG** represent a unique disk serial number.

BPS	0x000B	This is an Intel format **WORD** (low byte first) which indicates the number of bytes per sector on the disk.
SPC	0x000D	This is a **BYTE** which indicates the number of sectors per cluster on the disk.
RES	0x000E	This is an Intel format **WORD** which indicates the number of reserved sectors at the beginning of the media (usually one for floppies).
NFATS	0x0010	This is a **BYTE** indicating the number of File Allocation Table's (FAT's) on the disk.
NDIRS	0x0011	This is an Intel format **WORD** indicating the number of ROOT directory entries.
NSECTS	0x0013	This is an Intel format **WORD** indicating the number of sectors on the disk (including those reserved).
MEDIA	0x0015	This **BYTE** is a media descriptor. Hard disks set this value to 0xF8, otherwise it is unused.
SPF	0x0016	This is an Intel format **WORD** indicating the number of sectors per FAT.
SPT	0x0018	This is an Intel format **WORD** indicating the number of sectors per track.
NSIDES	0x001A	This is an Intel format **WORD** indicating the number of sides on the disk.
NHID	0x001C	This is an Intel format **WORD** indicating the number of hidden sectors on a disk (currently ignored).
BOOTCODE	0x001E	This area is used by any executable boot code. The code must be completely relocatable as its loaded position in memory is not guaranteed.
CHECKSUM	0x01FE	The entire boot sector **WORD** summed with this Motorola format **WORD** will equal 0x1234 if the boot sector is executable or some other value if not.

The boot sector may be found on side 0, track 0, sector 1 of each physical disk.

The Floppy Drive

The **XBIOS** provides several functions used for reading, writing, verifying, and formatting sectors on the hard disk.

Floprd() and **Flopwr()** read and write from the floppy drive at the sector level rather than the file level. For example, these functions could be used to create executable boot sectors on a floppy disk. **Flopver()** can be used to verify written sectors against data still in memory.

Formatting a floppy disk is accomplished with **Flopfmt()**. After a floppy is completely formatted use the function **Protobt()** to create a prototype boot sector (as shown above) which can then be written to sector #1 to make the disk usable by **TOS**.

ASCI and SCSI DMA

The functions **DMAread()** and **DMAwrite()** were added as of **TOS** 2.00. These functions provide a method of accessing ACSI and SCSI devices at the sector level.

ASCI accesses must not use alternate RAM as a transfer buffer because they are performing DMA. The TT030 uses handshaking for SCSI so alternate RAM transfers are safe. SCSI transfers on the Falcon030 do, however, use DMA so alternate RAM must be avoided.

If you need to transfer data using these functions to an alternate RAM buffer, use the special standard memory block pointed to by the cookie '_FRB' as an intermediary point between the two types of RAM. You must also use the '*_flock*' system variable (at 0x43E) to lock out other attempted uses of this buffer.

Each physical hard disk drive must contain a boot sector. The boot sector for hard disk drives is the same as floppies except for the following locations:

Name	Offset	Contents
hd_siz	0x01C2	This is a Motorola format **LONG** that indicates the number of physical 512-byte sectors on the device.
Partition Header #0	0x01C6	This section contains a 12 **BYTE** partition information block for the first logical partition.
Partition Header #1	0x01D2	This section contains a 12 **BYTE** partition information block for the second logical partition.
Partition Header #2	0x1DE	This section contains a 12 **BYTE** partition information block for the third logical partition.
Partition Header #3	0x1EA	This section contains a 12 **BYTE** partition information block for the fourth logical partition.
bst_st	0x1F6	This is a Motorola format **LONG** that indicates the sector offset to the bad sector list (from the beginning of the physical disk).
bst_cnt	0x01FA	This is a Motorola format **LONG** that indicates the number of 512-byte sectors reserved for the bad sector list.

The partition information block is defined as follows:

Name	Offset	Contents
p_flg	0x00	This is a **BYTE** size bit field indicating the partition state. If bit 0 is set, the partition exists, otherwise it does not. If bit 7 is set, the partition is bootable, otherwise it is not. Bits 1-6 are unused.
p_id	0x01	This is a three **BYTE** field that indicates the partition type as follows: **Contents** — **Meaning** '**GEM**' — Regular Partition (<16MB) '**BGM**' — Big Partition (>=16MB) '**XGM**' — Extended Partition
p_st	0x04	This is a Motorola format **LONG** that indicates the start of the partition as an offset specified in 512-byte sectors.
p_size	0x08	This is a Motorola format **LONG** that indicates the size of the partition in 512-byte sectors.

A hard disk may have up to four standard (GEM or BGM) partitions or three standard and one extended (XGM) partition. The first partition of a hard disk must be a standard one.

Extended Partitions

The first sector of an extended partition contains a standard boot sector with hard disk information except that the *hd_siz*, *bst_st*, and *bst_cnt* fields are unused. At least one, but no more than two (not necessarily the first two), partition headers are used. The first partition header is the same as described above except that *p_st* describes the offset from the beginning of the extended partition rather than the beginning of the physical disk.

If another partition needs to be linked, the second partition block should contain 'XGM' in its *p_id* field and an offset to the next extended partition in *p_st*.

The Bad Sector List

The bad sector list is a group of three-byte entries describing which physical sectors on the hard disk are unusable. The first three-byte entry contains the number of bad sectors recorded. The second three-byte entry is a checksum and when added to the entire bad sector list bytewise should cause the list to **BYTE** sum to 0xA5. If this is not the case then the bad sector list is considered bad itself.

The Serial Port

Application writers who develop communication programs will need to use some of the special functions the **XBIOS** provides for control of the serial port(s). Older Atari computers support only one serial port connected by the Multi-Function Peripheral (MFP) chip.

The Atari TT030 contains two MFP chips to provide two serial ports and one Serial Communications Chip (SCC) which controls two more serial ports. One of the SCC ports, however, can be switched over to control a Localtalk compatible network port as follows:

Switch to Serial 2 Connector:

```
Ongibit(0x80);
```

Switch to LAN connector:

```
Offgibit(0x7F);
```

The Mega STe is similar to the TT030, however, it has only one MFP chip to provide one less serial device.

The Atari Falcon030 uses a SCC chip to drive its single serial port and networking port. The Falcon030 does contain a MFP chip but it does not control any of the serial device hardware. The MFP's ring indicator has, however, been wired across the SCC to provide compatibility with older applications.

Serial Port Mapping

BIOS input and output calls to device #1 and **XBIOS** calls which configure the serial port always refer to the currently 'mapped' device as set with **Bconmap()**. The Modem CPX allows a user to map any installed device as the default. A program which is aware of the extra ports on newer machines can access them through their own **BIOS** device number as follows:

Device Number	Mega ST	TT030	Falcon030
1	Currently mapped device. **DEV_AUX**	Currently mapped device. **DEV_AUX**	Currently mapped device. **DEV_AUX**
6	Modem 1 (ST MFP) **DEV_MEGAMODEM1**	Modem 1 (ST MFP) **DEV_TTMODEM1**	—
7	Modem 2 (SCC B) **DEV_MEGAMODEM2**	Modem 2 (SCC B) **DEV_TTMODEM2**	Modem (SCC B) **DEV_FALCONMODEM**
8	Serial/LAN (SCC A) **DEV_MEGALAN**	Serial 1 (TT MFP) **DEV_TTSERIAL1**	LAN (SCC A) **DEV_FALCONLAN**
9	—	Serial 2/LAN (SCC A) **DEV_TTLAN**	—

Configuring the Serial Port

Rsconf() and **Iorec()** set the communication mode and input/output buffers of the currently mapped serial port. You should note that while some ports support transfer rates of greater than 19200 baud, this is the limit of the **Rsconf()** call. Other rates must currently be set in hardware (or with the **Fcntl()** when **MiNT** is present).

MFP Interrupts

Each MFP chip supports a number of interrupts used by the serial port and other system needs. The function **Mfpint()** should be used to set define a function in your application that handles one of these interrupts. **Jenabint()** and **Jdisint()** are used to enable/disable these interrupts respectively.

All MFP interrupt calls only work on ST compatible MFP serial ports. The RS-232 ring indicator is the only interrupt that has been wired through the MFP on a Falcon. Because of this, the ring indicator interrupt is the only RS-232 interrupt that may be changed with **Mfpint()** on a Falcon.

SCC Interrupts

The **XBIOS** functions used for setting MFP interrupts do not affect the SCC interrupts regardless of the **Bconmap()** mapping. Refer to the memory map for the location of SCC interrupt registers.

Printer Control

The **XBIOS** contains two functions used for controlling printers. Both functions are very outdated and should not be relied on in any ST.

Scrdmp() triggers the built-in ALT-HELP screen dump code. **Prtblk()** enables the built-in screen dump routine of the ST printing only the desired block to an Atari or Epson dot-matrix printer.

Setprt() configures the built-in screen dump routine as to the basic configuration of the attached printer.

Other XBIOS Functions

NVMaccess() accesses the non-volatile RAM present in the TT, Mega STe, and Falcon030. You should not read or write to this area as all of its locations are currently reserved.

The functions **Settime()** and **Gettime()** set the **BIOS** time and date. As of **TOS** 1.02, they also update the **GEMDOS** time as well.

Besides the sound capabilities of the **XBIOS** when running on a Falcon, the function **Dosound()** generates music on any Atari computer using the FM sound generator. The function works at the interrupt level processing a 'sound command list' you specify. It can be used to reproduce a single tone or a complete song in as many as three parts of harmony.

Random() generates a pseudo-random number using a built-in algorithm whose seed comes from the system 60kHz clock.

Ssbrk() is used by the operating system to reserve system RAM before **GEMDOS** is initialized. It should not be used by application programmers.

Puntaes() is useful only when using a disk-loaded version of **TOS**. It clears the OS from RAM and reboots the computer.

Midiws() is a similar function to **Ikbdws()** in that it writes to the MIDI controller. It is more useful at transferring large amounts of MIDI data than **Bconout()**.

The **Dbmsg() XBIOS** call is added by supporting debuggers as a method of transferring debugging messages between the application and debugger. The Atari Debugger (DB) currently supports this interface.

XBIOS Function Calling Procedure

XBIOS system functions are called via the TRAP #14 exception. Function arguments are pushed onto the current stack (user or supervisor) in reverse order followed by the function opcode. The calling application is responsible for correctly resetting the stack pointer after the call.

The **XBIOS**, like the **BIOS** may utilize registers D0-D2 and A0-A2 as scratch registers and their contents should not be depended upon at the completion of a call. In addition, the function opcode placed on the stack will be modified.

The following example for **Getrez()** illustrates calling the **XBIOS** from assembly language:

```
move.w      #$04,-(sp)
trap        #14
addq.l      #6,sp
```

A 'C' binding for a generic **XBIOS** handler would be as follows:

```
_xbios:
        ; Save the return code from the stack
        move.l  (sp)+,trp14ret
        trap    #14
        move.l  trp14ret,-(sp)
        rts

        .bss
trp14ret:
        .ds.l       1
```

The **XBIOS** is re-entrant to three levels, however there is no depth checking performed so interrupt handlers should avoid intense **XBIOS** usage. In addition, no disk or printer usage should be attempted from the system timer interrupt, critical error, or process-terminate handlers.

Calling the XBIOS from an Interrupt

The **BIOS** and **XBIOS** are the *only* two OS sub-systems which may be called from an interrupt handler. Precisely *one* interrupt handler at a time may use the **XBIOS** as shown in the following code segment:

```
savptr          equ     $4A2
savamt          equ     $23*2

myhandler:
                sub.l   #savamt,savptr

                ; BIOS calls may be performed here

                add.l   #savamt,savptr

                rte     ; (or rts?)
```

Certain **XBIOS** calls are not re-entrant because they call **GEMDOS** routines. The **Setscreen()** function, and any DSP function which loads data from disk should not be attempted during an interrupt.

It is not possible to use this method to call **XBIOS** functions during an interrupt when running under **MultiTOS**.