– CHAPTER 5 –

HARDWARE

Overview

This chapter will cover those aspects of Atari software programming that can only be accomplished by accessing hardware registers directly. In most cases, Atari has provided OS calls to manipulate the hardware. When an OS call exists to access hardware, it should *always* be used to ensure upward and backward compatibility. Keep in mind that access to hardware registers is limited to those applications operating in supervisor mode only (except where noted otherwise).

Besides those hardware registers discussed here, a complete list of I/O registers, system variables, and interrupt vectors are contained in *Appendix B:* ***Memory Map***.

The 680x0 Processor

Atari computers use the Motorola MC68000 or MC68030. Third party devices have also been created to allow the use of a MC68010, MC68020, or MC68040 processor. The system cookie '_CPU' should be used to determine the currently installed processor. The following table lists the 680x0's interrupt priority assignments:

Level	Assignment
7	NMI
6	MK68901 MFP
5	SCC[1]
4	VBLANK (Sync)
3	VME Interrupter[2]
2	HBLANK (Sync)
1	Unused

Interrupts may be disabled by setting the system interrupt mask (bits 8-10 of the SR register) to a value higher than the interrupts you wish to disable. Setting the mask to a value of 7 will effectively disable all interrupts (except the level 7 non-maskable interrupt).

The Data/Instruction Caches

The Atari TT030 and Falcon030 contain onboard data and instruction caches. These caches may be controlled by writing to the CACR register (in supervisor mode). The following table lists longword values that may be written to the CACR to enable or disable the caches:

Value to Write to CACR	Effect
0xA0A	Flush and disable both caches.
0x101	Enable both caches.
0xA00	Flush and disable the data cache.
0x100	Enable the data cache.

[1] On a computer without an SCC chip, this interrupt is unused.

[2] On a computer without a VME bus, this interrupt is unused.

0xA	Flush and disable the instruction cache.
0x1	Enable the instruction cache.

The 68881/882 Floating Point Coprocessor

A MC6888x math coprocessor may be installed in a Mega ST, Mega STe, or a Falcon030. The TT030 has one installed in its standard configuration. The 6888x is interfaced to the 68000 in peripheral mode and to the 68030 in coprocessor mode. Thus, the TT030 and Falcon030 computers access the 6888x in coprocessor mode while the Mega ST and MegaSTe computers access the 6888x in peripheral mode.

Coprocessor Mode

When the 6888x is interfaced in coprocessor mode, using it is as simple as placing floating-point instructions in the standard instruction stream (use a coprocessor ID of 1). The 68030 will properly dispatch the instruction and respond to exceptions through the following vectors:

Vector Address	Assignment
0x0000001C	FTRAPcc Instruction
0x0000002C	F-Line Emulator
0x00000034	Co-processor Protocol Violation
0x000000C0	Branch or Set on Unordered Condition
0x000000C4	Inexact Result
0x000000C8	Floating-Point Divide by Zero
0x000000CC	Underflow
0x000000D0	Operand Error
0x000000D4	Overflow
0x000000D8	Signaling NAN

Peripheral Mode

Utilizing an installed math coprocessor interfaced using peripheral mode requires the use of several hardware registers mapped to special coprocessor registers. Unlike most hardware registers, these do not have to be accessed in supervisor mode. Atari computers map the 6888x registers to the following locations:

Address	Length	Register	Description
0xFFFFFA40	**WORD**	***FPCIR***	Status register
0xFFFFFA42	**WORD**	***FPCTL***	Control Register
0xFFFFFA44	**WORD**	***FPSAV***	Save Register
0xFFFFFA46	**WORD**	***FPREST***	Restore Register
0xFFFFFA48	**WORD**	***FPOPR***	Operation word register
0xFFFFFA4A	**WORD**	***FPCMD***	Command register
0xFFFFFA4C	**WORD**	***FPRES***	Reserved
0xFFFFFA4E	**WORD**	***FPCCR***	Condition Code Register
0xFFFFFA50	**LONG**	***FPOP***	Operand Register

To execute a floating point instruction, use the following protocol for communicating data with the 6888x:

1. Wait for the chip to be idle.
2. Write a valid 6888x command to ***FPCMD***.
3. If necessary for the command, write an operand to ***FPOP***.
4. Wait for the status port to indicate the command is complete.
5. Read any return data from ***FPOP***.

Step one is achieved by waiting for a value of 0x0802 to appear in the status register (after ANDing with 0xBFFF) as follows:

```
while( ( FPCIR & 0xBFFF) != 0x0802 ) ;
```

Steps two and three involve writing the command word to ***FPCMD*** and any necessary operand data to ***FPOP***. A primitive response code will be generated (and should be read) between each write to either ***FPCMD*** or ***FPOP***. For a listing of primitive response codes returned by the 68881, consult the **MC68881/68882 Floating-Point Coprocessor User's Manual (2nd edition), Motorola publication MC68881UM/AD rev. 2, ISBN 0-13-567-009-8**.

After the operation is complete (step 4), data may be read from the 68881 in ***FPOP*** (step 5).

When sending or receiving data in **FPOP**, the following chart details the transfer ordering and alignment:

	Order	31 — 24	23 — 16	15 — 8	7 — 0
BYTE	1st	BYTE	UNUSED		
WORD	1st	WORD		UNUSED	
LONG/ SINGLE	1st	LONG/SINGLE			
DOUBLE	1st	MSB Double Precision			
	2nd	Operand			LSB
EXTENDED	1st	MSB Extended			
	2nd	Precision			
	3rd	Operand			LSB

The following code demonstrates transferring two single precision floating-point numbers to the 68881, multiplying them, and returning the result.

```
/* Number of iterations before an error is triggered */
#define FPCOUNT       0x80

#define FPCIR             ((WORD *)(0xFFFFFA40L))
#define FPCMD             ((WORD *)(0xFFFFFA4AL))
#define FPOP          ((float *)(0xFFFFFA50L))
```

```
WORD fpcount, dum;

/* fperr() is user-defined */

#define FPwait() {    fpcount = FPCOUNT; \
                      while((*FPCIR & 0xBFFF) != 0x0802) \
                          if(!(--fpcount)) fperr();  }

#define FPsglset(r,v) { FPwait(); \
                        *FPCMD = (0x5400 | ((r) << 7)); \
                        while((*FPCIR & 0xFFF0) != 0x8C00) \
                          if(!(--fpcount)) fperr();     \
                        *FPOP = (v); }

#define FPsglmul(r1,r2) {     FPwait();    \
                          *FPCMD = (0x0027 | ((r2) << 10) | ((r1) << 7));     \
                          dum = *FPCIR + 1;}

/* dum = FPCIR +1; forces the status register to be read
   (we assume the data's good) */

#define FPsglget(r,var) { FPwait();    \
                          *FPCMD = (0x6400 | ((r) << 7));         \
                          while(*FPCIR != 0xb104)        \
                              if(!(--fpcount)) fperr();     \
                          var = *FPOP;     }

/*
 * void sglmul( float *f1, float *f2 );
 *
 * Multiplies f1 by f2. Returns result in f1.
 *
 */

void
sglmul( float &f1, float &f2 )
{
        FPsglset( 0, *f1 );
        FPsglset( 1, *f2 );
        FPsglmul( 0, 1 );
        FPsglget( 0, *f1 );
}
```

Cartridges

All Atari computers support an external 128K ROM cartridge port. Cartridges may be created to support applications or diagnostic tools. The 128K of address space allocated to cartridges appears from address 0xFA0000 to 0xFBFFFF. Newer Atari computers support larger cartridges (this is because the address space would no longer overlap the OS). All program code must be compiled to be relative of this base address.

The **LONG** appearing at 0xFA0000 determines the type of cartridge installed as follows:

Cartridge	LONG Value
Application	0xABCDEF42

Diagnostic	0xFA52255F

Diagnostic Cartridges

Diagnostic cartridges are executed almost immediately after a system reset. The OS uses a 680x0 JMP instruction to begin execution at address 0xFA0004 after having set the Interrupt Priority Level (IPL) to 7, entering supervisor mode, and executing a RESET instruction to reset external hardware devices.

Upon execution, register A6 will contain a return address which should be JMP'd to if you wish to continue system initialization at any point. The stack pointers will contain garbage. In addition, keep in mind that no hardware has been initialized, particularly the memory controller. All system memory sizing and initialization must be performed by the diagnostic cartridge.

Application Cartridges

Application cartridges should contain one or more application headers beginning at location 0xFA0004 as follows (one cartridge may contain one or many applications):

Name	Offset	Meaning
CA_NEXT	0x00	Pointer to the next application header (or **NULL** if there are no more).
CA_INIT	0x04	Pointer to the application's initialization code. The high eight bits of this pointer have a special meaning as follows: **Bit Set** / **Meaning** 0 Execute prior to display memory and interrupt vector initialization. 1 Execute just before **GEMDOS** is initialized. 2 (unused) 3 Execute prior to boot disk. 4 (unused) 5 Application is a Desk Accessory. 6 Application is not a **GEM** application. 7 Application needs parameters.
CA_RUN	0x08	Pointer to application's main entry point.
CA_TIME	0x0C	Standard **GEMDOS** time stamp.
CA_DATE	0x0E	Standard **GEMDOS** date stamp.
CA_SIZE	0x10	Size of application in bytes.
CA_NAME	0x14	**NULL** terminated ASCII filename in standard **GEMDOS** 8+3 format.

When application cartridges are present, **GEMDOS** will allow a special 'c' (lowercase) drive to be accessed. Executable files appear on this drive as they would on any standard disk. This 'drive' may also be installed on the desktop.

Game Controllers

The Atari 1040STe and Falcon030 support new enhanced joystick controls as well as older style CX-40 controls. For the usage and polling of the older style controls, refer to the following section which discusses the IKBD controller. This section will focus specifically on the newer style of controllers.

Joysticks

Enhanced joysticks are read by a two-step process. The **WORD** at address 0xFF9202 is written to using a mask which determines which values may subsequently be read from the **WORD**s at address 0xFF9200 and 0xFF9202. Valid mask values and the keys that may be read follow:

Read Controller 0 at 0xFF9200		
Write Mask	Bit 0 Clear	Bit 1 Clear
0xFFFE	Pause	Fire 0
0xFFFD	-	Fire 1
0xFFFB	-	Fire 2
0xFFF7	-	Option

Read Controller 1 at 0xFF9200		
Write Mask	Bit 2 Clear	Bit 3 Clear
0xFFEF	Pause	Fire 0
0xFFDF	-	Fire 1
0xFFBF	-	Fire 2
0xFF7F	-	Option

Read Controller 0 at 0xFF9202				
Write Mask	Bit 8 Clear	Bit 9 Clear	Bit 10 Clear	Bit 11 Clear
0xFFFE	Up	Down	Left	Right
0xFFFD	Key *	Key 7	Key 4	Key 1
0xFFFB	Key 0	Key 8	Key 5	Key 2
0xFFF7	Key #	Key 9	Key 6	Key 3

Read Controller 1 at 0xFF9202				
Mask	Bit 12 Clear	Bit 13 Clear	Bit 14 Clear	Bit 15 Clear
0xFFEF	Up	Down	Left	Right
0xFFDF	Key *	Key 7	Key 4	Key 1
0xFFBF	Key 0	Key 8	Key 5	Key 2
0xFF7F	Key #	Key 9	Key 6	Key 3

To read the joystick, write a mask value corresponding to the row of keys/positions you wish to interrogate to 0xFF9202. Next, read back a **WORD** from either 0xFF9200 or 0xFF9202. As indicated in the table, cleared bits mean that a key is being pressed or a joystick is moved in that direction.

Paddles

Two paddles may be plugged into each joystick port. Each paddle returns an 8-bit value indicating its position (0 = full counter-clockwise, 255 = full clockwise) at the addresses shown below. Unlike joysticks, paddle positions are returned automatically with no need to write to an address prior to a read. Paddle fire buttons, however, are mapped and read in the same manner as the joysticks. See the discussion of joysticks above for an explanation.

Byte Address	Paddle
0xFF9211	X Paddle 0
0xFF9213	Y Paddle 0
0xFF9215	X Paddle 1
0xFF9217	Y Paddle 1

Light Gun/Pen

Joystick port 0 supports a light gun or pen. The position that the gun is pointing to is returned in the **WORD** registers at 0xFF9220 (X position) and 0xFF9222 (Y position). Only the lower 10 bits are significant giving a range of values from 0-1023.

The IKBD Controller

The Atari 16/32 bit computer line uses the Intelligent Keyboard Controller (IKBD) for keyboard, joystick (old-style CX-40), mouse, and clock communication. The 6850 ACIA serial communications chip is used to transfer data packets from the IKBD interface to the host computer.

The **TOS** calls **Bconout(** 4, ??? **)**, **Ikbdws()**, and **Initmous()** handle communication to the controller. Return messages from the controller must be processed by placing a specialized handler in the vector table returned by the **XBIOS** call **Kbdvbase()**. **Kbdvbase()** returns the pointer to a vector table as follows:

```
typedef struct
{
        void (*midivec)( UBYTE data );       /* Passed in d0 */
        void (*vkbderr)( UBYTE data );       /* Passed in d0 */
        void (*vmiderr)( UBYTE data );       /* Passed in d0 */
        void (*statvec)( char *packet );     /* Passed in a0 */
        void (*mousevec)( char *packet );        /* Passed in
        a0 */
        void (*clockvec)( char *packet );        /* Passed in
        a0 */
        void (*joyvec)( char *packet );      /* Passed in a0 */
        void (*midisys)( VOID );
        void (*ikbdsys)( VOID );
        char ikbdstate;
```

```
} KBDVECS;
```

When an IKBD message is pending, the interrupt handler for the ACIAs calls either the midisys handler or the ikbdsys handler to retrieve the data and handle any errors. The default action for the *ikbdsys* handler is to decide whether the packet contains error, status, joystick, clock, or mouse information and to route it appropriately to *vkbderr*, *statvec*, *joyvec*, *clockvec*, or *mousevec*. Keyboard packets are handled internally by *ikbdsys*.

Your handler should be patched into the appropriate vector and, if appropriate, expect the packet buffer to be pointed to by register A0. Unless your handler is designed to completely replace the functions of the default handler you should jump through the original vector pointer upon exit, otherwise simply use the 680x0 RTS instruction.

Each byte received through the keyboard ACIA falls into one of the following categories as follows:

Category	Value(s)	Meaning
Keyboard Make Code	0x00–0x7F	One of these values is generated each time a key is depressed.This value may be used with **Keytbl()** to generate an ASCII code for the scan code.
Keyboard Break Code	0x80–0xFF	This code is generated when a key previously depressed has been released. It represents the make code logically OR'ed with 0x80.
Status Packet Header	0xF6	This codes indicate the beginning of a multiple byte status packet.
Absolute Mouse Position	0xF7	See Below
Relative Mouse Position	0xF8-0xFB	See Below
Time-of-Day	0xFC	See Below
Joystick Report	0xFD	See Below
Joystick 0 Event	0xFE	See Below
Joystick 1 Event	0xFF	See Below
Status Packet Data	Any	When the *ikbdstate* variable (found in the **KBDVECS** structure) is non-zero, it represents the number of remaining bytes to retrieve that are part of a status packet and should thus not be treated as any of the above codes.

The Keyboard

Keyboard keys generate both a 'make' and 'break' code for each complete press and release respectively. The 'make' code is equivalent to the high byte of the IKBD scan code. 'make' codes are not related in any way to ASCII codes. They represent the physical position of the key in the keyboard matrix and may vary in keyboards designed for other countries. The **XBIOS** function **Keytbl()** provides lookup values which make internationalization possible. The key 'break' code is the 'make' code logically ORed with 0x80.

It should be noted that 'key repeats' are not generated by the ACIA but by a coordination of the *ikbdsys* and system timer handlers.

The Mouse

The mouse may be programmed to return position reports in either absolute, relative, or keycode mode (it is by default programmed to return relative position reports).

In relative reporting mode, the IKBD generates a mouse packet each time a mouse button is pressed or released, and every time the mouse is moved over a preset threshold distance (which is configurable). A relative mouse report packet is headed by a byte value between 0xF8 and 0xFB followed by the X and Y movement of the mouse as signed bytes. If the movement is greater than can be represented as signed bytes (-128 to 127), multiple packets are sent.

The header byte determines the state of the mouse buttons as follows:

Header	Mouse Button State
0xF8	No buttons depressed.
0xF9	Left button depressed.
0xFA	Right button depressed.
0xFB	Both buttons depressed.

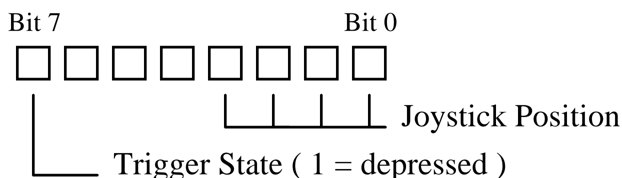
In absolute reporting mode, the IKBD only generates a mouse packet when interrogated. Mouse packets in absolute mode are headed by a 0xF7 byte followed by the MSB and LSB of the X value and the MSB and LSB of the Y value respectively. The minimum and maximum X and Y values are user-definable.

Keycode reporting mode generates keyboard 'make' and 'break' codes for mouse movements rather than sending standard mouse packets. Mouse movement is translated into the arrow keys and the codes 0x74 and 0x75 represent the left and right mouse button respectively. 'break' codes are sent immediately after the corresponding 'make' code is delivered.

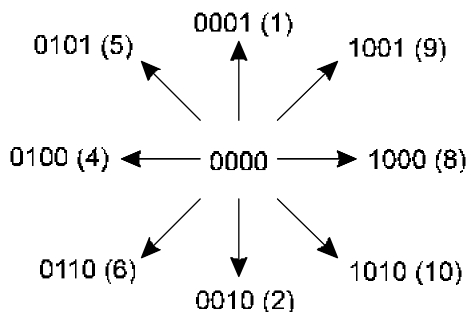
The Joystick

The basic CX-40 style joystick controls are still present on every Atari computer. Atari recommends that these ports should not be supported when STe/Falcon030 enhanced joysticks are present unless the option for four-player play is desired. While no direct **TOS** support is available for reading these ports, it is possible using the IKBD controller in one of several joystick reporting modes.

Joystick event reporting mode (the default) sends a joystick packet each time the status of one of the joysticks changes. The joystick packet header is 0xFE if the state of joystick 0 has changed or 0xFF if the status of joystick 1 has changed. This header byte is followed by a **BYTE** containing the new state of the joystick as follows:



The four bits corresponding to joystick position can be interpreted as follows:



Joysticks may be interrogated at any time by sending an interrogate command (as described later in this chapter). The packet response to this command is 0xFD followed by the **BYTE** report of joystick 0 and 1 (as shown above).

The joysticks may be placed into joystick monitoring or fire button monitoring mode. In these modes, all other IKBD communication is stopped and all processor time is devoted to the processing of packets. Joystick monitoring mode cause the IKBD to send a continuous stream of two-byte packets as follows: The first byte contains the status of joystick buttons 0 and 1 in bits 1 and 0 respectively. The second byte contains the position state of joystick 0 in the high nibble and joystick 1 in the lower nibble (the position state can be interpreted as shown in the diagram above).

Fire button monitoring mode constantly scans joystick button 1 and returns the results in **BYTE**s packed with 8 reports each (one per bit). These modes may be paused or halted using the appropriate commands.

Joystick keycode mode is similar to mouse keycode mode. This mode translates all joystick position information into arrow keys. A 'make' code of 0x74 is generated when joystick button 0 is depressed and a 'make' code of 0x75 is generated when joystick button 1 is depressed. The rate at which the IKBD controller generates these joystick events can be controlled using commands discussed in the following section.

Time-of-Day

The IKBD controller maintains a separate time-of day clock that is kept synchronized with **GEMDOS** time by OS calls. A time-of-day packet may be requested using the method shown below under IKBD commands.

The response packet from the IKBD is seven bytes in length identified by its header byte of 0xFC and followed by six UBYTES containing the year (last two digits), month, day, hours (0-24), minutes, and seconds in BCD format (ex: a month byte in December would be 0x12).

IKBD Commands

Commands may be sent to the IKBD using any of the **TOS** function calls described above. Some commands may generate packets while other commands change the operating state of the IKBD controller. Unrecognized command codes are treated as NOPs. The following lists valid IKBD command codes:

Command BYTE	Result
0x07	Set mouse button action. This command **BYTE** should be followed by a **BYTE** which describes how the mouse buttons should be treated as follows: **BYTE** — **Meaning** 0x00 — Default mode. 0x01 — Mouse button press triggers an absolute position report. 0x02 — Mouse button release triggers an absolute position report. 0x03 — Mouse button press and release triggers absolute position reports. 0x04 — Mouse buttons report key presses.
0x08	Enable relative mouse position reporting (default).
0x09	Enable absolute mouse position reporting. This command is followed by the MSB and LSB of the X and Y coordinate maximum values for the mouse.
0x0A	Enable mouse keycode mode. This command is followed by two **BYTE**s indicating the maximum number of mouse 'ticks' required to generate a keycode for the X and Y axis respectively.

0x0B	Set mouse threshold. This command is followed by two **BYTE**s which determine the number of mouse 'ticks' required to generate a mouse position report in relative positioning mode.
0x0C	Set mouse scale. This command is followed by two **BYTE**s which determine the number of mouse 'ticks' for each single coordinate on the X and Y axis respectively.
0x0D	Interrogate mouse position. This command generates an absolute mouse position report.
0x0E	Load mouse position. This command sets the mouse position based on the current coordinate system in absolute reporting mode. The command is followed by a filler **BYTE** of 0x00 and the MSB and LSB of the new X and Y axis for the mouse.
0x0F	Set Y=0 to the bottom. This command changes the origin of the mouse coordinate system to the upper left of the screen.
0x10	Set Y=0 to the top. This command changes the origin of the mouse coordinate system to the lower left of the screen.
0x11	Resume sending data. This command (or for that matter any command) will cause the IKBD to resume sending packet data to the host.
0x12	Disable all mouse packet reporting. Any valid mouse command resets this state. If the mouse buttons have been programmed to act like keyboard keys, this command will have no effect on them.
0x13	Pause output. All output from the IKBD controller is halted until a 'Resume' or other command is received.
0x14	Set joystick event reporting mode. This command causes a joystick report to be generated whenever the state of either joystick changes.
0x15	Set joystick interrogation mode. This command causes the IKBD to generate joystick packets only when requested by the host.
0x16	Joystick interrogation. This command causes a joystick packet indicating the status of both joysticks to be generated.
0x17	Enables joystick monitoring mode. Besides serial communication and the maintenance of the time-of-day clock, this command causes only special joystick reports to be generated. The command **BYTE** should be followed by a **BYTE** indicating how often the joystick should be polled in increments of 1/100ths of a second.
0x18	Enables fire button monitoring mode. As above, this mode limits the IKBD to serial communication, updating the time-of-day clock, and the reporting of the state of joystick button 1.

0x19	Set joystick keycode mode. This command is followed by six **BYTE**s as follows: **BYTE** – **Meaning** 1 – The length of time (in tenths of a second) before the horizontal breakpoint is reached. 2 – Same as above for the vertical plane. 3 – The length of time (in tenths of a second) between key repeats before the velocity breakpoint is reached. 4 – Same as above for the vertical plane. 5 – The length of time (in tenths of a second) between key repeats after the velocity breakpoint is reached. 6 – Same as above for the vertical plane.
0x1A	Disable joystick event reporting.
0x1B	Set the time of day clock. This command is followed by six **BYTE**s used to set the IKBD clock. These **BYTE**s are in binary-coded decimal (BCD) format. Each **BYTE** contains two digits (0-9), one in each nibble. The format for these **BYTE**s is as follows: **BYTE** – **Meaning** 1 – Year (last two digits) 2 – Month 3 – Date 4 – Hours (0-23) 5 – Minutes (0-59) 6 – Seconds (0-59)
0x1C	Interrogate the time-of-day clock. This command returns a packet headed by the value 0xFC followed by six **BYTE**s as indicated above.
0x20	Load **BYTE**s into the IKBD memory. This command is followed by at least three **BYTE**s containing the MSB and LSB of the address into which to load the data, the number of **BYTE**s to load (0-127), and the data itself.
0x21	Read **BYTE**s from the IKBD controller. This command is followed by two **BYTE**s containing the MSB and LSB of the address to read from. This returns a packet headed by the **BYTE** values 0xF6 and 0x20 followed by the memory data.
0x22	Execute a subroutine on the IKBD controller. This command **BYTE** is followed by two **BYTE**s containing the MSB and LSB of the memory location of the subroutine to execute.
0x80	Reset the IKBD controller. This command is actually a two-**BYTE** command. The **BYTE** 0x80 must be followed by a **BYTE** of 0x01 or the command will be ignored.

<table>
<tr><td>0x87</td><td>Return a status message containing the current mouse action state. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at statvec) as follows:

<u>BYTE</u> <u>Meaning</u>
1 0xF6
2 0x07
3 Current mouse action state (see command 0x07)
4-8 0</td></tr>
<tr><td>0x88</td><td>Return a status message containing the current mouse mode. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at statvec) as follows:

<u>BYTE</u> <u>Meaning</u>
1 0xF6
2 Current mode as follows:
0x08 = Relative mode
0x09 = Absolute mode
0x0A = Keycode mode
3 Absolute mode: MSB of maximum X position (units to current scale).
Keycode mode: Horizontal distance threshold that must be passed prior to sending a keycode.
Relative mode: 0
4 Absolute mode: LSB of maximum X position.
Keycode mode: Vertical distance threshold that must be passed prior to sending a keycode.
Relative mode: 0
5 Absolute mode: MSB of maximum Y position (units to current scale).
Keycode mode: 0
Relative mode: 0
6 Absolute mode: LSB of maximum Y position.
Keycode mode: 0
Relative mode: 0
7-8 0</td></tr>
<tr><td>0X89</td><td>Same as 0x88.</td></tr>
<tr><td>0X8A</td><td>Same as 0x88.</td></tr>
</table>

0x8B	Return a status message containing the current mouse threshold state. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at *statvec*) as follows: **<u>BYTE</u>** **<u>Meaning</u>** 1 0xF6 2 0x0B 3 Number of horizontal mouse 'ticks' that must be traveled prior to sending a mouse packet. 4 Number of vertical mouse 'ticks' that must be traveled prior to sending a mouse packet. 5-8 0
0x8C	Return a status message containing the current mouse scaling factor. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at *statvec*) as follows: **<u>BYTE</u>** **<u>Meaning</u>** 1 0xF6 2 0x0C 3 Horizontal mouse 'ticks' between a change in mouse position on the X axis. 4 Vertical mouse 'ticks' between a change in mouse position on the Y axis. 5-8 0
0x8F	Return a status message containing the current origin point of the Y axis used for mouse position reporting. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at *statvec*) as follows: **<u>BYTE</u>** **<u>Meaning</u>** 1 0xF6 2 0x0F = Bottom is (Y=0) 0x10 = Top is (Y=0) 3-8 0
0x90	Same as 0x8F.
0x92	Return a status message containing the current state of mouse reporting. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at *statvec*) as follows: **<u>BYTE</u>** **<u>Meaning</u>** 1 0xF6 2 0x00 = Mouse reporting enabled. 0x12 = Mouse reporting disabled. 3-8 0

<table>
<tr><td>0x94</td><td>Return a status message containing the current joystick mode. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at statvec) as follows:

<u>BYTE</u> <u>Meaning</u>
1 0xF6

2 Current mode as follows:
0x14 = Event reporting mode
0x15 = Interrogation mode
0x19 = Keycode mode

3 Keycode mode: This value represents the amount of time (in tenths of a second) that keycodes are returned to the host for horizontal position events at the initial velocity level (after this time expires, the secondary velocity level is used).
Event recording mode: 0
Interrogation mode: 0

4 Keycode mode: Same as BYTE 3 for vertical events.
Event recording mode: 0
Interrogation mode: 0

5 Keycode mode: This value represents the initial horizontal velocity level (in tenths of a second). This is the initial rate at which keycodes are generated.
Event recording mode: 0
Interrogation mode: 0

6 Keycode mode: Same as byte 5 for vertical events.
Event recording mode: 0
Interrogation mode: 0

7 Keycode mode: This value represents the secondary horizontal velocity level (in tenths of a second). This is the rate used after the amount of time specified in bytes 3-4 expires.
Event recording mode: 0
Interrogation mode: 0

8 Keycode mode: Same as byte 7 for vertical events.
Event recording mode: 0
Interrogation mode: 0</td></tr>
<tr><td>0x95</td><td>Same as 0x94.</td></tr>
<tr><td>0x99</td><td>Same as 0x94.</td></tr>
</table>

0x9A	Return a status message containing the current status of the joystick. After receiving this command the IKBD will respond by sending a status packet (which may be intercepted at *statvec*) as follows: **BYTE** **Meaning** 1 0xF6 2 0x00 = Joystick enabled 0x1A = Joystick disabled 3-8 0

STe/TT030 DMA Sound

The Atari STe, Mega STe, TT030, and Falcon030 are all equipped with the ability to playback stereo digital audio. Only the Falcon030, however, has supporting **XBIOS** calls which eliminate the need for the programmer to directly access the sound system hardware. Although the Falcon030 has a more sophisticated sound system than the earlier Atari machines, the hardware registers have been kept compatible so older applications should function as expected. Programmers designing Falcon030 applications which use digital audio should use the appropriate **XBIOS** calls.

The STe, MegaSTe, and TT030 support 8-bit monophonic or stereophonic sound samples. Samples should be signed (-128 to 127) with alternating left and right channels (for stereo) beginning with the left channel. Samples may be played at 50 kHz, 25 kHz, 12.5 kHz, or 6.25 kHz (6.25 kHz is not supported on the Falcon030).

DMA Sound Registers

Several hardware registers control DMA sound output as follows:

Address	Bit Layout	Meaning
0xFF8900	`---- ---- ---- --cc`	Sound DMA Control
0xFF8902	`---- ---- 00xx xxxx`	Frame Base Address High (bits 21-16)
0xFF8904	`---- ---- xxxx xxxx`	Frame Base Address Middle (bits 15-8)
0xFF8906	`---- ---- xxxx xxx0`	Frame Base Address Low (bits 7-1)
0xFF8908	`---- ---- 00xx xxxx`	Frame Address Counter (bits 21-16)
0xFF890A	`---- ---- xxxx xxxx`	Frame Address Counter (bits 15-8)
0xFF890C	`---- ---- xxxx xxx0`	Frame Address Counter (bits 7-1)
0xFF890E	`---- ---- 00xx xxxx`	Frame End Address High (bits 21-16)
0xFF8910	`---- ---- xxxx xxxx`	Frame End Address Middle (bits 15-8)
0xFF8912	`---- ---- xxxx xxx0`	Frame End Address Low (bits 7-1)
0xFF8920	`0000 0000 m000 00rr`	Sound Mode Control

Addresses placed in the three groups of address pointer registers must begin on an even address. In addition, only sounds within the first 4 megabytes of memory may be accessed (this limitation has been lifted on the Falcon030). Sounds may not be played from alternate RAM.

Playing a Sound

To begin sound playback, place the start address of the sound in the Frame Base Address registers. Place the address of the end of the sound in the Frame End Address registers. The address of the end of the sound should actually be the first byte in memory past the last byte of the sample.

Set the Sound Mode Control register to the proper value. Bit 7, notated as 'm' should be set to 1 for a monophonic sample or 0 for a stereophonic sample. Bits 0 and 1, notated as 'r', control the sample playback rate as follows:

'r'	Playback Rate
00	6258 Hz
01	12517 Hz
10	25033 Hz
11	50066 Hz

To begin the sample playback, set bits 0 and 1 of the Sound DMA Control register, notated as 'c', as follows:

'c'	Sound Control
00	Sound Disabled (this will stop any sound currently being played)
01	Sound Enabled (play once)
11	Sound Enabled (repeat until stopped)

Sound playback may be prematurely halted by writing a 0 to address 0x00FF8900.

Sound Interrupts using MFP Timer A

Discontinuous sample frames may be linked together using the MFP Timer A interrupt. When a sound is played using repeat mode an interrupt is generated at the end of every frame. By configuring Timer A to 'event count' mode you can ensure the seamless linkage and variable repeating of frames.

For example, suppose you have three sample frames, A, B, and C, in memory and you want to play A five times, B five times, and C only once. Use the following steps to properly configure Timer A and achieve the desired result:

- Use **Xbtimer()** to set Timer A to event count mode with a data value of 4 (the first data value should be one less than actually desired since the sound will play once before the interrupt occurs).
- Configure the sound registers as desired and start sound playback in repeat mode.
- When the interrupt fires, place the address of frame B in the sound playback registers (these values aren't actually used until the current frame finishes).
- Reset Timer A's data register to 5 and exit your interrupt handler.

- When the second interrupt fires, place the address of frame C in the sound playback registers.
- Reset Timer A's data register to 1 and exit your interrupt handler.
- When the final interrupt is triggered, write a 0x01 to the sound control register to cause sound playback to end at the end of the current frame.

Sound Interrupts using GPIP 7

Another method of generating interrupts at the end of sound frames is by using the MFP's General Purpose Interrupt Port (GPIP) 7. This interrupt does not support an event count mode so it will generate an interrupt at the end of every frame. In addition, the interrupt must be configured differently depending on the type on monitor connected to the system (this is because GPIP 7 serves double-duty as the monochrome detect signal).

To program GPIP 7 for interrupts, disable all DMA sound by placing a 0x00 in the sound control register. Next, check bit 7 of the GPIP port at location 0xFFFA01. If a monochrome monitor is connected the bit will be 0. The bit will be 1 if a color monitor is connected.

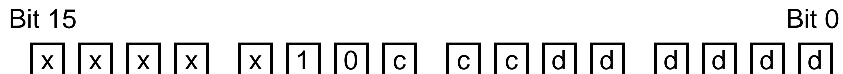
Bit 7 of the MFP's active edge register (at 0xFFFA03) should be set to the opposite of the GPIP port's bit 7. This will cause an interrupt to trigger at the end of every frame. Use **Mfpint()** to set the location of your interrupt handler and **Jenabint()** to enable interrupts. From this point, interrupts will be generated at the end of every frame playing in 'play once' mode or repeat mode until the interrupt is disabled.

The MICROWIRE Interface

The STe and TT030 computers use the MICROWIRE interface to control volume, mixing of the PSG and DMA output, and tone control. The original ST is limited to amplitude control through the use of the appropriate PSG register. The Falcon030 supports new **XBIOS** calls which allow volume and mixing control.

The MICROWIRE interface is a write-only device accessed using two hardware registers 0xFFFF8924 (mask) and 0xFFFF8922 (data). To write a command to the MICROWIRE you must first place the value 0x07FF into the mask register and then write the appropriate command to the data register. The format for the data **WORD** is shown below:

Bit 15 Bit 0



Bits labeled 'x' will be ignored. Bits 9 and 10 should always be %10 to correctly specify the device address which is a constant. Bits labeled 'c' specify the command and bits labeled 'd' contain the appropriate data for the command. The following table explains the valid MICROWIRE commands:

Command	'ccc'	'dddddd'	
Set Master Volume	011	**Example Value**	**Result**
		%000000	-80dB Attenuation
		%010100	-40dB Attenuation
		%101000	0dB Attenuation (Maximum)
Set Left Channel Volume	101	**Example Value**	**Result**
		%000000	-40dB Attenuation
		%001010	-20dB Attenuation
		%010100	0dB Attenuation (Maximum)
Set Right Channel Volume	100	**Example Value**	**Result**
		%000000	-40dB Attenuation
		%001010	-20dB Attenuation
		%010100	0dB Attenuation (Maximum)
Set Treble	010	**Example Value**	**Result**
		%000000	-12dB Attenuation
		%000110	0dB Attenuation
		%001100	+12dB Attenuation (Maximum)
Set Bass	001	**Example Value**	**Result**
		%000000	-12dB Attenuation
		%000110	0dB Attenuation
		%001100	+12dB Attenuation (Maximum)
Set PSG/DMA Mix	000	**Example Value**	**Result**
		%000000	-12dB Attenuation
		%000001	Mix PSG sound output.
		%000010	Don't Mix PSG sound output.

When configuring multiple settings at once, you should program a delay between writes since the MICROWIRE takes at least 16µsec to completely read the data register. During a read the MICROWIRE rotates the mask register one bit at a time. You will know a read operation has completed when the mask register returns to 0x07FF. The following assembly segment illustrates this by setting the left and right channel volumes to their maximum values:

```
MWMASK      EQU     $FFFF8924
MWDATA      EQU     $FFFF8922

MASKVAL     EQU     $7FF
HIGHLVOL    EQU     $554
HIGHRVOL    EQU     $514

            .text

maxvol:
            move.w  MASKVAL,MWMASK          ; First write the mask and data values
            move.w  #HIGHLVOL,MWDATA
mwwrite:
            cmp.w   MASKVAL,MWMASK
            bne.s   mwwrite                 ; loop until MWMASK reaches $7FF again
            move.w  #HIGHRVOL,MWDATA        ; ok, safe to write second value
            rts

            .end
```

Video Hardware

Video Resolutions

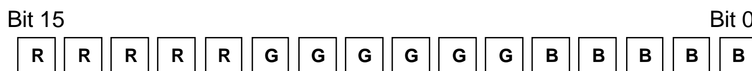
Atari computers support a wide range of video resolutions as shown in the following tables:

Computer System	Modes (width ´ height ´ colors)	Possible Colors
ST, Mega ST	320x200x16 640x200x4 640x400x2	512
STe, Mega STe	320x200x16 640x200x4 640x400x2	4096
STacy	640x400x2	N/A
TT030	320x200x256 640x200x4 640x400x2 320x480x256 640x480x16	4096
Falcon030	See below.	262,144

Falcon030 Video Modes

The Falcon030 is equipped with a much more flexible video controller than earlier Atari computers. The display area may be output on a standard television, an Atari color or monochrome monitor, or a VGA monitor. Overscan is supported with all monitor configurations with the exception of VGA. Also, hardware support for NTSC and PAL monitors is software configurable.

The Falcon030 supports graphic modes of 40 or 80 columns (320 or 640 pixels across) containing 1, 2, 4, 8, or 16 bits per pixel resulting in 2, 4, 16, 256, or 262,144 colors respectively. All modes except the 16 bit per pixel mode supply the video shifter with palette indexes. The 16 bit per pixel mode is a ‘true-color’ mode where each 16 bit value determines the color rather than being an index into a palette. Each 16 bit **WORD** value is arranged as follows:



Falcon030 True-Color Video Word

The ‘R’, ‘G’, and ‘B’, represent the red, green, and blue components of the color. Because red and blue are each allocated five bits, they can represent a color range of 0-31. The green component is allocated six bits so it can represent a color range of 0-63.

The Falcon030 also supports an overlay mode (see **VsetMask()**) where certain colors can be defined as transparent to a connected Genlock (or similar) device. In this mode, the least signifigant green bit (Bit #5) is treated as the transparent flag bit and the resolution of the green

color component is slightly reduced. If the transparent flag bit of a pixel is set, that pixel will display video from the Falcon030’s video shifter, otherwise the external video source will be responsible for its display.

Another feature of the Falcon030’s video shifter is an optional interlace/double-line mode. When operating on a VGA monitor, this mode doubles the pixel height effectively reducing the vertical screen resolution by half. On any other video display, this mode engages interlacing which increases the video resolution.

The operating system calls **VsetMode()** or **VsetScreen()** can be used to manipulate the operating mode of the Falcon030’s video shifter. These calls do not, however, do any checking to ensure the selected video mode is actually attainable on the connected monitor or that the mode is legal. In particular, you should not attempt to set the video shifter to either 40 column mode with only one bit per pixel or 80 column VGA mode with 16 bits per pixel.

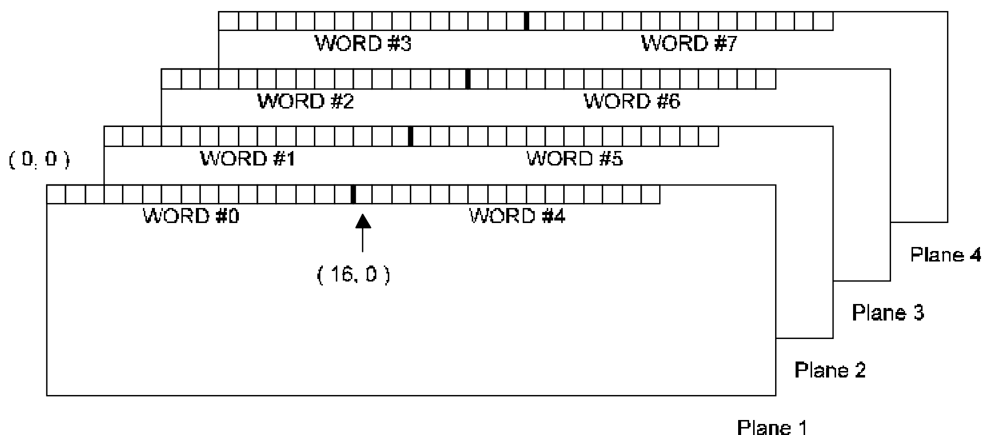
Video Memory

Most of the available video modes are palette-based. The number of bits required per pixel depends on the number of palette entries as shown in the table below. The Falcon030 also offers a true color video mode which requires 16 bits per pixel.

Palette Entries	Bits per Pixel
2	1
4	2
16	4
256	8

Directly accessing video memory is normally not recommended because it may create compatibility problems with future machines and wreak havoc with other system applications. The **VDI** provides a rich set of function calls which should be used when outputting to the screen. The function call **vr_trnfm()**, for instance, can be useful in transforming video images into a pattern compatible with the current video shifter. Certain software, however, does need exclusive access to video memory.

With the exception of the 16-bit true color mode of the Falcon030, all video images are stored in memory in **WORD** interleaved format. The video shifter grabs one at a time from each plane present as shown in the following diagram which represents a 16-color (four plane) screen layout:



The Falcon030's 16-bit true color mode is pixel-packed so that **WORD** #0 in memory is the complete color **WORD** for the pixel at (0, 0), **WORD** #1 is the complete color **WORD** for the pixel at (1, 0), etc.

Fine Scrolling

All Atari computers except the original ST and Mega ST support both horizontal and vertical fine scrolling in hardware. To accomplish this, an application must place a special handler in the vertical blank vector (at 0x00000070) which resets the scroll registers and video base address as needed.

The following registers are manipulated during the vertical-blank period to shift the screen across any number of virtual 'screens':

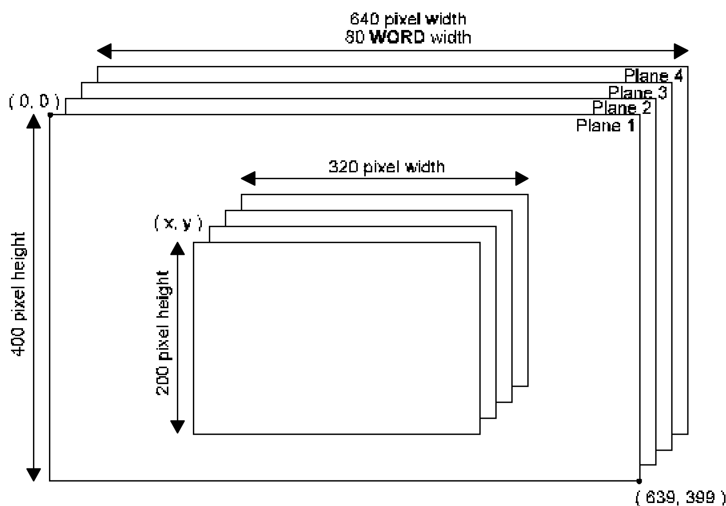
Register	Address	Contents
VBASEHI	0xFFFF8200	Low byte contains bits 23-16 of the video display base address.
VBASEMID	0xFFFF8202	Low byte contains bits 15-8 of the video display base address.
VBASELO	0xFFFF820C	Low byte contains bits 7-0 of the video display base address.
LINEWID	0xFFFF820E	Number of extra **WORD**s per scanline (normally 0).
HSCROLL	0xFFFF8264	Low four bits contain the bitwise offset (0-15) of the screen (normally 0 unless scrolling is in effect).
VCOUNTHI	0xFFFF8204	Low byte contains bits 23-16 of the current video refresh address (use with care).
VCOUNTMID	0xFFFF8206	Low byte contains bits 15-8 of the current video refresh address (use with care).
VCOUNTLO	0xFFFF8208	Low byte contains bits 7-0 of the current video refresh address (use with care).

To accommodate virtual screens wider than the display can show, set ***LINEWID*** to the number of extra **WORD**s per scanline. For instance, to create a virtual display two screens wide for a 320x200 16-color display, set ***LINEWID*** to 80.

To scroll vertically, simply alter the video base address by adding or subtracting the number of **WORD**s per scanline for each line you wish to scroll during the vertical blank.

To scroll horizontally, alter the video base address in **WORD** increments to move the physical screen left and right over the virtual screen. This by itself will cause the screen to skip in 16 pixel jumps. To scroll smoothly, use the ***HSCROLL*** register to shift the display accordingly. When ***HSCROLL*** is non-zero, subtract one from ***LINEWID*** for each plane.

To illustrate this more clearly, imagine a physical screen of 320x200 (16 colors) which is laid out over 4 virtual screens in a 2x2 grid. The following diagram and table shows example values to move the physical screen to the desired virtual coordinates:



Sample Values			
Virtual Coordinates	***VBASE* Address**	***LINEWID***	***HSCROLL***
(0, 0)	0x80000	80	0
(16, 0)	0x80004	80	0
(0, 1)	0x80140	80	0
(1, 0)	0x80000	76	1
(0, 10)	0x80B40	80	0
(100, 100)	0x87BE4	76	4