XBIOS Function Reference

Bconmap()

LONG Bconmap(***devno*** **)**
WORD ***devno*****;**

Bconmap() maps a serial device to **BIOS** device #1. It is also used to add serial device drivers to the system.

OPCODE 44 (0x2C)

AVAILABILITY To reliably check that **Bconmap()** is supported, the **TOS** version must be 1.02 or higher and the following function should return a **TRUE** value.

```
#define BMAP_EXISTS  0

BOOL IsBconmap( VOID )
{
       return (Bconmap(0) == BMAP_EXISTS);
}
```

PARAMETERS The value of *devno* has the following effect:

Name	*devno*	Meaning
BMAP_CHECK	0	Verify the existence of the call (systems without **Bconmap()** will return the function opcode 44).
—	1-5	These are illegal values (will return 0).
*See **XBIOS** Serial Port Mapping for constants.*	6-	Redefine **BIOS** device 1 (the **GEMDOS** 'aux:' device) to map to the named serial device. All **Bcon...(1,...)**, **Rsconf()**, and **Iorec()** calls will return information for the named device. Returns the old value.
BMAP_INQUIRE	-1	Don't change anything, simply return the old value.
BMAP_MAPTAB	-2	Return a pointer to the serial device vector table (see below).

BINDING

```
move.w      devno,-(sp)
move.w      #$2C,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE See above.

CAVEATS You should never install the 38th device (**BIOS** device number 44). It would be indistinguishable from the case where **Bconmap()** was unavailable. In the unlikely event that this case arises, you should install two new devices and assign your new device to the second one.

All current versions of Falcon030 **TOS** (4.00 – 4.04) contain a bug that prevents

the **BIOS** from accessing the extra available devices. A patch program named FPATCH2.PRG is available from Atari Corporation to correct this bug in software.

COMMENTS To add a serial device to the table, use **Bconmap(-2)** to return a pointer to a **BCONMAP** structure. *maptab* points to a list of **MAPTAB** structures (the first entry in **MAPTAB** is the table for device number 6). The list will contain *maptabsize* devices. Allocate a block of memory large enough to store the old table plus your new entry and copy the old table and your new device structure there making sure to increment *maptabsize*. Finally, alter *maptab* to point to your new structure.

```
typedef struct
{
       WORD  (*Bconstat)();
       LONG  (*Bconin)();
       LONG  (*Bcostat)();
       VOID  (*Bconout)();
       ULONG (*Rsconf)();
       IOREC *iorec;          /* See Iorec() */
} MAPTAB;

typedef struct
{
       MAPTAB    *maptab;
       WORD      maptabsize;
} BCONMAP;
```

SEE ALSO **Bconin(), Bconout(), Rsconf(), Iorec()**

Bioskeys()

VOID Bioskeys(VOID)

Bioskeys() is used to reset to the power-up defaults of the keyboard configuration tables.

OPCODE 24 (0x18)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w       #$18,-(sp)
trap         #14
addq.l       #4,sp
```

COMMENTS This call is only necessary to restore changes made by modifying the tables given by **Keytbl()**.

SEE ALSO **Keytbl()**

Blitmode()

WORD Blitmode(*mode*)
WORD *mode*;

Blitmode() detects a hardware BLiTTER chip and can alter its configuration if present.

OPCODE 64 (0x40)

AVAILABILITY This call is available as of **TOS** 1.02.

PARAMETERS *mode* is used to set the BLiTTER configuration. If *mode* is **BLIT_INQUIRE** (-1), the call will return the current state of the BLiTTER without modifying its state. To change the method of OS blit operations, call **Blitmode()** with one of the following values:

Name	*mode*	Meaning
BLIT_SOFT	0	If set, use hardware BLiTTER chip, otherwise use software routines.
BLIT_HARD	1	If set, hardware BLiTTER chip is available.

BINDING

```
move.w      mode,-(sp)
move.w      #$40,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Blitmode()** returns the old *mode* value. Bit #0 of *mode* contains the currently set blitter mode as shown above. Bit #1 is set to indicate the presence of a hardware blitter chip or clear if no blitter chip is installed.

COMMENTS You should use this call once to verify the existence of the BLiTTER prior to attempting to change its configuration.

Buffoper()

LONG Buffoper(*mode*)
WORD *mode*;

Buffoper() sets/reads the state of the hardware sound system.

OPCODE 136 (0x88)

AVAILABILITY Available if '_SND' cookie has third bit set.

PARAMETERS *mode* is a bit array which may be composed of all or none of the following flags indicating the desired sound system state as follows:

Name	Bit Mask	Meaning
PLAY_ENABLE	0x01	Enable DMA Sound Playback. The sound must have been previously identified to the **XBIOS** with the **Buffptr()** function.
PLAY_REPEAT	0x02	Setting this flag will cause any sound currently playing or started as a result of this call to be looped indefinitely (until **Buffoper(**0**)** is used).
RECORD_ENABLE	0x04	Enable DMA Sound Recording. The sound must have been previously identified to the **XBIOS** with the **Buffptr()** function.
RECORD_REPEAT	0x08	Setting this flag during a record will cause the recording to continue indefinitely within the currently set recording buffer (as set by **Buffptr()**)

Alternately, calling this function with a *mode* parameter of **SND_INQUIRE** (-1) will return a bit mask indicating the current sound system state as shown above.

BINDING

```
move.w      mode,-(sp)
move.w      #$88,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Buffoper()** normally returns 0 for no error or non-zero otherwise (except in inquire mode as indicated above.

COMMENTS The sound system uses a 32 bit FIFO. The FIFO is only guaranteed to be clear when the record enable bit is clear. When transferring new data to the record buffers, the record enable bit should be cleared to flush the FIFO.

SEE ALSO **Setbuffer()**

Buffptr()

LONG Buffptr(*sptr*)
**SBUFPTR ***sptr*;

Buffptr() returns the current position of the playback and record pointers.

OPCODE 141 (0x8D)

AVAILABILITY Available if '_SND' cookie has third bit set.

PARAMETER *sptr* is a pointer to a **SBUFPTR** structure which is filled in with the current pointer values. **SBUFPTR** is defined as follows:

```
typedef struct
{
       VOIDP playptr;
       VOIDP recordptr;
       VOIDP reserved1;
       VOIDP reserved2;
} SBUFPTR;
```

BINDING

```
pea          sptr
move.w       #$8d,-(sp)
trap         #14
addq.l       #6,sp
```

RETURN VALUE **Buffptr()** returns 0 if the operation was successful or non-zero otherwise.

SEE ALSO **Setbuffer(), Buffoper()**

Cursconf()

WORD Cursconf(***mode, rate*** **)**
WORD ***mode, rate*****;**

Cursconf() configures the VT-52 cursor.

OPCODE 21 (0x15)

AVAILABILITY All **TOS** versions.

PARAMETERS *mode* defines the operation as follows:

Name	*mode*	Meaning
CURS_HIDE	0	Hide cursor.
CURS_SHOW	1	Show cursor.
CURS_BLINK	2	Enable cursor blink.
CURS_NOBLINK	3	Disable cursor blink.
CURS_SETRATE	4	Set blink rate to *rate.*
CURS_GETRATE	5	Return current blink rate.

BINDING

```
move.w       rate,-(sp)
move.w       mode,-(sp)
move.w       #$15,-(sp)
```

```
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Cursconf()** only returns a meaningful value under *mode* 5 in which it returns the current blink rate.

COMMENTS The blink rate is specified in number of vertical blanks per blink.

Dbmsg()

VOID Dbmsg(***rsrvd, msg_num, msg_arg*** **)**
WORD ***rsrvd, msg_num*****;**
LONG ***msg_arg*****;**

Dbmsg() allows special debugging messages to be sent to a resident debugger application.

OPCODE 11 (0x0B)

AVAILABILITY The only debugger that currently supports this call is the Atari Debugger.

PARAMETERS *rsrvd* is currently reserved and should always be 5. *msg_num* is the message number which you want to send to the debugging host. Values of 0x0000 to 0xEFFF are reserved for applications to define. Values of 0xF000 to 0xFFFF are reserved for special debugging messages.

If *msg_num* is in the application defined range, it and the **LONG** contained in *msg_arg* will be displayed by the debugger and the application will be halted.

If *msg_num* is between 0xF001 and 0xF0FF inclusive then *msg_arg* is interpreted as a character pointer pointing to a string to be output by the debugger and debugging to halt. The string length is determined by the low byte of *msg_num*. If *msg_num* is **DB_NULLSTRING** (0xF000), the string will be output until a **NULL** is reached.

If *msg_num* is **DB_COMMAND** (0xF100), *msg_arg* is interpreted as a character pointer to a string containing a debugger command. The command format is specific to the debugger which you are running.

A useful example of this format when running under the Atari debugger allows a string to be output to the debugger without terminating debugging as shown in the following example:

```
Dbmsg( 5, DB_COMMAND, “echo ‘Debugging Message’;g” );
```

BINDING

```
move.l      msg_arg,-(sp)
move.w      msg_num,-(sp)
move.w      #$5,-(sp)
move.w      #$0B,-(sp)
trap        #14
lea         10(sp),sp
```

COMMENTS

The Atari Debugger only understands the value **DB_COMMAND** (0xF100) for *msg_num* as of version 3.

Though it is normally harmless to run an application with embedded debugging messages when no debugger is present in the system, distribution versions of applications should have these instructions removed.

Devconnect()

LONG Devconnect(***source, dest, clk, prescale, protocol*** **)**
WORD ***source, dest, clk, prescale, protocol*****;**

Devconnect() attaches a source device in the sound system to one or multiple destination devices through the use of the connection matrix.

OPCODE

139 (0x8B)

AVAILABILITY

Available if '_SND' cookie has third bit set.

PARAMETERS

source indicates the source device to connect as follows:

Name	*source*	Meaning
DMAPLAY	0	DMA Playback
DSPXMIT	1	DSP Transmit
EXTINP	2	External Input
ADC	3	Microphone/Yamaha PSG

dest is a bit mask which is used to choose which destination devices to connect as follows:

Name	Mask	Meaning
DMAREC	0x01	DMA Record
DSPRECV	0x02	DSP Receive
EXTOUT	0x04	External Out
DAC	0x08	DAC (Headphone or Internal Speaker)

clk is the clock the source device will use as follows:

Name	*clk*	Meaning
CLK_25M	0	Internal 25.175 MHz clock
CLK_EXT	1	External clock
CLK_32M	2	Internal 32 MHz clock

prescale chooses the source clock prescaler. Sample rate is determined by the formula:

$$rate = \frac{clockrate / 256}{prescale + 1}$$

Valid prescaler values for the internal CODEC using the 25.175 MHz clock are:

Name	*prescale*	Meaning/Sample Rate
CLK_COMPAT	0	TT030/STe compatiblity mode. Use prescale value set with **Soundcmd()**.
CLK_50K	1	49170 Hz
CLK_33K	2	32880 Hz
CLK_25K	3	24585 Hz
CLK_20K	4	19668 Hz
CLK_16K	5	16390 Hz
CLK_12K	7	12292 Hz
CLK_10K	9	9834 Hz
CLK_8K	11	8195 Hz

protocol sets the handshaking mode. A value of **HANDSHAKE** (0) enables handshaking, **NO_SHAKE** (1) disables it. When transferring sound or video data through the CODEC it is usually recommended that handshaking be disabled. When incoming data must be 100% error free, however, handshaking should be enabled.

BINDING

```
move.w      protocol,-(sp)
move.w      prescale,-(sp)
move.w      clk,-(sp)
move.w      dest,-(sp)
move.w      source,-(sp)
move.w      #$8B,-(sp)
trap        #14
lea         12(sp),sp
```

RETURN VALUE

Devconnect() returns 0 if the operation was successful or non-zero otherwise.

CAVEATS

Setting the prescaler to an invalid value will result in a mute condition.

SEE ALSO **Soundcmd()**

DMAread()

LONG DMAread(***sector, count, buf, dev*** **)**
LONG ***sector*****;**
WORD ***count*****;**
VOIDP ***buf*****;**
WORD ***dev*****;**

DMAread() reads raw sectors from a ACSI or SCSI device.

OPCODE 42 (0x2A)

AVAILABILITY This call is available as of **TOS** version 2.00.

PARAMETERS *sector* specifies the sector number to begin reading at. *count* specifies the number of sectors to read. *buf* is a pointer to the address where incoming data will be stored. *dev* specifies the device to read from as follows:

dev	Meaning
0-7	ACSI devices 0-7
8-15	SCSI devices 0-7

BINDING

```
move.w      dev,-(sp)
pea         buf
move.w      count,-(sp)
move.l      sector,-(sp)
move.w      #$2A,-(sp)
trap        #14
lea         14(sp),sp
```

RETURN VALUE **DMAread()** returns 0 if the operation was successful or a negative **BIOS** error code otherwise.

CAVEATS SCSI devices will write data until the device exits its data transfer phase. Since this call is not dependent on sector size, you should ensure that the buffer is large enough to hold sectors from devices with large sectors (CD-ROM = 2K, for example).

COMMENTS ACSI transfers must be done to normal RAM. If you need to read sectors into alternative RAM, use the 64KB pointer found with the '_FRB' cookie as an intermediate transfer point while correctly managing the '*_flock*' system variable.

SCSI transfers on the TT030 do not actually use DMA. Handshaking is used to

transfer bytes individually. This means that alternative RAM may be used. The Falcon030 uses DMA for SCSI transfers making transfers to alternative RAM illegal.

SEE ALSO **DMAwrite(), Rwabs()**

DMAwrite()

LONG DMAwrite(***sector, count, buf, dev*** **)**
LONG ***sector*****;**
WORD ***count*****;**
VOIDP ***buf*****;**
WORD ***dev*****;**

DMAwrite() writes raw sectors to ACSI or SCSI devices.

OPCODE 43 (0x2B)

AVAILABILITY **TOS** versions >= 2.00

PARAMETERS *sector* is the starting sector number to write data to. *count* is the number of sectors to write. *buf* defines the starting address of the data to write. *dev* is the device number as specified in **DMAread()**.

BINDING

```
move.w      dev,-(sp)
pea         buf
move.w      count,-(sp)
move.l      sector,-(sp)
move.w      #$2B,-(sp)
trap        #14
lea         14(sp),sp
```

RETURN VALUE **DMAwrite()** returns 0 if successful or a negative **BIOS** error code otherwise.

COMMENTS ACSI transfers must be done from normal RAM. If you need to read sectors into alternative RAM, use the 64KB pointer found with the '_FRB' cookie as an intermediate transfer point while correctly managing the '*_flock*' system variable.

SCSI transfers do not actually use DMA. Handshaking is used to transfer bytes individually.

SEE ALSO **DMAread(), Rwabs()**

Dosound()

VOID Dosound(*cmdlist*)
char ***cmdlist*;

Dosound() initializes and starts an interrupt driven sound playback routine using the PSG.

OPCODE 32 (0x20)

AVAILABILITY All **TOS** versions.

PARAMETERS If *cmdlist* is positive, it will be interpreted as a pointer to a character array containing a sequential list of commands required for the sound playback. Each command is executed in order and has a meaning as follows:

Command Byte	Meaning
0x00 - 0x0F	Select a PSG register (the register number is the command byte). The next byte in the list will be loaded into this register. See **Appendix I** for a detailed listing of registers, musical frequencies, and sound durations.
0x80	Store the next byte in a temporary register for use by command 0x81.
0x81	Three bytes follow this command. The first is the PSG register to load with the value in the temporary register (set with command 0x80). The second is a signed value to add to the temporary register until the value in the third byte is met.
0x82	If a 0 follows this command, this signals the end of processing, otherwise the value indicates the number of 50Hz ticks to wait until the processing of the next command.

Passing the value **DS_INQUIRE** (-1) for *cmdlist* will cause the pointer to the current sound buffer to be returned or **NULL** if no sound is currently playing.

BINDING

```
pea         cmdlist
move.w      #$20,-(sp)
trap        #14
addq.l      #6,sp
```

CAVEATS This routine is driven by interrupts. Do not use an array created on the stack to store the command list that may go out of scope before the sound is complete.

This function will cause the OS to crash under **MultiTOS** versions prior to 1.08 if every running application is not set to ‘Supervisor’ or ‘Global’ memory protection.

Dosound(DS_INQUIRE) will cause the OS to crash under **MultiTOS** versions 1.08 and below.

Dsp_Available()

VOID Dsp_Available(*xavail, yavail*)
LONG **xavail, *yavail*;

Dsp_Available() returns the amount of free program space in X and Y DSP memory.

OPCODE 106 (0x6A)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS Upon return, the longwords pointed to by *xavail* and *yavail* will contain the length of memory (in bytes) available for DSP programs and subroutines.

BINDING

```
pea         yavail
pea         xavail
move.w      #$6A,-(sp)
trap        #14
lea         10(sp),sp
```

SEE ALSO **Dsp_Reserve()**

Dsp_BlkBytes()

VOID Dsp_BlkBytes(*data_in, size_in, data_out, size_out*)
UBYTE **data_in*;
LONG *size_in*;
UBYTE **data_out*;
LONG *size_out*;

Dsp_BlkBytes() transfers a block of unsigned character data to the DSP and returns the output from the running program or subroutine.

OPCODE 124 (0x7C)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to an unsigned character array which is transferred to the DSP. *size_in* is the length (in bytes) of data to transfer.

data_out is a pointer to the unsigned character array to be filled in from the low byte of the DSP's transfer register. *size_out* is the length (in bytes) of the output buffer array.

BINDING

```
move.l      size_out,-(sp)
pea         data_out
move.l      size_in,-(sp)
pea         data_in
move.w      #$7C,-(sp)
trap        #14
lea         18(sp),sp
```

CAVEATS No handshaking is performed with this call. Error sensitive data should be transferred with **Dsp_BlkHandShake()**.

COMMENTS Bytes are not sign extended before transfer. Also, due to the length of static memory in the DSP, *size_in* and *size_out* should not exceed 65536.

SEE ALSO **Dsp_BlkWords()**

Dsp_BlkHandShake

VOID Dsp_BlkHandShake(*data_in*, *size_in*, *data_out*, *size_out*)
char ****data_in*;**
LONG *size_in*;
char ****data_out*;**
LONG *size_out*;

Dsp_BlkHandShake() handshakes a block of bytes to the DSP and returns the output generated by the running subroutine or program.

OPCODE 97 (0x61)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to data being sent to the DSP. *size_in* specifies the number of DSP words of data to be transferred. **Dsp_GetWordSize()** can be used to determine the number of bytes that occur for a DSP word.

data_out is a pointer to the buffer to which processed data will be returned from the DSP. *size_out* indicates the number of DSP words to transfer.

BINDING

```
move.l      size_out,-(sp)
pea         data_out
move.l      size_in,-(sp)
pea         data_in
move.w      #$61,-(sp)
trap        #14
lea         18(sp),sp
```

COMMENTS **Dsp_BlkHandshake()** is identical to **Dsp_DoBlock()**, however, this function handshakes each byte to prevent errors in sensitive data.

SEE ALSO **Dsp_DoBlock()**

Dsp_BlkUnpacked()

VOID Dsp_BlkUnpacked(***data_in*, *size_in*, *data_out*, *size_out*** **)**
LONG *da*ta_in*;
LONG *size_in*;
LONG ***data_out*;
LONG *size_out*;

Dsp_BlkUnpacked() transfers data to the DSP from a longword array. Data processed by the running subroutine or program is returned.

OPCODE 98 (0x62)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to an array of **LONG**s from which data is transferred to the DSP. As many bytes are transferred from each **LONG** as there are bytes in a DSP **WORD**. For example, if **Dsp_GetWordSize()** returns 3, the lower three bytes of each **LONG** are transferred into each DSP **WORD**.

size_in represents the number of **LONG**s in the array to transfer. *data_out* is a pointer to an array of **LONG**s *size_out* in length in which data sent from the DSP is returned.

BINDING

```
move.l      size_out,-(sp)
pea         data_out
move.l      size_in,-(sp)
pea         data_in
move.w      #$62,-(sp)
trap        #14
lea         18(sp),sp
```

CAVEATS This function only works with DSP's which return 4 or less from **Dsp_GetWordSize()**. In addition, no handshaking is performed with this call. Data which is sensitive to errors should use **Dsp_BlkHandShake()**.

SEE ALSO **Dsp_DoBlock()**

Dsp_BlkWords()

VOID Dsp_BlkWords(***data_in*****,** ***size_in*****,** ***data_out*****,** ***size_out*** **)**
WORD ****data_in***;**
LONG ***size_in*****;**
WORD ****data_out***;**
LONG ***size_out*****;**

Dsp_BlkWords() transfers an array of **WORD**s to the DSP and returns the output generated by the running subroutine or program.

OPCODE 123 (0x7B)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to the **WORD** array to be transferred to the DSP. *size_in* is the length (in **WORD**s) of data to transfer.

data_out is a pointer to the **WORD** array to be filled in during the data output phase of the DSP from the middle and low bytes of the transfer register. *size_out* is the length (in **WORD**s) of the buffer for the output array.

BINDING

```
move.l      size_out,-(sp)
pea         data_out
move.l      size_in,-(sp)
pea         data_in
move.w      #$7B,-(sp)
trap        #14
lea         18(sp),sp
```

CAVEATS No handshaking is performed with this call. Data which is sensitive to errors should use **Dsp_BlkHandShake()**.

COMMENTS **WORD**s are sign extended before transfer. Also, due to the length of static memory in the DSP, *size_in* and *size_out* should not exceed 32768.

SEE ALSO **Dsp_BlkBytes()**

Dsp_DoBlock()

VOID Dsp_DoBlock(***data_in*****,** ***size_in*****,** ***data_out*****,** ***size_out*** **)**
char ****data_in*;**
LONG ***size_in***;
char ****data_out*;**
LONG ***size_out***;

Dsp_DoBlock() transfers bytewise packed data to the DSP and returns the data processed by the running subroutine or program.

OPCODE 96 (0x60)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a character array containing data to transfer to the DSP. *size_in* specifies the number of DSP words to transfer. For example, if **Dsp_GetWordSize()** returns 3, the first 3 bytes from *data_in* are stored in the first DSP word, the next 3 bytes are stored in the next DSP word and so on.

data_out points to a character array where the output will be stored in a similar manner. *size_out* represents the size of this array.

BINDING

```
move.l      size_out,-(sp)
pea         data_out
move.l      size_in,-(sp)
pea         data_in
move.w      #$60,-(sp)
trap        #14
lea         18(sp),sp
```

CAVEATS No handshaking is performed with this call. Data which is sensitive to errors should use **Dsp_BlkHandShake()**.

SEE ALSO **Dsp_BlkHandShake()**

Dsp_ExecBoot()

VOID Dsp_ExecBoot(***codeptr, codesize, ability*** **)**
char ****codeptr*;**
LONG ***codesize***;
WORD ***ability***;

Dsp_ExecBoot() completely resets the DSP and loads a new bootstrap program into the first 512 DSP words of memory.

OPCODE 110 (0x6E)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *codeptr* points to the beginning of the DSP program data to be transferred. *codesize* indicates the size (in DSP words) of program data to transfer. *ability* indicates the bootstrapper's unique ability code.

BINDING

```
move.w      ability,-(sp)
move.l      codesize,-(sp)
pea         codeptr
move.w      #$6E,-(sp)
trap        #14
lea         12(sp),sp
```

COMMENTS This call is only designed for special development and testing purposes. Use of this call takes over control of the DSP system.

This call is limited to transferring up to 512 DSP words of code.

SEE ALSO **Dsp_LoadProg(), Dsp_ExecProg()**

Dsp_ExecProg()

VOID Dsp_ExecProg(***codeptr, codesize, ability*** **)**
char ****codeptr*;**
LONG ***codesize***;
WORD ***ability***;

Dsp_ExecProg() transfers a DSP program stored in binary format in memory to the DSP and executes it.

OPCODE 109 (0x6D)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *codeptr* points to the start of the binary program in memory. *codesize* indicates the number of DSP words to transfer. *ability* indicates the program's unique ability code.

BINDING

```
move.w      ability,-(sp)
move.l      codesize,-(sp)
pea         codeptr
move.w      #$6D,-(sp)
trap        #14
lea         12(sp),sp
```

COMMENTS *codesize* should not exceed the amount of memory reserved by the **Dsp_Reserve()** call.

SEE ALSO **Dsp_LoadProg(), Dsp_Reserve()**

Dsp_FlushSubroutines()

VOID Dsp_FlushSubroutines(VOID)

Dsp_FlushSubroutines() removes all subroutines from the DSP.

OPCODE 115 (0x73)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$73,-(sp)
trap        #14
addq.l      #2,sp
```

COMMENTS This call should only be used when a program requires more memory than is returned by **Dsp_Available()**.

SEE ALSO **Dsp_Available()**

Dsp_GetProgAbility()

WORD Dsp_GetProgAbility(VOID)

Dsp_GetProgAbility() returns the current ability code for the program currently residing in DSP memory.

OPCODE 114 (0x72)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$72,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_GetProgAbility()** returns the **WORD** ability code for the current program loaded in the DSP.

COMMENTS If you know the defined ability code of the program you wish to use, you can use this call to see if the program already exists on the DSP and avoid reloading it.

SEE ALSO **Dsp_InqSubrAbility()**

Dsp_GetWordSize()

WORD Dsp_GetWordSize(VOID)

Dsp_GetWordSize() returns the size of a DSP word in the installed Digital Signal Processor.

OPCODE 103 (0x67)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$67,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_GetWordSize()** returns the number of bytes per DSP word.

COMMENTS This value is useful with many DSP-related **XBIOS** calls to provide upward compatibility as the DSP hardware is not guaranteed to remain the same.

Dsp_Hf0()

WORD Dsp_Hf0(***flag*** **)**
WORD *flag*;

Dsp_Hf0() reads/writes to bit #3 of the HSR.

OPCODE 119 (0x77)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *flag* has three legal values as follows:

Name	*flag*	Meaning
HF_CLEAR	0	Clear bit #3 of the DSP's HSR.
HF_SET	1	Set bit #3 of the DSP's HSR.
HF_INQUIRE	-1	Return the current value of bit #3 of the DSP's HSR.

BINDING

```
move.w      flag,-(sp)
move.w      #$77,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE If *flag* is **HF_INQUIRE** (-1), **Dsp_Hf0()** returns the current state of bit #3 of the HSR register.

SEE ALSO **Dsp_Hf1()**

Dsp_Hf1()

WORD Dsp_Hf1(*flag*)
WORD *flag*;

Dsp_Hf1() reads/writes to bit #4 of the HSR.

OPCODE 120 (0x78)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *flag* has three legal values as follows:

Name	*flag*	Meaning
HF_CLEAR	0	Clear bit #4 of the DSP's HSR.
HF_SET	1	Set bit #4 of the DSP's HSR.
HF_INQUIRE	-1	Return the current value of bit #4 of the DSP's HSR.

BINDING

```
move.w      flag,-(sp)
move.w      #$78,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE If *flag* is **HF_INQUIRE** (-1), **Dsp_Hf1()** returns the current state of bit #4 of the HSR register.

SEE ALSO **Dsp_Hf0()**

Dsp_Hf2()

WORD Dsp_Hf2(VOID)

Dsp_Hf2() returns the current status of bit #3 of the DSP's HCR.

OPCODE 121 (0x79)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$79,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_Hf2()** returns the current setting of bit #3 of the HCR register (valid values are 0 or 1).

SEE ALSO **Dsp_Hf3()**

Dsp_Hf3()

WORD Dsp_Hf3(VOID)

Dsp_Hf3() returns the current status of bit #4 of the DSP's HCR.

OPCODE 122 (0x7A)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$7A,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_Hf3()** returns the current setting of bit #4 of the HCR register (valid values are 0 or 1).

SEE ALSO **Dsp_Hf2()**

Dsp_HStat()

BYTE Dsp_Hstat(VOID)

Dsp_HStat() returns the value of the DSP's ICR register.

OPCODE 125 (0x7D)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$7D,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_Hstat()** returns an 8-bit value representing the current state of the DSP's ICR register as follows:

Name	Bit	Meaning
ICR_RXDF	0	ISR Receive data register full (RXDF)
ICR_TXDE	1	ISR Transmit data register empty (TXDE)
ICR_TRDY	2	ISR Transmitter ready (TRDY)
ICR_HF2	3	ISR Host flag 2 (HF2)
ICR_HF3	4	ISR Host flag 3 (HF3)
—	5	Reserved
ICR_DMA	6	ISR DMA Status (DMA)
ICR_HREQ	7	ISR Host Request (HREQ)

Dsp_InqSubrAbility()

WORD Dsp_InqSubrAbility(*ability*)
WORD *ability*;

Dsp_InqSubrAbility() determines if a subroutine with the specified ability code exists in the DSP.

OPCODE 117 (0x75)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *ability* is the ability code you wish to check.

BINDING

```
move.w      ability,-(sp)
move.w      #$75,-(sp)
```

```
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_InqSubrAbility()** returns a handle to the subroutine if found or 0 if not.

SEE ALSO **Dsp_RunSubroutine()**

Dsp_InStream()

VOID Dsp_InStream(***data_in, block_size, num_blocks, blocks_done*** **)**
char ****data_in*;**
LONG ***block_size***;
LONG ***num_blocks***;
LONG ****blocks_done*;**

Dsp_InStream() passes data to the DSP via an interrupt handler.

OPCODE 99 (0x63)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to unsigned character data which should be transferred to the DSP. *block_size* indicates the number of DSP **WORD**s that will be transferred at each interrupt. *num_blocks* indicates the number of blocks to transfer.

The **LONG** pointed to by *blocks_done* will be constantly updated to let the application know the progress of the transfer.

BINDING

```
pea         blocks_done
move.l      num_blocks,-(sp)
move.l      block_size,-(sp)
pea         data_in
move.w      #$63,-(sp)
trap        #14
lea         18(sp),sp
```

CAVEATS No handshaking is performed with this call. If the data you are transmitting is error sensitive, use **Dsp_BlkHandShake()**.

COMMENTS This call is suited for transferring small blocks while other blocks are being prepared for transfer. For larger blocks, **Dsp_DoBlock()** would be more suitable.

SEE ALSO **Dsp_BlkHandShake(), Dsp_DoBlock()**

Dsp_IOStream()

VOID Dsp_IOStream(***data_in, data_out, block_insize, block_outsize, num_blocks, blocks_done*** **)**
char * ***data_in*****, *** ***data_out*****;**
LONG ***block_insize, block_outsize, num_blocks*****;**
LONG * ***blocks_done*****;**

Dsp_IOStream() uses two interrupt handlers to transmit and receive data from the DSP.

OPCODE 101 (0x65)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *data_in* is a pointer to a buffer in which each output block is placed. *data_out* is a pointer to a buffer used to receive each data block from the DSP.

block_insize and *block_outsize* represent the size of the blocks to send and receive, respectively, in DSP **WORD**s. *num_blocks* is the total number of blocks to transfer.

The **LONG** pointed at by *blocks_done* is constantly updated to indicate the number of blocks actually transferred.

BINDING

```
pea         blocks_done
move.l      num_blocks,-(sp)
move.l      block_outsize,-(sp)
move.l      block_insize,-(sp)
pea         data_out
pea         data_in
move.w      #$65,-(sp)
trap        #14
lea         26(sp),sp
```

CAVEATS This call makes the assumption that the DSP will be ready to accept a new block as input every time it finishes sending a block back to the host.

COMMENTS No handshaking is performed with this call. If your data is error-sensitive, you should use **Dsp_BlkHandShake()**.

SEE ALSO **Dsp_InStream(), Dsp_OutStream()**

Dsp_LoadProg()

WORD Dsp_LoadProg(*file*, *ability*, *buf*)
char ****file*;
WORD ***ability*;**
char ****buf*;

Dsp_LoadProg() loads a '.LOD' file from disk, transmits it to the DSP, and executes it.

OPCODE

108 (0x6C)

AVAILABILITY

This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS

file is a pointer to a **NULL**-terminated string containing a valid **GEMDOS** file specification. *ability* is the unique ability code that will be assigned to this program. *buf* should point to a temporary buffer where the DSP will place the binary code it generates. The minimum size of the buffer is determined by the following formula:

```
3 * ( #program/data words + (3 * #blocks in program))
```

BINDING

```
pea         buf
move.w      ability,-(sp)
pea         file
move.w      #$6C,-(sp)
trap        #14
lea         12(sp),sp
```

RETURN VALUE

Dsp_LoadProg() returns a 0 is successful or -1 otherwise.

COMMENTS

Before loading you should determine if a program already exists on the DSP with your chosen ability with **Dsp_GetProgAbility()**.

SEE ALSO

Dsp_LoadSubroutine()

Dsp_LoadSubroutine()

WORD Dsp_LoadSubroutine(*ptr*, *size*, *ability*)
char **ptr*;
LONG *size*;
WORD *ability*;

Dsp_LoadSubroutine() transmits subroutine code to the DSP.

OPCODE 116 (0x74)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *ptr* points to a memory buffer which contains DSP binary subroutine code. *size* is the length of code to transfer (specified in DSP words). *ability* is the **WORD** identifier for the unique ability of this subroutine.

BINDING

```
move.w      ability,-(sp)
move.l      size,-(sp)
pea         ptr
move.w      #$74,-(sp)
trap        #14
lea         12(sp),sp
```

RETURN VALUE **Dsp_LoadSubroutine()** returns the handle assigned to the subroutine or 0 if an error occurred.

COMMENTS DSP subroutines have many restrictions and you should see the previous discussion of the DSP for more information.

SEE ALSO **Dsp_RunSubroutine(), Dsp_InqSubrAbility()**

Dsp_Lock()

WORD Dsp_Lock(VOID)

Dsp_Lock() locks the use of the DSP to the calling application.

OPCODE 104 (0x68)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$68,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_Lock()** returns a 0 if successful or -1 if the DSP has been locked by another application.

COMMENTS **Dsp_Lock()** should be performed before each use of the DSP to prevent other applications from modifying DSP memory or flushing subroutines. A corresponding **Dsp_Unlock()** should be issued at the end of each usage. You should limit the amount of time the DSP is locked so other applications may utilize it.

SEE ALSO **Dsp_Unlock()**

Dsp_LodToBinary()

LONG Dsp_LodToBinary(*file, codeptr*)
char **file,*codeptr*;

Dsp_LodToBinary() reads a '.LOD' file and converts the ASCII data to binary program code ready to be sent to the DSP via **Dsp_ExecProg()** or **Dsp_ExecBoot()**.

OPCODE 111 (0x6F)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *file* is a character pointer to a null-terminated **GEMDOS** file specification. *codeptr* should point to a large enough buffer to hold the resulting binary program code.

BINDING

```
pea         codeptr
pea         file
move.w      #$6F,-(sp)
trap        #14
lea         10(sp),sp
```

RETURN VALUE **Dsp_LodToBinary()** returns the size of the resulting program code in DSP words or a negative error code.

SEE ALSO **Dsp_ExecProg(), Dsp_LoadProg()**

Dsp_MultBlocks()

VOID Dsp_MultBlocks(***numsend, numreceive, sendblks, receiveblks*** **)**
LONG ***numsend, numreceive*****;**
DSPBLOCK ****sendblks, *receiveblks*****;**

Dsp_MultBlocks() transmit and receive multiple blocks of DSP data of varying size.

OPCODE

127 (0x7F)

AVAILABILITY

This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS

numsend and *numreceive* indicate the number of blocks of DSP data to send and receive respectively. *sendblks* and *receiveblks* are both pointers to arrays of type **DSPBLOCK** which contain information for each block. **DSPBLOCK** is defined as follows:

```
typedef struct
{
#define BLOCK_LONG    0
#define BLOCK_WORD    1
#define BLOCK_UBYTE   2
       /* 0 = LONGs, 1 = WORDs, 2 = UBYTEs */
       WORD blocktype;

       /* Num elements in block */
       LONG blocksize;

       /* Start address of block */
       VOIDP blockaddr;
} DSPBLOCK;
```

BINDING

```
pea          receiveblks
pea          sendblks
move.l       numreceive,-(sp)
move.l       numsend,-(sp)
move.w       #$7F,-(sp)
trap         #14
lea          20(sp),sp
```

CAVEATS

No handshaking is performed with this call. To transfer blocks with handshaking use **Dsp_BlkHandShake()**.

Dsp_OutStream()

VOID Dsp_OutStream(***data_out, block_size, num_blocks, blocks_done*** **)**
char * ***data_out*****;**
LONG ***block_size*****;**
LONG ***num_blocks*****;**
LONG * ***blocks_done*****;**

Dsp_OutStream() transfers data from the DSP to a user-specified buffer using interrupts.

OPCODE 100 (0x64)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS This call transfers data from the DSP to the buffer pointed to by *data_out* via an interrupt handler. *block_size* specifies the number of DSP **WORD**s to be transferred and *num_blocks* specifies the number of blocks to transfer.

The **LONG** pointed to by *blocks_done* will be constantly updated by the interrupt handler to indicate the number of blocks successfully transferred. The process is complete when *blocks_done* is equal to *num_blocks*.

BINDING

```
pea         blocks_done
move.l      num_blocks,-(sp)
move.l      block_size,-(sp)
pea         data_out
move.w      #$64,-(sp)
trap        #1
lea         18(sp),sp
```

SEE ALSO **Dsp_DoBlock(), Dsp_MultBlocks(), Dsp_InStream()**

Dsp_RemoveInterrupts()

VOID Dsp_RemoveInterrupts(***mask*** **)**
WORD ***mask*****;**

Dsp_RemoveInterrupts() turns off the generation of DSP interrupts.

OPCODE 102 (0x66)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *mask* is an **WORD** bit mask indicating which interrupts to turn off composed of one or both of the following values:

Name	Mask	Meaning
RTS_OFF	0x01	Disable DSP Ready to Send Interrupts
RTR_OFF	0x02	Disable DSP Ready to Receive Interrupts

BINDING

```
move.w      mask,-(sp)
move.w      #$66,-(sp)
trap        #14
addq.l      #4,sp
```

COMMENTS This call is used to terminate interrupts when an interrupt driven block transfer function does not terminate as expected (this will occur when less than the expected number of blocks is returned) and to shut off interrupts installed by **Dsp_SetVectors()**.

SEE ALSO **Dsp_SetVectors()**

Dsp_RequestUniqueAbility()

WORD Dsp_RequestUniqueAbility(VOID)

Dsp_RequestUniqueAbility() generates a random ability code that is currently not in use.

OPCODE 113 (0x71)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$71,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Dsp_RequestUniqueAbility()** returns a unique ability code to assign to a subroutine or program.

COMMENTS Using this function allows you to call **Dsp_InqSubrAbility()** and **Dsp_GetProgAbility()** to determine if the DSP code your application has already loaded is still present (i.e. has not been flushed by another application).

SEE ALSO **DspInqSubrAbility(), Dsp_GetProgAbility()**

Dsp_Reserve()

WORD Dsp_Reserve(***xreserve, yreserve*** **)**
LONG ***xreserve, yreserve*****;**

Dsp_Reserve() reserves DSP memory for program usage.

OPCODE 107 (0x6B)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *xreserve* and *yreserve* specify the amount of memory (in DSP words) to reserve for a DSP program in X and Y memory space respectively. *xreserve* and *yreserve* must include all program/data space so that subroutines do not overwrite your reserved area.

BINDING

```
move.l      yreserve,-(sp)
move.l      xreserve,-(sp)
move.w      #$6B,-(sp)
trap        #14
lea         10(sp),sp
```

RETURN VALUE **Dsp_Reserve()** returns a 0 if the memory was reserved successfully or -1 if not enough DSP memory was available.

COMMENTS If this call fails you should call **Dsp_FlushSubroutines()** and then retry it. If it fails a second time, the DSP lacks enough memory space to run your program.

Dsp_RunSubroutine()

WORD Dsp_RunSubroutine(***handle*** **)**
WORD ***handle*****;**

Dsp_RunSubroutine() begins execution of the specified subroutine.

OPCODE 118 (0x76)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *handle* is the **WORD** identifier of the DSP subroutine to engage.

BINDING

```
move.w      handle,-(sp)
move.w      #$76,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Dsp_RunSubroutine()** returns a 0 if successful or a negative code indicating failure.

SEE ALSO **Dsp_LoadSubroutine()**

Dsp_SetVectors()

VOID Dsp_SetVectors(*receiver*, *transmitter*)
VOID (receiver*)();**
LONG (transmitter*)();**

Dsp_SetVectors() sets the location of application interrupt handlers that are called when the DSP is either ready to send or receive data.

OPCODE 126 (0x7E)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *receiver* is the address of an interrupt handler which is called when the DSP is ready to send a DSP word of data or **NULLFUNC** (VOID (*)() 0L) if you do not wish to set this interrupt.

Likewise, *transmitter* is a pointer to an interrupt handler which is called when the DSP is ready to receive a DSP word of data or **NULLFUNC** if you do not wish to install a *transmitter* interrupt.

Any function installed to handle *transmitter* interrupts should return a **LONG** which has one of the following values:

Name	*transmitter* Return Value	Meaning
DSPSEND_NOTHING	0x00000000	Do not send any data to the DSP.
DSPSEND_ZERO	0xFF000000	Transmit a DSP word of 0 to the DSP.
—	Any other	Transmit the low 24 bits to the DSP.

BINDING

```
move.l      #transmitter,-(sp)
move.l      #receiver,-(sp)
move.w      #$7E,-(sp)
trap        #14
lea         10(sp),sp
```

COMMENTS Use **Dsp_RemoveInterrupts()** to turn off interrupts set with this call.

SEE ALSO **Dsp_RemoveInterrupts()**

Dsp_TriggerHC()

VOID Dsp_TriggerHC(***vector*** **);**
WORD ***vector*****;**

Dsp_TriggerHC() causes a host command set aside for DSP programs to execute.

OPCODE 112 (0x70)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

PARAMETERS *vector* specifies the vector to execute.

BINDING

```
move.w      vector,-(sp)
move.w      #$70,-(sp)
trap        #14
addq.l      #4,sp
```

CAVEATS Currently vectors 0x13 and 0x14 are the only vectors available for this purpose. All other vectors are overwritten by the system on program load and are used by the system and subroutines.

Dsp_Unlock()

VOID Dsp_Unlock(VOID)

Dsp_Unlock() unlocks the sound system from use by a process which locked it previously using **Dsp_Lock()**.

OPCODE 105 (0x69)

AVAILABILITY This call is only available if the fifth bit of the '_SND' cookie is set.

BINDING

```
move.w      #$69,-(sp)
trap        #14
addq.l      #2,sp
```

SEE ALSO **Dsp_Lock()**

Dsptristate()

LONG Dsptristate(***dspxmit, dsprec*** **)**
WORD ***dspxmit, dsprec*****;**

Dsptristate() connects or disconnects the DSP from the connection matrix.

OPCODE 137 (0x89)

AVAILABILITY Available if '_SND' cookie has bits 3 and 4 set.

PARAMETERS *dpsxmit* and *dsprec* specify whether data being transmitted and/or recorded into the DSP passes through the connection matrix. A value of **DSP_TRISTATE** (0) indicates a 'tristate' condition where data is not fed through the matrix. A value of **DSP_ENABLE** (1) enables the use of the connection matrix.

BINDING

```
move.w      dsprec,-(sp)
move.w      dspxmit,-(sp)
move.w      #$89,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Dsptristate()** returns 0 if no error occurred or non-zero otherwise.

COMMENTS This call is used in conjunction with **Devconnect()** to link the DSP to the internal sound system.

SEE ALSO **Devconnect()**

EgetPalette()

VOID EgetPalette(***start, count, paldata*** **)**
WORD ***start, count*****;**
WORD ****paldata***;**

EgetPalette() copies the current TT030 color palette data into a specified buffer..

OPCODE 85 (0x55)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *start* gives the index (0-255) of the first color register to copy data into. *count* specifies the total number of registers to copy. *paldata* is a pointer to an array where the TT030 palette data will be stored. Each **WORD** will be formatted as

follows:

Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
Reserved	Red	Green	Blue

BINDING

```
pea         palette
move.w      count,-(sp)
move.w      start,-(sp)
move.w      #$55,-(sp)
trap        #14
lea         10(sp),sp
```

CAVEATS This call is machine-dependent to the TT030. It is therefore recommended that **vq_color()** be used in most instances.

COMMENTS Unlike **Setpalette()** this call encodes color nibbles from the most signifigant to least signifigant bit (3-2-1-0) as opposed to the compatibilty method of 0-3-2-1.

SEE ALSO **Esetpalette(), vq_color()**

EgetShift()

WORD EgetShift(VOID)

EgetShift() returns the current mode of the video shifter.

OPCODE 81 (0x51)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

BINDING

```
move.w      #$51,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **EgetShift()** returns a **WORD** bit array which is divided as follows:

Mask Name	Bit(s)	Meaning
ES_BANK	0–3	These bits determine the current color bank being used by the TT (in all modes with less than 256 colors). The macro **ColorBank()** as defined below will extract the current bank code. `#define ColorBank(x) ((x) & ES_BANK)`
—	4–7	Unused

ES_MODE	8–10	These bits determine the current mode of the TT video shifter as follows: **Name** **Value** **ST_LOW** 0x0000 **ST_MED** 0x0100 **ST_HIGH** 0x0200 **TT_MED** 0x0300 **TT_HIGH** 0x0600 **TT_LOW** 0x0700 The current shifter mode code can be extracted with the following macro: `#define ScreenMode(x) ((x) & ES_MODE)`
—	11	Unused
ES_GRAY	12	This bit determines if the TT video shifter is currently in grayscale mode. The following macro can be used to extract this information: `#define IsGrayMode(x) ((x) & ES_GRAY)`
—	13–14	Unused
ES_SMEAR	15	If this bit is set, the TT video shifter is currently in smear mode. The following macro can be used to extract this information: `#define IsSmearMode(x)  ((x) & ES_SMEAR)`

SEE ALSO **EsetGray(), EsetShift(), EsetSmear(), EsetBank()**

EsetBank()

WORD EsetBank(***bank*** **)**
WORD ***bank*****;**

EsetBank() chooses which of 16 banks of color registers is currently active.

OPCODE 82 (0x52)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *bank* specifies the index of the color bank to activate. A value of **ESB_INQUIRE** (-1) does not change anything but still returns the current bank.

BINDING

```
move.w      bank,-(sp)
move.w      #$52,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **EsetBank()** returns the index of the old blank.

CAVEATS This call is machine-dependent to the TT030.

SEE ALSO **EgetShift()**

EsetColor()

WORD EsetColor(*idx, color*)
WORD *idx, color*;

EsetColor() sets an individual color in the TT030's palette.

OPCODE 83 (0x53)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *idx* specifies the color index to modify (0-255). *color* is a TT030 format color **WORD** bit array divided as follows:

Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
Reserved	Red	Green	Blue

If *color* is **EC_INQUIRE** (-1) then the call does not change the register but still returns it value.

BINDING

```
move.w      color,-(sp)
move.w      idx,-(sp)
move.w      #$53,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **EsetColor()** returns the old value of the color register.

CAVEATS This call is machine-dependent to the TT030. It is therefore recommended that **vs_color()** be used instead for compatibility.

COMMENTS Unlike **Setpalette()** this call encodes color nibbles from the most signifigant to least signifigant bit (3-2-1-0) as opposed to the compatibilty method of 0-3-2-1.

SEE ALSO **EsetPalette(), vs_color()**

EsetGray()

WORD EsetGray(*mode*)
WORD *mode*;

EsetGray() reads/modifies the TT030's video shifter gray mode bit.

OPCODE 86 (0x56)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *mode* is defined as follows:

Name	*mode*	Meaning
ESG_INQUIRE	-1	Return the gray bit of the video shifter.
ESG_COLOR	0	Set the video shifter to interpret the lower 16 bits of a palette entry as a TT030 color value (RGB 0-15).
ESG_GRAY	1	Set the video shifter to interpret the lower 8 bits of a palette entry as a TT030 gray value (0-255)

BINDING

```
move.w      mode,-(sp)
move.w      #$56,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **EsetGray()** returns the previous value of the video shifter's gray bit.

CAVEATS This call is machine-dependent to the TT030.

SEE ALSO **EgetShift()**

EsetPalette()

VOID EsetPalette(*start, count, paldata*)
WORD *start,count*;
WORD **paldata*;

EsetPalette() copies TT030 color **WORD**s from the specified buffer into the TT030 Color Lookup Table (CLUT).

OPCODE 84 (0x54)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *start* specifies the index of the starting color register to copy color data to. *count* indicates the number of palette **WORD**s to copy. *paldata* is a pointer to an array of palette **WORD**s to copy.

BINDING

```
pea         palette
move.w      count,-(sp)
move.w      start,-(sp)
move.w      #$54,-(sp)
trap        #14
lea         10(sp),sp
```

CAVEATS This call is machine-dependent to the TT030. It is therefore recommended that **vs_color()** be used instead for compatibility.

COMMENTS For the format of the color **WORD**s, see **EgetPalette()**.

SEE ALSO **EgetPalette(), vq_color()**

EsetShift()

WORD EsetShift(***mode*** **)**
WORD *mode*;

EsetShift() reads/modifies the TT030 video shifter.

OPCODE 80 (0x50)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *mode* is a **WORD** bit array which defines the new setting of the video shifter as follows:

Name	Bit(s)	Meaning
—	0–3	These bits determine the current color bank being used by the TT (in all modes with less than 256 colors).
—	4–7	Unused
—	8–10	These bits determine the current mode of the TT video shifter as follows: **Name** — **Bit Mask** **ST_LOW** 0x0000 **ST_MED** 0x0100 **ST_HIGH** 0x0200 **TT_MED** 0x0300 **TT_HIGH** 0x0600 **TT_LOW** 0x0700

—	11	Unused
ES_GRAY	12	Setting this bit places the TT video shifter in grayscale mode.
—	13–14	Unused
ES_SMEAR	15	Setting this bit places the TT video shifter in smearsmear mode.

BINDING

```
move.w      mode,-(sp)
move.w      #$50,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **EsetShift()** returns the old *mode* setting of the video shifter.

CAVEATS This call is machine-dependent to the TT030.

SEE ALSO **EgetShift(), EsetGray(), EsetSmear(), EsetBank()**

EsetSmear()

WORD EsetSmear(*mode*)
WORD *mode*;

EsetSmear() reads/modifies the current state of the video shifter's smear mode bit.

OPCODE 87 (0x57)

AVAILABILITY This call is available when the high word of the '_VDO' cookie has a value of 2.

PARAMETERS *mode* specifies the action of this call as follows:

Name	*mode*	Meaning
ESM_INQUIRE	-1	Return the smear bit of the video shifter.
ESM_NORMAL	0	Set the video shifter to process video data normally.
ESM_SMEAR	1	Set the video shifter to repeat the color of the last displayed pixel each time a 0x0000 is read from video memory.

BINDING

```
move.w      mode,-(sp)
move.w      #$57,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **EsetSmear()** returns the prior setting of the video shifter's smear mode bit.

SEE ALSO **Egetshift(), EsetShift()**

Flopfmt()

WORD Flopfmt(***buf, skew, dev, spt, track, side, intlv, magic, virgin*** **)**
VOIDP ***buf*****;**
WORD ****skew*****;**
WORD ***dev, spt, track, side, intlv*****;**
LONG ***magic*****;**
WORD ***virgin*****;**

Flopfmt() formats a specified track on a floppy disk.

OPCODE 10 (0x0A)

AVAILABILITY All **TOS** versions.

PARAMETERS *buf* is a pointer to a word-aligned buffer large enough to hold one disk track which is used to build a copy of each sector to write. *skew* should be **NULL** for non-interleaved sectors or point to a **WORD** array containing *spt* entries which specifies the sector interleave order.

dev specifies which floppy drive to format ('A:' = **FLOP_DRIVEA** (0), 'B:' = **FLOP_DRIVEB** (1)). *spt* indicates the number of sectors to format. *track* indicates which track to format.

side indicates the side to format. *intlv* should be **FLOP_NOSKEW** (1) for consecutive sectors or **FLOP_SKEW** (-1) to interleave the sectors based on the array pointed to by *skew*.

magic is a fixed magic number which must be **FLOP_MAGIC** (0x87654321). *virgin* is the value to assign to uninitialized sector data (should be **FLOP_VIRGIN** (0xE5E5)).

BINDING

```
move.w      virgin,-(sp)
move.l      magic,-(sp)
move.w      intlv,-(sp)
move.w      side,-(sp)
move.w      track,-(sp)
move.w      spt,-(sp)
move.w      dev,-(sp)
pea         skew
pea         buf
move.w      #$0A,-(sp)
trap        #14
lea         26(sp),sp
```

RETURN VALUE **Flopfmt()** returns 0 if the track was formatted successfully or non-zero otherwise.

Also, upon exit, *buf* will be filled in with a **WORD** array of sectors that failed formatting terminated by an entry of 0. If no errors occurred then the first **WORD** of *buf* will be 0.

COMMENTS

The steps required to a format a floppy disk are as follows:

1. Call **Flopfmt()** to format the disk as desired.
2. Call **Protobt()** to create a prototype boot sector in memory.
3. Call **Flopwr()** to write the prototype boot sector to track 0, side 0, sector 1.

Interleaved sector formatting is only possible as of **TOS** 1.2. *skew* should be set to **NULL** and *intlv* should be set to **FLOP_NOSKEW** under **TOS** 1.0.

Specifying an *intlv* value of **FLOP_SKEW** and a *skew* array equalling { 1, 2, 3, 4, 5, 6, 7, 8, 9 } is the same as specifying an *intlv* value of **FLOP_NOSKEW**. To accomplish a 9 sector 2:1 interleave you would use a *skew* array which looked like: { 1, 6, 2, 7, 3, 8, 4, 9, 5 }.

The '_FDC' cookie (if present) contains specific information regarding the installed floppy drives. The lower three bytes of the cookie value contain a three-letter code indicating the manufacturer of the drive (Atari is 0x415443 'ATC'). The high byte determines the capabilities of the highest density floppy drive currently installed as follows:

Name	Value	Meaning
FLOPPY_DSDD	0	Standard Density (720K)
FLOPPY_DSHD	1	High Density (1.44MB)
FLOPPY_DSED	2	Extra High Density (2.88MB)

To format a high density diskette, multiple the *spt* parameter by 2. To format a extra-high density diskette, multiply the *spt* parameter by 4.

This call forces a 'media changed' state on the device which will be returned on the next **Mediach()** or **Rwabs()** call.

SEE ALSO **Floprate(), Floprd(), Flopwr()**

Floprate()

WORD Floprate(***dev, rate*** **)**
WORD ***dev, rate*****;**

Floprate() sets the seek rate of the specified floppy drive.

OPCODE 41 (0x29)

AVAILABILITY Available on all **TOS** versions except 1.00.

PARAMETERS *dev* indicates the floppy drive whose seek rate you wish to modify ('A:' = **FLOP_DRIVEA** (0), 'B:' = **FLOP_DRIVEB** (1)). *rate* specifies the seek rate as follows:

Name	*rate*	Meaning
FRATE_6	0	Set seek rate to 6ms
FRATE_12	1	Set seek rate to 12ms
FRATE_2	2	Set seek rate to 2ms
FRATE_3	3	Set seek rate to 3ms

A *rate* value of **FRATE_INQUIRE** (-1) will inquire the current seek rate without modifying it.

BINDING

```
move.w      rate,-(sp)
move.w      dev,-(sp)
move.w      #$29,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Floprate()** returns the prior seek rate for the specified drive.

COMMENTS **TOS** version 1.00 can have its seek rates set by setting the system variable (*_seekrate* (**WORD** *)0x440) to the desired value (as in *rate*). Note that you can only set the seek rate for *both* drives in this manner.

Floprd()

WORD Floprd(*buf, rsrvd, dev, sector, track, side, count*)
VOIDP *buf*;
LONG *rsrvd*;
WORD *dev, sector, track, side, count*;

Floprd() reads sectors from a floppy disk.

OPCODE 8 (0x08)

AVAILABILITY All **TOS** versions.

PARAMETERS *buf* points to a word-aligned buffer where the data to be read will be stored. *rsrvd* is currently unused and should be 0. *dev* specifies the floppy drive to read from

('A:' = **FLOP_DRIVEA** (0), 'B:' = **FLOP_DRIVEB** (1)). The function reads *count* physical sectors starting at sector *sector*, track *track*, side *side*.

BINDING

```
move.w      count,-(sp)
move.w      side,-(sp)
move.w      track,-(sp)
move.w      sector,-(sp)
move.w      dev,-(sp)
move.l      rsrvd,-(sp)
pea         buf
move.w      #$08,-(sp)
trap        #14
lea         20(sp),sp
```

RETURN VALUE **Floprd()** returns 0 if the operation was successful or non-zero otherwise.

CAVEATS This function reads sectors in physical order (not taking interleave into account). Use **Rwabs()** to read logical sectors.

SEE ALSO **Flopwr(), Flopfmt(), Flopver(), Rwabs()**

Flopver()

WORD Flopver(***buf, rsrvd, dev, sector, track, side, count*** **)**
VOIDP ***buf*****;**
LONG ***rsrvd*****;**
WORD ***dev, sector, track, side, count*****;**

Flopver() verifies data on a floppy disk with data in memory.

OPCODE 19 (0x13)

AVAILABILITY All **TOS** versions.

PARAMETERS *buf* is a pointer to a word-aligned buffer to compare the sector against. *rsrvd* is unused and should be 0. *dev* specifies the drive to verify ('A:' = **FLOP_DRIVEA** (0), 'B:' = **FLOP_DRIVEB** (1)). This function verifies *count* sectors starting at sector *sector*, track *track*, side *side*.

BINDING

```
move.w      count,-(sp)
move.w      side,-(sp)
move.w      track,-(sp)
move.w      sector,-(sp)
move.w      dev,-(sp)
move.l      rsrvd,-(sp)
pea         buf
move.w      #$13,-(sp)
trap        #14
lea         20(sp),sp
```

RETURN VALUE **Flopver()** returns 0 if all sectors were successfully verified or a non-zero value otherwise.

CAVEATS This function only verifies sectors in physical order.

COMMENTS As with **Flopfmt()**, upon the return of the function, *buf* is filled in with a **WORD** array containing a list of any sectors which failed. The array is terminated with a **NULL**.

SEE ALSO **Flopwr(), Flopfmt()**

Flopwr()

WORD Flopwr(***buf, rsrvd, dev, sector, track, side, count*** **)**
VOIDP ***buf*****;**
LONG ***rsrvd*****;**
WORD ***dev, sector, track, side, count*****;**

Flopwr() writes sectors to the floppy drive.

OPCODE 9 (0x09)

AVAILABILITY All **TOS** versions.

PARAMETERS *buf* is a pointer containing data to write. *rsrvd* is currently unused and should be set to 0. *dev* specifies the floppy drive to write to ('A:' = 0,'B:' = 1). This function writes *count* sectors starting at sector *sector*, track *track*, side *side*.

BINDING

```
move.w      count,-(sp)
move.w      side,-(sp)
move.w      track,-(sp)
move.w      sector,-(sp)
move.w      dev,-(sp)
move.l      rsrvd,-(sp)
pea         buf
move.w      #$09,-(sp)
trap        #14
lea         20(sp),sp
```

RETURN VALUE **Flopwr()** returns 0 if the sectors were successfully written or non-zero otherwise.

CAVEATS This function writes sectors in physical order only (ignoring interleave). Use **Rwabs()** to write sectors in logical order.

COMMENTS If this call is used to write to track 0, sector 1, side 0, the device will enter a

'media might have changed' state indicated upon the next **Rwabs()** or **Mediach()** call.

SEE ALSO **Floprd(), Flopfmt(), Flopver(),Rwabs()**

Getrez()

WORD Getrez(VOID)

Getrez() returns a machine-dependent code representing the current screen mode/ratio.

OPCODE 4 (0x04)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$04,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Getrez()** returns a value representing the current video display mode. To find the value you will receive back based on current Atari manufactured video hardware, refer to the following chart:

Colors: / Screen Dimension:	2	4	16	256	True Color
320x200	X	X	0	0	X
320x240	X	0	0	0	0
320x480	X	7	7	7	7
640x200	1	X	X	X	X
640x400	2	X	X	X	X
640x480	2	2	2†	2	2
1280x960	6	X	X	X	X

† This value varies. TT030 Medium resolution returns a value of 4, however, the Falcon returns a value of 2.

CAVEATS This call is *extremely* machine-dependent. Dependence on this call will make your program incompatible with third-party video boards and future hardware. Use the values returned by **v_opnvwk()** to determine screen attributes.

COMMENTS Use of this call in preparing to call **v_opnvwk()** is acceptable and must be done to specify the correct fonts to load from **GDOS**.

SEE ALSO **VsetMode(), Egetshift(), Setscreen()**

Gettime()

LONG Gettime(VOID)

Gettime() returns the current IKBD time.

OPCODE 23 (0x17)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$17,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Gettime()** returns a **LONG** bit array packed with the current IKBD time as follows:

Bits	Meaning
0-4	Seconds/2 (0-29)
5-10	Minute (0-59)
11-15	Hour (0-23)
16-20	Day (1-31)
21-24	Month (1-12)
25-31	Year-1980 (0-127)

The return value can be represented in a C structure as follows:

```
typedef struct
{
       unsigned year:7;
       unsigned month:4;
       unsigned day:5;
       unsigned hour:5;
       unsigned minute:6;
       unsigned second:5;
} BIOS_TIME;
```

SEE ALSO **Settime(), Tgettime(), Tgetdate()**

Giaccess()

WORD Giaccess(*data, register*)
WORD *data, register*;

Giaccess() reads/sets the registers of the FM sound chip and Port A/B peripherals.

OPCODE 28 (0x1C)

AVAILABILITY All **TOS** versions.

PARAMETERS The lower eight bits of *data* are written to the register selected by *register* if the value for *register* is OR'ed with 0x80 (high bit set). If this bit is not set, *data* is ignored and the value of the *register* is returned. *register* selects the register to read/write to as follows:

Name	*register*	Meaning
PSG_APITCHLOW **PSG_BPITCHHIGH**	0 1	Set the pitch of the PSG's channel A to the value in registers 0 and 1. Register 0 contains the lower 8 bits of the frequency and the lower 4 bits of register 1 contain the upper 4 bits of the frequency's 12-bit value.
PSG_BPITCHLOW **PSG_BPITCHHIGH**	2 3	Set the pitch of the PSG's channel B to the value in registers 0 and 1. Register 0 contains the lower 8 bits of the frequency and the lower 4 bits of register 1 contain the upper 4 bits of the frequency's 12-bit value.
PSG_CPITCHLOW **PSG_CPITCHHIGH**	2 3	Set the pitch of the PSG's channel C to the value in registers 0 and 1. Register 0 contains the lower 8 bits of the frequency and the lower 4 bits of register 1 contain the upper 4 bits of the frequency's 12-bit value.
PSG_NOISEPITCH	6	The lower five bits of this register set the pitch of white noise. The lower the value, the higher the pitch.
PSG_MODE	7	This register contains an eight bit map which determines various aspects of sound generation. Setting each bit on causes the following actions: **Name** — **Bit Mask** — **Meaning** **PSG_ENABLEA** 0x01 Chnl A tone enable **PSG_ENABLEB** 0x02 Chnl B tone enable **PSG_ENABLEC** 0x04 Chnl C tone enable **PSG_NOISEA** 0x08 Chnl A white noise on **PSG_NOISEB** 0x10 Chnl B white noise on **PSG_NOISEC** 0x20 Chnl C white noise on **PSG_PRTAOUT** 0x40 Port A: 0 = input, 1 = output **PSG_PRTBOUT** 0x80 Port B: 0 - input, 1 = output

PSG_AVOLUME	8	This register controls the volume of channel A. Values from 0-15 are absolute volumes with 0 being the softest and 15 being the loudest. Setting bit 4 causes the PSG to ignore the volume setting and to use the envelope setting in register 13.
PSG_BVOLUME	9	This register controls the volume of channel B. Values from 0-15 are absolute volumes with 0 being the softest and 15 being the loudest. Setting bit 4 causes the PSG to ignore the volume setting and to use the envelope setting in register 13.
PSG_CVOLUME	10	This register controls the volume of channel C. Values from 0-15 are absolute volumes with 0 being the softest and 15 being the loudest. Setting bit 4 causes the PSG to ignore the volume setting and to use the envelope setting in register 13.
PSG_FREQLOW **PSG_FREQHIGH**	11 12	Register 11 contains the low byte and register 12 contains the high byte of the frequency of the waveform specified in register 13. This value may range from 0 to 65535.
PSG_ENVELOPE	13	The lower four bits of the register contain a value which defines the envelope wavefrom of the PSG. The best definition of values is obtained through experimentation.
PSG_PORTA	14	This register accesses Port A of the Yamaha PSG. It is recommended that the functions **Ongibit()** and **Offgibit()** be used to access this register.
PSG_PORTB	15	This register accesses Port B of the Yamaha PSG. This register is currently assigned to the data in/out line of the Centronics Parallel port.

BINDING

```
move.w      register,-(sp)
move.w      data,-(sp)
move.w      #$1C,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Giaccess()** returns the value of the register in the lower eight bits of the word if *data* was OR'ed with 0x80.

Gpio()

LONG Gpio(*mode, data*)
WORD *mode, data*;

Gpio() reads/writes data over the general purpose pins on the DSP connector.

OPCODE 138 (0x8A)

AVAILABILITY Available if '_SND' cookie has bit 3 set.

PARAMETERS *mode* specifies the meaning of *data* and the return value as follows:

Name	*mode*	Meaning
GPIO_INQUIRE	0	Return the old value.
GPIO_READ	1	Read the three general purpose pins and return their state in the lower three bits of the returned value. *data* is ignored.
GPIO_WRITE	2	Write the lower three bits of *data* to the corresponding DSP pins. The return value is 0.

BINDING

```
move.w      data,-(sp)
move.w      mode,-(sp)
move.w      #$8A,-(sp)
trap        #14
addq.l      #6,sp
```

Ikbdws()

VOID Ikbdws(*len*, *buf*)
WORD *len*;
CHAR **buf*;

Ikbdws() writes the contents of a buffer to the intelligent keyboard controller.

OPCODE 25 (0x19)

AVAILABILITY All **TOS** versions.

PARAMETERS This function writes *len* + 1 characters from buffer *buf* to the IKBD.

BINDING

```
pea         buf
move.w      len,-(sp)
move.w      #$19,-(sp)
trap        #14
addq.l      #8,sp
```

Initmous()

VOID Initmous(***mode, param, vec*** **)**
WORD ***mode*****;**
VOIDP ***param*****;**
VOID (***vec*****)();**

Initmous() determines the method of handling IKBD mouse packets from the system.

OPCODE 0 (0x00)

AVAILABILITY All **TOS** versions.

PARAMETERS *mode* indicates a IKBD reporting mode and defines the meaning of the other parameters as listed below. *hand* points to a mouse packet handler which is called when each mouse packet is sent. Register A0 contains the mouse packet address when called.

Name	*mode*	Meaning
IM_DISABLE	0	Disable mouse reporting.

IM_RELATIVE	1	Enable relative mouse reporting mode. Packets report offsets from the previous mouse position. In this mode, *param* is a pointer to a structure as follows:
		
		`struct param`
		`{`
		`  BYTE topmode;`
		`  BYTE buttons;`
		`  BYTE xparam;`
		`  BYTE yparam;`
		`}`
		
		topmode is **IM_YBOT** (0) to indicate that Y=0 means bottom of the screen. A *topmode* value of **IM_YTOP** (1) indicates that Y=0 means the top of the screen.
		
		buttons is a bit array which affect the way mouse clicks are handled. A value of **IM_KEYS** (4) causes mouse buttons to generate keycodes rather than mouse packets. A value of **IM_PACKETS** (3) causes the absolute mouse position to be reported on each button press.
		
		xparam and *yparam* specify the number of mouse X/Y increments between position report packets.
		
		This mode is the default mode of the **AES** and **VDI**.

IM_ABSOLUTE	2	Enable absolute mouse reporting mode. Packets report actual screen positions. In this mode, *param* is a pointer to a structure as follows:
		
		`struct param`
		`{`
		`  BYTE topmode;`
		`  BYTE buttons;`
		`  BYTE xparam;`
		`  BYTE yparam;`
		`  WORD xmax;`
		`  WORD ymax;`
		`  WORD xinitial;`
		`  WORD yinitial;`
		`}`
		
		topmode, *buttons*, *xparam*, and *yparam* are the same as for mode 2.
		
		xmax and *ymax* specify the maximum X and Y positions the mouse should be allowed to move to. *xinital* and *yinitial* specify the mouse's initial location.

—	3	Unused
IM_KEYCODE	4	Enable mouse keycode mode. Keyboard codes for mouse movements are sent rather than actual mouse packets. *param* is handled the same as in mode 1.

BINDING

```
pea         hand
pea         param
move.w      mode,-(sp)
clr.w       -(sp)
trap        #14
lea         12(sp),sp
```

CAVEATS Changing the mouse packet handler to anything but relative mode will cause the **AES** and **VDI** to stop receiving mouse input.

SEE ALSO **Kbdvbase()**

Iorec()

IOREC *Iorec(*dev*)
WORD *dev*;

Iorec() returns the address in memory of system data structures relating to the buffering of input data.

OPCODE 14 (0x0E)

AVAILABILITY All **TOS** versions.

PARAMETERS *dev* specifies the device to return information about as follows:

Name	*dev*	Meaning
IO_SERIAL	0	Currently mapped serial device (see **Bconmap()**)
IO_KEYBOARD	1	Keyboard
IO_MIDI	2	MIDI

BINDING

```
move.w      dev,-(sp)
move.w      #$0E,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Iorec()** returns the address of an **IOREC** array with either one element (Keyboard or MIDI) or two elements (RS-232 - 1st = input, 2nd = output). The **IOREC** structure is defined as follows:

```
typedef struct
{
        /* start of buffer */
        char *ibuf;

        /* size of buffer */
        WORD ibufsize;

        /* head index mark of buffer */
        WORD ibufhd;

        /* tail index mark of buffer */
        WORD ibuftl;

        /* low-water mark of buffer */
        WORD ibuflow;

        /* high-water mark of buffer */
        WORD ibufhi;
} IOREC;
```

SEE ALSO **Bconmap()**

Jdisint()

VOID Jdisint(*intno*)
WORD *intno*;

Jdisint() disables an MFP interrupt.

OPCODE 26 (0x1A)

AVAILABILITY All **TOS** versions.

PARAMETERS *intno* specifies the interrupt to disable (see **Mfpint()** for a list).

BINDING

```
move.w      intno,-(sp)
move.w      #$1A,-(sp)
trap        #14
addq.l      #4,sp
```

SEE ALSO **Jenabint(), Mfpint()**

Jenabint()

VOID Jenabint(*intno*)
WORD *intno*;

Jenabint() enables an MFP interrupt.

OPCODE 27 (0x1B)

AVAILABILITY All **TOS** versions.

PARAMETERS *intno* specifies the interrupt to enable (see **Mfpint()** for a list).

BINDING

```
move.w      intno,-(sp)
move.w      #$1B,-(sp)
trap        #14
addq.l      #4,sp
```

SEE ALSO **Jdsint(), Mfpint()**

Kbdvbase()

KBDVECS *Kbdvbase(VOID)

Kbdvbase() returns a pointer to a system structure containing a 'jump' table to system vector handlers.

OPCODE 34 (0x22)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$22,-(sp)
trap        #14
```

```
addq.l      #2,sp
```

RETURN VALUE **Kbdvbase()** returns a pointer to a system structure **KBDVECS** which is defined as follows:

```
typedef struct
{
       VOID (*midivec)( UBYTE data );  /* MIDI Input */
       VOID (*vkbderr)( UBYTE data );  /* IKBD Error */
       VOID (*vmiderr)( UBYTE data );  /* MIDI Error */
       VOID (*statvec)(char *buf);     /* IKBD Status */
       VOID (*mousevec)(char *buf);        /* IKBD Mouse */
       VOID (*clockvec)(char *buf);        /* IKBD Clock */
       VOID (*joyvec)(char *buf);      /* IKBD Joystick */
       VOID (*midisys)( VOID );        /* Main MIDI Vector */
       VOID (*ikbdsys)( VOID );        /* Main IKBD Vector */
       char ikbdstate;                 /* See below */
} KBDVECS;
```

midivec is called with the received data byte in d0. If an overflow error occurred on either ACIA, *vkbderr* or *vmiderr* will be called, as appropriate by *midisys* or ikbdsys with the contents of the ACIA data register in d0.

statvec, *mousevec*, *clockvec*, and *joyvec* all are called with the address of the packet in register A0.

midisys and *ikbdsys* are called by the MFP ACIA interrupt handler when a character is ready to be read from either the midi or keyboard ports.

ikbdstate is set to the number of bytes remaining to be read by the *ikbdsys* handler from a multiple-byte status packet.

COMMENTS If you intercept any of these routines you should either JMP through the old handler or RTS.

SEE ALSO **Initmous()**

Kbrate()

WORD Kbrate(*delay, rate*)
WORD *delay, rate*;

Kbrate() reads/modifies the keyboard repeat/delay rate.

OPCODE 35 (0x23)

AVAILABILITY All **TOS** versions.

PARAMETERS *delay* specifies the amount of time (in 50Hz ticks) before a key begins repeating. *rate* indicates the amount of time between repeats (in 50Hz ticks). A parameter of **KB_INQUIRE** (-1) for either of these values leaves the value unchanged.

BINDING

```
move.w      rate,-(sp)
move.w      delay,-(sp)
move.w      #$23,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Kbrate()** returns a **WORD** with the low byte being the old value for *rate* and the high byte being the old value for *delay*.

Keytbl()

KEYTAB *Keytbl(*normal, shift, caps*)
char **unshift, *shift, *caps*;

Keytbl() reads/modifies the internal keyboard mapping tables.

OPCODE 16 (0x10)

AVAILABILITY All **TOS** versions.

PARAMETERS *normal* is a pointer to an array of 128 **CHAR**s which can be indexed by a keyboard scancode to return the correct ASCII value for a given unshifted key. *shift* and *caps* point to similar array except their values are only utilized when SHIFT and CAPS-LOCK respectively are used. Passing a value of **KT_NOCHANGE** ((char *)-1) will leave the table unchanged.

BINDING

```
pea         caps
pea         shift
pea         normal
move.w      #$10,-(sp)
trap        #14
lea         14(sp),sp
```

RETURN VALUE **Keytbl()** returns a pointer to a **KEYTAB** structure defined as follows:

```
typedef struct
{
        char *unshift;
        char *shift;
        char *caps;
} KEYTAB;
```

The entries in this table each point to the current keyboard lookup table in their category.

Entries are indexed with a keyboard scancode to obtain the ASCII value of a key. A value of 0 indicates that no ASCII equivalent exists.

SEE ALSO **Bioskeys()**

Locksnd()

LONG Locksnd(VOID)

Locksnd() prevents other applications from simultaneously attempting to use the sound system.

OPCODE 128 (0x80)

AVAILABILITY Available if the '_SND' cookie has bit 2 set.

BINDING

```
move.w      #$80,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Locksnd()** returns 1 if the sound system was successfully locked or **SNDLOCKED** (-129) if the sound system was already locked.

COMMENTS This call should be used prior to any usage of the 16-bit DMA sound system.

SEE ALSO **Unlocksnd()**

Logbase()

VOIDP Logbase(VOID)

Logbase() returns a pointer to the base of the logical screen.

OPCODE 3 (0x03)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$03,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Logbase()** returns a pointer to the base of the logical screen.

COMMENTS The logical screen should not be confused with the physical screen. The logical screen is the memory area where the **VDI** does any drawing. The physical screen is the memory area where the video shifter gets its data from. Normally they are the same; however, keeping the addresses separate facilitates screen flipping.

SEE ALSO **Physbase()**

Metainit()

VOID Metainit(*metainfo*)
METAINFO ****metainfo***;**

Metainit() returns information regarding the current version and installed drives of **MetaDOS**.

OPCODE 48 (0x30)

AVAILABILITY To test for the availability of **MetaDOS** the following steps must be taken:

1. Fill the **METAINFO** structure with all zeros.
2. Call **Metainit()**.
3. If *metainfo.version* is **NULL**, **MetaDOS** is not installed.

PARAMETERS *metainfo* is a pointer to a **METAINFO** structure which is filled in by the call. **METAINFO** is defined as:

```
typedef struct
{
       /* Bitmap of drives (Bit 0 = A, 1 = B, etc... */
       ULONG drivemap;

       /* String containing name and version */
       char *version;

       /* Currently unused */
       LONG reserved[2];
} METAINFO;
```

BINDING

```
pea          metainfo
move.w       #$30,-(sp)
trap         #14
addq.l       #6,sp
```

Mfpint()

VOID Mfpint(***intno, vector*** **)**
WORD ***intno*****;**
VOID (***vector*****)();**

Mfpint() defines an interrupt handler for an MFP interrupt.

OPCODE 13 (0x0D)

AVAILABILITY All **TOS** versions.

PARAMETERS *intno* is an index to a vector to replace with *vector* as follows:

Name	*intno*	Vector
MFP_PARALLEL	0	Parallel port
MFP_DCD	1	RS-232 Data Carrier Detect
MFP_CTS	2	RS-232 Clear To Send
MFP_BITBLT	3	BitBlt Complete
MFP_TIMERD or **MFP_BAUDRATE**	4	Timer D (RS-232 baud rate generator)
MFP_200HZ	5	Timer C (200Hz system clock)
MFP_ACIA	6	Keyboard/MIDI vector
MFP_DISK	7	Floppy/Hard disk vector
MFP_TIMERB or **MFP_HBLANK**	8	Timer B (Horizontal blank)
MFP_TERR	9	RS-232 transmit error
MFP_TBE	10	RS-232 transmit buffer empty
MFP_RERR	11	RS-232 receive error
MFP_RBF	12	RS-232 receive buffer full.
MFP_TIMERA or **MFP_DMASOUND**	13	Timer A (DMA sound)
MFP_RING	14	RS-232 ring indicator
MFP_MONODETECT	15	Mono monitor detect/DMA sound complete

BINDING

```
pea         vector
move.w      intno,-(sp)
move.w      #$0D,-(sp)
trap        #14
addq.l      #8,sp
```

CAVEATS This call does not return the address of the old handler.

The only RS-232 vector that may be set on the Falcon030 with this function is the ring indicator.

COMMENTS Newly installed interrupts must be enabled with **Jenabint()**.

SEE ALSO **Jenabint(), Jdisint()**

Midiws()

VOID Midiws(***count, buf*** **)**
WORD ***count*****;**
char * ***buf*****;**

Midiws() outputs a data buffer to the MIDI port.

OPCODE 12 (0x0C)

AVAILABILITY All **TOS** versions.

PARAMETERS *count* + 1 characters are written from the buffer pointed to by *buf*.

BINDING

```
pea         buf
move.w      count,-(sp)
move.w      #$0C,-(sp)
trap        #14
addq.l      #8,sp
```

NVMaccess()

WORD NVMaccess(***op, start, count, buffer*** **)**
WORD ***op, start, count*****;**
char * ***buffer*****;**

NVMaccess() reads/modifies data in non-volatile (battery backed-up) memory.

OPCODE 46 (0x2E)

AVAILABILITY This function's availability is variable. If it returns 0x2E (its opcode) when called, the function is non-existent and the operation was not carried out.

PARAMETERS *op* indicates the operation to perform as follows:

Name	*op*	Meaning
NVM_READ	0	Read *count* bytes of data starting at offset *start* and place the data in *buffer*.
NVM_WRITE	1	Write *count* bytes of data from *buffer* starting at offset *start*.
NVM_RESET	2	Resets and clears all data in non-volatile memory.

BINDING

```
pea         buffer
move.w      count,-(sp)
move.w      start,-(sp)
move.w      op,-(sp)
move.w      #$2E,-(sp)
trap        #14
lea         12(sp),sp
```

RETURN VALUE **NVMaccess()** returns 0 if the operation succeeded or a negative error code otherwise.

CAVEATS All of the locations are reserved for use by Atari and none are currently documented.

COMMENTS Currently there is a total of 50 bytes in non-volatile RAM.

Offgibit()

VOID Offgibit(***mask*** **)**
WORD *mask*;

Offgibit() clears individual bits of the sound chip's Port A.

OPCODE 29 (0x1D)

AVAILABILITY All **TOS** versions.

PARAMETERS *mask* is a bit mask arranged as shown below. For each of the lower eight bits in *mask* set to 0, that bit will be reset. Other bits (set as 1) will remain unchanged.

Name	Mask	Meaning
GI_FLOPPYSIDE	0x01	Floppy side select
GI_FLOPPYA	0x02	Floppy A select
GI_FLOPPYB	0x04	Floppy B select
GI_RTS	0x08	RS-232 Request To Send
GI_DTR	0x10	RS-232 Data Terminal Ready
GI_STROBE	0x20	Centronics strobe
GI_GPO	0x40	General purpose output (On a Falcon030, this bit controls the state of the internal speaker)
GI_SCCPORT	0x80	On a Mega STe or TT030, calling **Ongibit(** 0x80 **)** will cause SCC channel A to control the Serial 2 port rather than the LAN. To select the LAN, use **Offgibit(** 0x7F **)**.

BINDING

```
move.w      mask,-(sp)
```

```
move.w      #$1D,-(sp)
trap        #14
addq.l      #4,sp
```

SEE ALSO **Giaccess(), Ongibit()**

Ongibit()

VOID Ongibit(*mask*)
WORD *mask*;

Ongibit() sets individual bits of the sound chip's assigned Port A.

OPCODE 30 (0x1E)

AVAILABILITY All **TOS** versions.

PARAMETERS *mask* is a bit mask arranged as defined in **Offgibit()**. For each of the lower eight bits in *mask* set to 1, that bit will be set. Other bits (set as 0) will remain unchanged.

BINDING

```
move.w      mask,-(sp)
move.w      #$1E,-(sp)
trap        #14
addq.l      #4,sp
```

SEE ALSO **Giaccess(), Offgibit()**

Physbase()

VOIDP Physbase(VOID)

Physbase() returns the address of the physical base of screen memory.

OPCODE 2 (0x02)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$02,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Physbase()** returns the physical base address of the screen.

COMMENTS The physical base address is the memory area where the video shifter reads its

data. The logical address is the memory area where the **VDI** draws. These are normally the same but are addressed individually to enable screen flipping.

SEE ALSO **Logbase()**

Protobt()

VOID Protobt(***buf, serial, type, execflag*** **)**
VOIDP ***buf*****;**
LONG ***serial*****;**
WORD ***type, execflag*****;**

Protobt() creates a prototype floppy boot sector in memory for writing to a floppy drive.

OPCODE 18 (0x12)

AVAILABILITY All **TOS** versions.

PARAMETERS *buf* is a 512 byte long buffer where the prototyped buffer will be written. If you are creating an executable boot sector, the memory buffer should contain the code you require. *serial* can be any of the following values:

Name	*serial*	Meaning
SERIAL_NOCHANGE	-1	Don't change the serial number already in memory.
SERIAL_RANDOM	>0x01000000	Use a random number for the serial number
—	any other positive number	Set the serial number to *serial.*

type defines the type of disk to prototype as follows:

Name	*type*	Meaning
DISK_NOCHANGE	-1	Don't change disk type.
DISK_SSSD	0	40 Track, Single-Sided (180K)
DISK_DSSD	1	40 Track, Double-Sided (360K)
DISK_SSDD	2	80 Track, Single-Sided (360K)
DISK_DSDD	3	80 Track, Double-Sided (720K)
DISK_DSHD	4	High Density (1.44MB)
DISK_DSED	5	Extra-High Density (2.88MB)

execflag specifies the executable status of the boot sector as follows:

Name	*execflag*	Meaning
EXEC_NOCHANGE	-1	Don't alter executable status
EXEC_NO	0	Disk is not executable
EXEC_YES	1	Disk is executable

BINDING

```
move.w      execflag,-(sp)
move.w      type,-(sp)
move.l      serial,-(sp)
pea         buf
move.w      #$12,-(sp)
trap        #14
lea         14(sp),sp
```

CAVEATS *type* values of **DISK_DSHD** and **DISK_DSED** are only available when the high byte of the '_FDC' cookie has a value of **FLOPPY_DSHD** (1) and **FLOPPY_DSED** (2) respectively.

COMMENTS To create an MS-DOS compatible disk you must set the first three bytes of the prototyped boot sector to 0xE9, 0x00, and 0x4E.

SEE ALSO **Flopfmt(), Flopwr()**

Prtblk()

WORD Prtblk(*blk*)
PRTBLK ****blk***;**

Prtblk() accesses the built-in bitmap/text printing code.

OPCODE 36 (0x24)

AVAILABILITY All **TOS** versions.

PARAMETERS *blk* is a **PRTBLK** pointer containing information about the bitmap or text to print. **PRTBLK** is defined as follows:

```
typedef struct
{
       VOIDP blkptr;      /* pointer to screen scanline */
       UWORD offset;      /* bit offset of first column */
       UWORD width;       /* width of bitmap in bits */
       UWORD height;      /* height of bitmap in scanlines */
       UWORD left;        /* left print margin (in pixels) */
       UWORD right;       /* right print margin (in pixels) */
       UWORD srcres;      /* same as Getrez() */
       UWORD destres;     /* 0 = draft, 1 = final */
       UWORD *colpal;     /* color palette pointer */
       /*
```

```
         * 0 = B/W Atari
         * 1 = Color Atari
         * 2 = Daisy Wheel
         * 3 = B/W Epson
         */
        UWORD type;
        /* 0 = parallel, 1 = serial */
        UWORD port;
        /* halftone mask pointer or NULL to use default */
        char *masks;
} PRTBLK;
```

BINDING

```
pea         prtblk
move.w      #$24,-(sp)
trap        #14
addq.l      #6,sp
```

CAVEATS This call is extremely device dependent. **v_bit_image()** with **GDOS** installed should be used instead. Only ST compatible screen resolution bitmaps may be printed with this utility function.

COMMENTS When printing text, *blkptr* should point to the text string, *width* should be the length of the text string, *height* should be 0, and *masks* should be **NULL**.

In graphic print mode, *masks* can be **NULL** to use the default halftone masks.

The system variable *_prt_cnt* (**WORD** *)0x4EE should be set to 1 to disable the ALT-HELP key before calling this function. It should be restored to a value of -1 when done.

SEE ALSO **Scrdump(), SetPrt()**

Puntaes()

VOID Puntaes(VOID)

Puntaes() discards the **AES** (if memory-resident) and restarts the system.

OPCODE 39 (0x27)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$27,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE If successful, this function will not return control to the caller.

CAVEATS **Puntaes()** is only valid with disk-loaded AES's.

COMMENTS **Puntaes()** discards the **AES** by freeing any memory it allocated, resetting the system variable *os_magic* (this variable should contain the magic number 0x87654321, however if reset, the **AES** will not initialize), and rebooting the system.

Random()

LONG Random(VOID)

Random() returns a 24 bit random number.

OPCODE 17 (0x11)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$11,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Random()** returns a 24-bit random value in the lower three bytes of the returned **LONG**.

CAVEATS The algorithm used provides an exact 50% occurrence of bit 0.

Rsconf()

ULONG Rsconf(*speed, flow, ucr, rsr, tsr, scr*)
WORD *speed, flow, ucr, rsr, tsr, scr*;

Rsconf() reads/modifies the configuration of the serial device currently mapped to **BIOS** device #1 (**GEMDOS** 'aux:').

OPCODE 15 (0x0F)

AVAILABILITY All **TOS** versions.

PARAMETERS *speed* sets the serial device speed as follows:

Name	*speed*	Baud Rate
BAUD_19200	0	19200
BAUD_9600	1	9600

Name	*speed*	Baud Rate
BAUD_600	8	600
BAUD_300	9	300

BAUD_4800	2	4800
BAUD_3600	3	3600
BAUD_2400	4	2400
BAUD_2000	5	2000
BAUD_1800	6	1800
BAUD_1200	7	1200

BAUD_200	10	200
BAUD_150	11	150
BAUD_134	12	134
BAUD_110	13	110
BAUD_75	14	75
BAUD_50	15	50

If *speed* is set to **BAUD_INQUIRE** (-2), the last baud rate set will be returned.

flow selects the flow control method as follows:

Name	*flow*	Meaning
FLOW_NONE	0	No flow control
FLOW_SOFT	1	XON/XOFF flow control (CTRL-S/CTRL-Q)
FLOW_HARD	2	RTS/CTS flow control (hardware)
FLOW_BOTH	3	Both methods of flow control

ucr, *rsr*, and *tsr* are each status bit arrays governing the serial devices. Each parameter uses only the lower eight bits of the **WORD**. They are defined as follows:

Mask	*ucr*	*rsr* and *tsr*
0x01	Unused	Receiver enable: **RS_RECVENABLE**
0x02	Enable odd parity **RS_ODDPARITY** (0x02) **RS_EVENPARITY** (0x00)	Sync strip **RS_SYNCSTRIP**
0x04	Parity enable **RS_PARITYENABLE**	Match busy **RS_MATCHBUSY**
0x08	Bits 3-4 of the ucr collectively define the start and stop bit configuration as follows: 00 = No Start or Stop bits **RS_NOSTOP** (0x00) 01 = 1 Start bit, 1 Stop bit **RS_1STOP** (0x08) 10 = 1 Start bit, 1½ Stop bits **RS_15STOP** (0x10) 11 = 1 Start bit, 2 Stop bits **RS_2STOP** (0x18)	Break detect **RS_BRKDETECT**
0x10	See above.	Frame error **RS_FRAMEERR**
0x20	Bits 5 and 6 together define the number of bits per word as follows: 00 = 8 bits **RS_8BITS** (0x00) 01 = 7 bits **RS_7BITS** (0x20)	Parity error **RS_PARITYERR**

	10 = 6 bits **RS_6BITS** (0x40) 11 = 5 bits **RS_5BITS** (0x60)	
0x40	See above.	Overrun error **RS_OVERRUNERR**
0x80	CLK/16 **RS_CLK16**	Buffer full **RS_BUFFULL**

scr sets the synchronous character register in which the low byte is used as the character to search for in an underrun error condition.

If a **RS_INQUIRE** (-1) is used for either *ucr*, *rsr*, *tsr*, or *scr*, then that parameter is read and the register is unmodified.

BINDING

```
move.w      scr,-(sp)
move.w      tsr,-(sp)
move.w      rsr,-(sp)
move.w      ucr,-(sp)
move.w      flow,-(sp)
move.w      speed,-(sp)
move.w      #$0F,-(sp)
trap        #14
lea         14(sp),sp
```

RETURN VALUE

Rsconf() returns the last set baud rate if *speed* is set to **RS_LASTBAUD** (-2). Otherwise, it returns the old settings in a packed **LONG** with *ucr* being in the high byte, down to *scr* being in the low byte.

COMMENTS

Bits in the *ucr*, *rsr*, *tsr*, and *scr* should be set atomically. To correctly change a value, read the old value, mask it as appropriate and then write it back.

Baud rates higher than 19,200 bps available with SCC-based serial devices may be set by using the appropriate **Fcntl()** call under **MiNT** or by directly programming the SCC chip.

CAVEATS

The baud rate inquiry mode (speed = **RS_LASTBAUD**) does not work at all on **TOS** versions less than 1.04. **TOS** version 1.04 requires the patch program TOS14FX2.PRG (available from Atari Corp.) to allow this mode to function. All other **TOS** versions support the function normally.

SEE ALSO

Bconmap()

Scrdmp()

VOID Scrdmp(VOID)

Scrdmp() starts the built-in hardware screen dump routine.

OPCODE 20 (0x14)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$14,-(sp)
trap        #14
addq.l      #2,sp
```

CAVEATS **Scrdmp()** only dumps ST compatible screen resolutions.

COMMENTS This routine is extremely device-dependent. You should use the **VDI** instead.

SEE ALSO **Prtblk(), v_hardcopy()**

Setbuffer()

LONG Setbuffer(***mode*****,** ***begaddr*****,** ***endaddr*** **)**
WORD ***mode*****;**
VOIDP ***begaddr*****;**
VOIDP ***endaddr*****;**

Setbuffer() sets the starting and ending addresses of the internal play and record buffers.

OPCODE 131 (0x83)

AVAILABILITY Available when bit #2 of the '_SND' cookie is set.

PARAMETERS *mode* specifies which registers are to be set. A *mode* value of **PLAY** (0) sets the play registers, a value of **RECORD** (1) sets the record registers. *begaddr* specifies the starting location of the buffer. *endaddr* specifies the first invalid location for sound data past *begaddr*.

BINDING

```
pea         endaddr
pea         begaddr
move.w      mode,-(sp)
move.w      #$83,-(sp)
trap        #14
lea         12(sp),sp
```

RETURN VALUE **Setbuffer()** returns a 0 if successful or non-zero otherwise.

SEE ALSO **Buffoper()**

Setcolor()

WORD Setcolor(*idx, new*)
WORD *idx, new*;

Setcolor() sets a ST/TT030 color register.

OPCODE 7 (0x07)

AVAILABILITY All **TOS** versions.

PARAMETERS *idx* specifies the color register to modify (0-16 on an ST, 0-255 on a STe or TT030). *new* is a bit array specifying the new color as follows:

Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
Unused	Red	Green	Blue

Each color value has its bits packed in an unusual manner to stay compatible between machines. Bits are ordered 0, 3, 2, 1 with 0 being the least signifigant bit. If *new* is **COL_INQUIRE** (-1) then the old color is returned.

BINDING

```
move.w      new,-(sp)
move.w      idx,-(sp)
move.w      #$06,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Setcolor()** returns the old value of the color register.

CAVEATS This call is extremely device-dependent. **vs_color()** should be used instead.

COMMENTS The top bit of each color nibble is unused on the original ST machines.

SEE ALSO **VsetRGB(), EsetColor(), Setpalette()**

Setinterrupt()

LONG Setinterrupt(*mode, cause*)
WORD *mode, cause*;

Setinterrupt() defines the conditions under which an interrupt is generated by the sound system

OPCODE 135 (0x87)

AVAILABILITY Available when bit #2 of the '_SND' cookie is set.

PARAMETERS *mode* configures interrupts to occur when the end of a buffer is reached. A value of **INT_TIMERA** (0) for *mode* sets Timer A, a value of **INT_I7** (1) sets the MFP i7 interrupt. *cause* defines the conditions for the interrupt as follows:

Name	*cause*	Meaning
INT_DISABLE	0	Disable interrupt
INT_PLAY	1	Interrupt at end of play buffer
INT_RECORD	2	Interrupt at end of record buffer
INT_BOTH	3	Interrupt at end of both buffers

BINDING

```
move.w      cause,-(sp)
move.w      mode,-(sp)
move.w      #$87,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Setinterrupt()** returns 0 if no error occurred or non-zero otherwise.

COMMENTS If either buffer is in repeat mode, these interrupts can be used to double-buffer sounds.

SEE ALSO **Buffoper()**

Setmode()

LONG Setmode(*mode*)
WORD *mode*;

Setmode() sets the mode of operation for the play and record registers.

OPCODE 132 (0x84)

AVAILABILITY Available if bit #2 of the '_SND' cookie is set.

PARAMETERS *mode* defines the playback and record mode as follows:

Name	*mode*	Meaning
MODE_STEREO8	0	8-bit Stereo Mode
MODE_STEREO16	1	16-bit Stereo Mode
MODE_MONO	2	8-bit Mono Mode

BINDING

```
move.w      mode,-(sp)
move.w      #$84,sp
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Setmode()** returns 0 if the operation was successful or non-zero otherwise.

CAVEATS Recording only works in 16-bit stereo mode.

SEE ALSO **Buffoper()**

Setmontracks()

LONG Setmontracks(*track*)
WORD *track*;

Setmontracks() defines which playback track is audible through the internal speaker.

OPCODE 134 (0x86)

AVAILABILITY Available only when bit #2 of the '_SND' cookie is set.

PARAMETERS *track* specifies the playback track to monitor (0-3).

BINDING

```
move.w      track,-(sp)
move.w      #$86,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Setmontracks()** returns a 0 if the operation was successful or non-zero otherwise.

Setpalette()

VOID Setpalette(***palette*** **)**
WORD *****palette**;***

Setpalette() loads the ST color lookup table with a new palette.

OPCODE 6 (0x06)

AVAILABILITY All **TOS** versions.

PARAMETERS *palette* is a pointer to a **WORD** array containing 16 color encoded **WORD**s as defined in **Setcolor()**.

BINDING

```
pea         palette
move.w      #$06,-(sp)
trap        #14
addq.l      #6,sp
```

COMMENTS The actual palette data is not copied from the specified array until the next vertical blank interrupt. For this reason, this call should be followed by **Vsync()** to be sure the array memory is not modified or reallocated prior to the transfer.

SEE ALSO **Setcolor(), EsetPalette(), VsetRGB(), vs_color()**

Setprt()

WORD Setprt(***new*** **)**
WORD ***new*****;**

Setprt() sets the OS's current printer configuration bits.

OPCODE 33 (0x21)

AVAILABILITY All **TOS** versions.

PARAMETERS *new* is a **WORD** bit array defined as follows:

Mask	When clear	When Set
0x01	Dot Matrix **PRT_DOTMATRIX**	Daisy Wheel **PRT_DAISY**
0x02	Monochrome **PRT_MONO**	Color **PRT_COLOR**
0x04	Atari Printer	Epson Printer

	PRT_ATARI	PRT_EPSON
0x08	Draft Mode **PRT_DRAFT**	Final Mode **PRT_FINAL**
0x10	Parallel Port **PRT_PARALLEL**	Serial Port **PRT_SERIAL**
0x20	Continuous Feed **PRT_CONTINUOUS**	Single Sheet Feed **PRT_SINGLE**
–	Unused	Unused

If *new* is set to **PRT_INQUIRE** (-1) **Setprt()** will return the current configuration without modifying the current setup.

BINDING

```
move.w      new,-(sp)
move.w      #$33,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Setprt()** returns the prior configuration.

CAVEATS This call only affects the internal screen dump code which only operates on ST compatible resolutions.

SEE ALSO **Prtblk(), Scrdmp(), v_hardcopy()**

Setscreen()

VOID Setscreen(*log, phys, mode*)
VOIDP *log, phys*;
WORD *mode*;

Setscreen() changes the base addresses and mode of the current screen.

OPCODE 5 (0x05)

AVAILABILITY All **TOS** versions.

PARAMETERS *log* is the address for the new logical screen base. *phys* is the new address for the physical screen base. *mode* defines the screen mode to switch to (same as **Getrez()**). If any of these three parameters is set to **SCR_NOCHANGE** (-1) then that value will be left unchanged.

BINDING

```
move.w      mode,-(sp)
pea         phys
pea         log
move.w      #$5,-(sp)
trap        #14
lea         12(sp),sp
```

CAVEATS Changing screen modes with this call does not reinitialize the **AES**. The **VDI** and **VT52** emulator are, however, correctly reinitialized. The **AES** should not be used after changing screen mode with this call until the old screen mode is restored.

COMMENTS The Atari ST and Mega ST required that its physical screen memory be on a 256 byte boundary. All other Atari computers only require a **WORD** boundary.

To access the unique video modes of the Falcon030 the call **VsetScreen()** (which is actually an alternate binding of this call with the same opcode) should be used in place of this call.

SEE ALSO **VsetMode(), VsetScreen(), EsetShift()**

Settime()

VOID Settime(***time*** **)**
LONG *time*;

Settime() sets a new **IKBD** date and time.

OPCODE 22 (0x16)

AVAILABILITY All **TOS** versions.

PARAMETERS *time* is a **LONG** bit array defined as follows:

Bits	Meaning
0-4	Seconds / 2 (0-29)
5-10	Minute (0-59)
11-15	Hour (0-23)
16-20	Day (1-31)
21-24	Month (1-12)
25-31	Year - 1980 (0-127)

The value can be represented in a C structure as follows:

```
typedef struct
{
       unsigned year:7;
       unsigned month:4;
       unsigned day:5;
       unsigned hour:5;
       unsigned minute:6;
       unsigned second:5;
```

```
} BIOS_TIME;
```

BINDING

```
move.l      time,-(sp)
move.w      #$16,-(sp)
trap        #14
addq.l      #6,sp
```

COMMENTS As of **TOS** 1.02, this function also updates the **GEMDOS** time.

SEE ALSO **Gettime(), Tsettime(), Tsetdate()**

Settracks()

LONG Settracks(***playtracks, rectracks*** **)**
WORD ***playtracks, rectracks*****;**

Setttracks() sets the number of recording and playback tracks.

OPCODE 133 (0x85)

AVAILABILITY Available only when bit #2 of the '_SND' cookie is set.

PARAMETERS *playtracks* specifies the number of playback tracks (0-3) and *rectracks* specifies the number of recording tracks.

BINDING

```
move.w      rectracks,-(sp)
move.w      playtracks,-(sp)
move.w      #$85,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Settracks()** returns 0 if the operation was successful or non-zero otherwise.

COMMENTS The tracks specified are stereo tracks. When in 8-bit Mono mode, two samples are read at a time.

SEE ALSO **Setmode(), Setmontracks()**

Sndstatus()

LONG Sndstatus(***reset*** **)**
WORD ***reset*****;**

Sndstatus() can be used to test the error condition of the sound system and to completely reset it.

OPCODE 140 (0x8C)

AVAILABILITY Available only when bit #2 of the '_SND' cookie is set.

PARAMETERS *reset* is a flag indicating whether the sound system should be reset. A value of **SND_RESET** (1) will reset the sound system.

BINDING

```
move.w      reset,-(sp)
move.w      #$8C,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Sndstatus()** returns a **LONG** bit array indicating the current error status of the sound system defined as follows:

Bit(s)	Meaning
0-3	These bits form a value indicating the error condition of the sound system as follows: **Name** / **Mask** / **Meaning** **SND_ERROR** / 0xF / Use to mask error code **Name** / **Value** / **Meaning** **SND_OK** / 0 / No Error **SND_BADCONTROL** / 1 / Invalid Control Field **SND_BADSYC** / 2 / Invalid Sync Format **SND_BADCLOCK** / 3 / Clock out of range
4	If this bit is set, left channel clipping has occurred. Use the mask **SND_LEFTCLIP** (0x10) to isolate this bit.
5	If this bit is set, right channel clipping has occurred. Use the mask **SND_RIGHTCLIP** (0x20) to isolate this bit.
6-31	Unused.

COMMENTS On reset, the following things happen:

- DSP is tristated
- Gain and attentuation are zeroed
- Old matrix connections are reset
- ADDERIN is disabled

- Mode is set to 8-Bit Stereo
- Play and record tracks are set to 0
- Monitor track is set to 0
- Interrupts are disabled
- Buffer operation is disabled

Soundcmd()

LONG Soundcmd(*mode, data*)
WORD *mode, data*;

Soundcmd() sets various configuration parameters in the sound system.

OPCODE 130 (0x82)

AVAILABILITY Available only when bit #2 of '_SND' cookie is set.

PARAMETERS *mode* specifies how *data* is interpreted as follows:

Name	*mode*	Meaning
LTATTEN	0	Set the left attenuation (increasing attentuation is the same as decreasing volume). *data* is a bit mask as follows: XXXX XXXX LLLL XXXX 'L' specifies a valid value between 0 and 15 used to set the attenuation of the left channel in -1.5db increments. The bits represented by 'X' are reserved and should be 0.
RATTEN	1	Set the right attentuation. *data* is a bit mask as follows: XXXX XXXX RRRR XXXX 'R' specifies a valid value between 0 and 15 used to set the attenuation of the right channel in -1.5db increments. The bits represented by 'X' are reserved and should be 0.
LTGAIN	2	Set the left channel gain (boost the input to the ADC). *data* is a bit mask as follows: XXXX XXXX LLLL XXXX 'L' specifies a valid value between 0 and 15 used to set the gain of the left channel in 1.5db increments. The bits represented by 'X' are reserved and should be 0.

RTGAIN	3	Set the right channel gain (boost the input to the ADC). *data* is a bit mask as follows: XXXX XXXX RRRR XXXX 'R' specifies a valid value between 0 and 15 used to set the gain of the right channel in 1.5Db increments. The bits represented by 'X' are reserved and should be 0.
ADDERIN	4	Set the 16 bit ADDER to receive its input from the source(s) specified in *data*. *data* is a bit mask where each bit indicates a possible souce. Bit 0 represents the ADC (**ADDR_ADC**). Bit 1 represents the connection matrix (**ADDR_MATRIX**). Setting either or both of these bits determines the source of the ADDER.
ADCINPUT	5	Set the inputs of the left and right channels of the ADC. *data* is a bit mask with bit 0 being the right channel: **LEFT_MIC** (0x00) or **LEFT_PSG** (0x02) and bit 1 being the left channel: **RIGHT_MIC** (0x00) or **RIGHT_PSG** (0x01). Setting a bit causes that channel to receive its input from the Yamaha PSG. Clearing a bit causes that channel to receive its input from the microphone.
SETPRESCALE	6	This mode is only valid when **Devconnect()** is used to set the prescaler to TT030 compatibility mode. In that case, *data* represents the TT030 compatible prescale value as follows: **Name** **Value** **Meaning** **CCLK_6K** 0 Divide by 1280 (6.25 MHz) **CCLK_12K** 1 Divide by 640 (12.5 Mhz) **CCLK_25K** 2 Divide by 320 (25 MHz) **CCLK_50K** 3 Divide by 160 (50 MHz)

Setting *data* to **SND_INQUIRE** (-1) with any command will cause that command's current value to be returned and the parameter unchanged.

BINDING

```
move.w      data,-(sp)
move.w      mode,-(sp)
move.w      #$82,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE

Soundcmd() returns the prior value of the specified command if data is **SND_INQUIRE** (-1).

Using the **SETPRESCALE** mode to set a frequency of 6.25 MHz (**CCLK_6K**) will cause the sound system to mute on a Falcon030 as it does not support this sample rate.

CAVEATS

On current systems, a bug exists that causes a *mode* value of **LTGAIN** to set the gain for both channels.

SEE ALSO

Devconnect()

Ssbrk()

VOIDP Ssbrk(*len*)
WORD *len*;

Ssbrk() is designed to reserve memory at the top of RAM prior to the initialization of **GEMDOS**.

OPCODE 1 (0x01)

AVAILABILITY All **TOS** versions.

PARAMETERS *len* is a **WORD** value specifying the number of bytes to reserve at the top of RAM.

BINDING

```
move.w      len,-(sp)
move.w      #$01,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **Ssbrk()** returns a pointer to the allocated block.

CAVEATS **Ssbrk()** was only used on early development systems. Currently the function is unimplemented and does not do anything.

Supexec()

LONG Supexec(*func*)
LONG (func*)(VOID);**

Supexec() executes a user-defined function in supervisor mode.

OPCODE 38 (0x26)

AVAILABILITY All **TOS** versions.

PARAMETERS *func* is the address to a function which will be called in supervisor mode.

BINDING

```
pea         func
move.w      #$26,-(sp)
trap        #14
addq.l      #6,sp
```

RETURN VALUE **Supexec()** returns the **LONG** value returned by the user function.

CAVEATS Care must be taken when calling the operating system in supervisor mode. The **AES** must not be called while in supervisor mode.

SEE ALSO **Super()**

Unlocksnd()

LONG Unlocksnd(VOID)

Unlocksnd() unlocks the sound system so that other applications may utilize it.

OPCODE 129 (0x81)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$81,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **Unlocksnd()** returns a 0 if the sound system was successfully unlocked or **SNDNOTLOCK** (-128) if the sound system wasn't locked prior to the call.

SEE ALSO **Locksnd()**

VgetMonitor()

WORD VgetMonitor(VOID)

VgetMonitor() returns a value which determines the kind of monitor currently being used.

OPCODE 89 (0x59)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

BINDING

```
move.w      #$59,-(sp)
trap        #14
addq.l      #2,sp
```

RETURN VALUE **VgetMonitor()** returns a value describing the monitor currently connected to the system as follows:

Name	Return Value	Monitor Type
MON_MONO	0	ST monochrome monitor
MON_COLOR	1	ST color monitor
MON_VGA	2	VGA monitor
MON_TV	3	Television

VgetRGB()

VOID VgetRGB(***index, count, rgb*** **)**
WORD ***index*****,** ***count*****;**
RGB ****rgb*;**

VgetRGB() returns palette information as 24-bit **RGB** data.

OPCODE 94 (0x5E)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS *index* specifies the beginning color index in the palette to read data from. *count* specifies the number of palette entries to read. *rgb* is a pointer to an array of **RGB**s which will be filled in by the functions. **RGB** is defined as:

```
typedef struct
{
        BYTE reserved;
        BYTE red;
        BYTE green;
        BYTE blue;
} RGB;
```

BINDING

```
pea         rgb
move.w      count,-(sp)
move.w      index,-(sp)
move.w      #$5E,-(sp)
trap        #14
lea         10(sp),sp
```

COMMENTS **VgetRGB()** is device-dependent in nature and it is therefore recommended that **vq_color()** be used instead.

SEE ALSO **VsetRGB()**

VgetSize()

LONG VgetSize(***mode*** **)**
WORD ***mode*****;**

VgetSize() returns the size of a screen mode in bytes.

OPCODE 91 (0x5B)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS *mode* is a modecode as defined in **VsetMode()**.

BINDING

```
move.w      mode,-(sp)
move.w      #$5B,-(sp)
trap        #14
addq.l      #4,sp
```

RETURN VALUE **VgetSize()** returns the size in bytes of a screen mode of type *mode*.

VsetMask()

VOID VsetMask(***ormask, andmask, overlay*** **)**
LONG ***ormask, andmask*****;**
WORD ***overlay*****;**

VsetMask() provides access to 'overlay' mode.

OPCODE 146 (0x92)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS When the **VDI** processes a **vs_color()** call. It converts the desired color into a hardware palette register. In 16-bit true-color mode, this is a **WORD** formatted as follows:

```
RRRR RGGG GGXB BBBB
```

The 'X' is the system overlay bit. In 24-bit true color a **LONG** is formatted as follows:

```
XXXXXXXX RRRRRRRR GGGGGGGG BBBBBBBB
```

VsetMask() sets a logical OR and AND mask which are applied to this register

before being stored. The default system value for *ormask* is 0x00000000 and the default value for *andmask* is 0xFFFFFFFF.

overlay should be **OVERLAY_ON** (1) to enable overlay mode or **OVERLAY_OFF** (0) to disable it.

BINDING

```
move.w   #overlay,-(sp)
move.l   #andmask,-(sp)
move.l   #ormask,-(sp)
move.w   #$92,-(sp)
trap     #14
add.l    #12,sp
```

COMMENTS

To make colors defined by the **VDI** transparent in 16-bit true color with overlay mode enabled, use an *andmask* value of 0xFFFFFFDF and an *ormask* value of 0x00000000. To make colors visible, use an *andmask* of 0x00000000 and an *ormask* of 0x00000020.

VsetMode()

WORD VsetMode(*mode*)
WORD *mode*;

VsetMode() places the video shifter into a specific video mode.

OPCODE

88 (0x58)

AVAILABILITY

Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS

mode is a **WORD** bit array arranged as follows:

Name	Bit(s)	Meaning
BPS1 (0x00) **BPS2** (0x01) **BPS4** (0x02) **BPS8** (0x03) **BPS16** (0x04)	0-2	These bits form a value so that 2 ^ X represents the number of bits per pixel.
COL80 (0x08) **COL40** (0x00)	3	80 Column Flag (if set, 80 columns, otherwise 40)
VGA (0x10) **TV** (0x00)	4	VGA Flag (if set, VGA mode will be used, otherwise television/monitor mode)
PAL (0x20) **NTSC** (0x00)	5	PAL Flag (if set, PAL will be used, otherwise NTSC)
OVERSCAN (0x40)	6	Overscan Flag (not valid with VGA)
STMODES (0x80)	7	ST Compatibility Flag
VERTFLAG (0x100)	8	Vertical Flag (is set, enables interlace mode on a color monitor or double-line mode on a VGA monitor)

–	9-15	Reserved (set to 0)

If *mode* is **VM_INQUIRE** (-1) then the current mode code is returned without changing the current settings.

BINDING

```
move.w      mode,-(sp)
move.w      #$58,sp
trap        #14
addq.l      #4,sp
```

RETURN VALUE **VsetMode()** returns the prior video mode.

CAVEATS **VsetMode()** does not reset the video base address, reserve memory, or reinitialize the **VDI**. To do this, use **VsetScreen()**.

COMMENTS Some video modes are not legal. 40 column monoplane modes and 80 column VGA true color modes are not supported.

SEE ALSO **VsetScreen(), Setscreen()**

VsetRGB()

VOID VsetRGB(*index, count, rgb*)
WORD *index, count*;
RGB **rgb*;

VsetRGB() sets palette registers using 24-bit **RGB** values.

OPCODE 93 (0x5D)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS *index* specifies the first palette index to modify. *count* specifies the number of palette entries to modify. *rgb* is a pointer to an array of **RGB** elements which will be copied into the palette.

BINDING

```
pea         rgb
move.w      count,-(sp)
move.w      index,-(sp)
move.w      #$5D,-(sp)
trap        #14
lea         10(sp),sp
```

COMMENTS This call is device-dependent by nature. It is therefore recommended that **vs_color()** be used instead.

SEE ALSO **VgetRGB(), EsetPalette(), Setpalette(), vs_color()**

VsetScreen()

VOID VsetScreen(*log*, *phys*, *mode*, *modecode*)
VOIDP *log*, *phys*;
WORD *mode*, *modecode*;

VsetScreen() changes the base addresses and mode of the current screen.

OPCODE 5 (0x05)

AVAILABILITY All **TOS** versions. The ability of this call to utilize the *modecode* parameter and the memory allocation feature is limited to systems having a '_VDO' cookie with a value of 0x00030000 or greater.

PARAMETERS *log* is the address for the new logical screen base. *phys* is the new address for the physical screen base. If either *log* or *phys* is **NULL**, the **XBIOS** will allocate a new block of memory large enough for the current screen and reset the parameter accordingly.

mode defines the screen mode to switch to (same as **Getrez()**). Setting *mode* to **SCR_MODECODE** (3) will cause *modecode* to be used to set the graphic mode (see **VsetMode()** for valid values for this parameter), otherwise *modecode* is ignored. If any of these three parameters is set to **SCR_NOCHANGE** (-1) then that value will be left unchanged.

BINDING

```
move.w      modecode,-(sp)
move.w      mode,-(sp)
pea         phys
pea         log
move.w      #$05,-(sp)
trap        #14
lea         14(sp),sp
```

CAVEATS Changing screen modes with this call does not reinitialize the **AES**. The **VDI** and **VT52** emulator are, however, correctly reinitialized. The **AES** should not be used after changing screen mode with this call until the old screen mode is restored.

COMMENTS TOS 1.00 and 1.02 required that its physical screen memory be on a 256 byte boundary. All other Atari computers only require a **WORD** boundary.

This call is actually a revised binding of **Setscreen()** developed to allow access to the newly available modecode parameter.

SEE ALSO **Setscreen(), VsetMode()**

VsetSync()

VOID VsetSync(***external*** **)**
WORD *external*;

VsetSync() sets the external video sync mode.

OPCODE 90 (0x5A)

AVAILABILITY Available if the '_VDO' cookie has a value of 0x00030000 or greater.

PARAMETERS *external* is a **WORD** bit array defined as follows:

Name	Bit	Meaning
VCLK_EXTERNAL	0	Use external clock.
VCLK_EXTVSYNC	1	Use external vertical sync.
VCLK_EXTHSYNC	2	Use external horizontal sync.
–	3-15	Reserved (set to 0)

BINDING

```
move.w      external,-(sp)
move.w      #$5A,-(sp)
trap        #14
addq.l      #2,sp
```

CAVEATS This call only works in Falcon video modes, not in compatibility or any four color modes.

Vsync()

VOID Vsync(VOID)

Vsync() pauses program execution until the next vertical blank interrupt.

OPCODE 37 (0x25)

AVAILABILITY All **TOS** versions.

BINDING

```
move.w      #$25,-(sp)
trap        #14
addq.l      #2,sp
```

WavePlay()

WORD WavePlay(*flags, rate, sptr, slen*)
WORD *flags*;
LONG *rate*;
VOIDP *sptr*;
LONG *slen*;

WavePlay() provides a easy method for applications to utilize the DMA sound system on the STe, TT030, and Falcon030 and playback user-defined event sound effects.

OPCODE 165 (0xA5)

AVAILABILITY Available only when the 'SAM\0' cookie exists.

PARAMETERS *flags* is a bit mask consisting of the following options:

Name	Mask	Meaning
WP_MONO	0x00	The sound to be played back is monophonic.
WP_STEREO	0x01	The sound to be played back is in stereo.
WP_8BIT	0x00	The sound to be played back was sampled at 8-bit resolution.
WP_16BIT	0x02	The sound to be played back was sampled at 16-bit resolution.
WP_MACRO	0x100	Play back a user-assigned macro or application global sound effect. This flag is exclusive and modifies the meaning of the other parameters to this call as shown below.

rate specifies the sample rate in Hertz (for example 49170L to play back at 49170 Hz). If **WP_MACRO** was specified in *flags*, then this parameter is ignored and should be set to 0L.

sptr is a pointer to the sound sample in memory. If **WP_MACRO** was specified in *flags* then this parameter should be a **LONG** containing either the application cookie specified in the .SAA file or the 'SAM\0' cookie to play an application global.

slen is the length of the sample in bytes. If **WP_MACRO** was specified in *flags* then *slen* is the macro or application global index as specified in the .SAA file. Valid application global values are as follows:

Name	*slen*	Usage

AG_FIND	0	Call **WavePlay()** with this value when the user requests display of the 'Find' dialog box.
AG_REPLACE	1	Call **WavePlay()** with this value when the user requests display of the 'Replace' dialog box.
AG_CUT	2	Call **WavePlay()** with this value when the user requests a 'Cut' operation.
AG_COPY	3	Call **WavePlay()** with this value when the user requests a 'Copy' operation.
AG_PASTE	4	Call **WavePlay()** with this value when the user requests a 'Paste' operation.
AG_DELETE	5	Call **WavePlay()** with this value when the user requests a 'Delete' operation. This should not be called when the user presses the 'Delete' key.
AG_HELP	6	Call **WavePlay()** with this value when the user requests display of application 'Help.' This should not be called when the user presses the 'Help' key.
AG_PRINT	7	Call **WavePlay()** with this value when the user requests display of the 'Print' dialog box.
AG_SAVE	8	Call **WavePlay()** with this value when the user requests that the current document be saved. This should not be used for any operation that calls the file selector.
AG_ERROR	9	Call **WavePlay()** with this value when the application encounters an error not presented to the user in an alert or error dialog (error dialogs may be assigned sounds).
AG_QUIT	10	Call **WavePlay()** with this value when the user requests that the application exit. Use this global after the user has confirmed a quit with any dialog box that may have been necessary.

BINDING

```
move.l      slen,-(sp)
pea         sptr
move.l      rate,-(sp)
move.w      flags,-(sp)
move.w      #$A5,-(sp)
trap        #14
lea         16(sp),sp
```

RETURN VALUE

WavePlay() returns **WP_OK** (0) if the call was successful, **WP_ERROR** (-1) if an error occurred, or **WP_NOSOUND** (1) to indicate that no sound was played (either because the user had not previously assigned a sound to the given macro or SAM was disabled).

CAVEATS

This function is only available when the System Audio Manager TSR (available from Atari Corp. or SDS) is installed. Extended development information is available online the Atari Developer's roundtable on GEnie.

Because of previously misdocumented sample rates, the value for rate must be 33880 to play back a sample at 32880 Hz, 20770 to play back a sample at 19668 Hz, and 16490 to play back a sample at 16390 Hz.

COMMENTS Even if an application does not install any custom events in a .SAA file, an application must still provide a .SAA file if it wishes to use application globals so that the SAM configuration accessory allows the user to assign those sounds.

A macro is commonly used to access the application global sounds available as follows:

```
#define WavePlayMacro(a)  WavePlay( WP_MACRO, 0L, SAM_COOKIE, a
       );
```

Xbtimer()

VOID Xbtimer(***timer, control, data, hand*** **)**
WORD ***timer, control, data*****;**
VOID (***hand*****)(VOID);**

Xbtimer() sets an interrupt on the 68901 chip.

OPCODE 31 (0x1F)

AVAILABILITY All **TOS** versions.

PARAMETERS *timer* is a value defining which timer to set as follows:

Name	Timer	Meaning
XB_TIMERA	0	Timer A (DMA sound counter)
XB_TIMERB	1	Timer B (Hblank counter)
XB_TIMERC	2	Timer C (200Hz system clock)
XB_TIMERD	3	Timer D (RS-232 baud rate generator)

control is placed into the control register of the timer. *data* is placed in the data register of the timer. *hand* is a pointer to the interrupt handler which is called by the interrupt.

BINDING

```
pea         hand
move.w      data,-(sp)
move.w      control,-(sp)
move.w      timer,-(sp)
move.w      #$1F,-(sp)
trap        #14
lea         12(sp),sp
```

SEE ALSO **Mfpint(), Jenabint(), Jdisint()**