– CHAPTER 3 –

BIOS

Overview

The Basic Input/Output System (**BIOS**) is responsible for the lowest level of communications between the operating system and hardware devices. This chapter will document the operating system functions of the **BIOS** and other system level operations.

System Startup

Upon a cold or warm boot[1], microprocessors in the 680x0 series load the initial supervisor stack pointer from the first longword in memory ($0) and begin execution at the PC found in the second longword ($4). The location this points to is the base initialization point for Atari computers.

Every Atari computer follows a predefined set of steps to accomplish system initialization. The following illustrates these steps leaving out some hardware initialization which is specific to the particular computer line (ST, TT, Falcon, etc.).

- The Interrupt Priority Level (IPL) is set to 7 and the OS switches to supervisor mode.
- A RESET instruction is executed to reset external hardware devices.
- The presence of a diagnostic cartridge is determined. If one is inserted, it is JMP'ed to with a return address in register A6.
- If running on a 68030, the CACR, VBR, TC, TT0, and TT1 registers are initialized.
- If a floating-point coprocessor is present it is initialized.
- If the *memvalid* ($420), *memval2* ($43A), and *memval3* ($51A) system variables are all valid, a warm boot is assumed and the memory controller is initialized with the value from *memcntrl* ($424).
- The initial color palette registers are loaded and the screen base is initialized to $100000.
- Memory is sized if it wasn't from a previous reset.
- Magic numbers are stored in low memory to indicate the successful sizing and initialization of memory.
- System variables and the cookie jar are initialized.
- The **BIOS** initialization point is executed.
- Installed cartridges of type 2 are executed.

[1] A cold boot occurs when the computer system experiences a total loss of power and no memory locations can be considered valid (this can be done artificially by zeroing memory, as is the case with the CTRL-ALT-RSHIFT-DELETE reset). A warm boot is a manual restart of the system which can be accomplished via software (like the CTRL-ALT-DELETE reset) or the external reset button found on some machines.

- The screen resolution is programmed.
- Installed cartridges of type 0 are executed.
- Interrupts are enabled by lowering the IPL to 3.
- Installed cartridges of type 1 are executed.
- The **GEMDOS** initialization point is executed.
- On systems running **TOS** 2.06 or **TOS** 3.06 and above, the Fuji logo is displayed and a memory test and hard disk spin-up sequence is executed.
- If at least one floppy drive is attached to the system, the first sector of the first floppy drive is loaded, and if executable, it is called.
- If at least one hard disk or other media is attached to the system, the first sector of each is loaded in succession until one with an executable sector is found or each has been tried.
- If a hard disk sector was found that was executable, it is executed.
- The text cursor is enabled.
- All "\AUTO*.PRG" files found on the boot disk are executed.
- If *_cmdload* ($482) is 0 then an environment string is created and the **AES** is launched, otherwise "\COMMAND.PRG" is loaded.
- If the **AES** ever terminates, the system is reset and system initialization begins again.

OS Header

The address of the start of operating system is stored in the system variable *_sysbase* ($4F2). The beginning of the operating system contains a table with contents as follows:

Offset (*_sysbase* + $x)	Size	Contents
$0	**WORD**	*os_entry*: BRA to reset hander (shadowed at $0).
$2	**WORD**	*os_version*: **TOS** version number. The high byte is the major revision number, and the low byte is the minor revision number.
$4	**LONG**	*reseth*: Pointer to the system reset handler.
$8	**LONG**	*os_beg*: Base address of the OS (same as *sysbase*).
$C	**LONG**	*os_end*: Address of the first byte of RAM not used by the operating system.
$10	**LONG**	*os_rsv1*: Reserved
$14	**LONG**	*os_magic*: Pointer to the **GEM** Memory Usage Parameter Block (**MUPB**). See below for more information.
$18	**LONG**	*os_date*: Date of system build ($YYYYMMDD).
$1C	**WORD**	*os_conf*: OS Configuration Bits. See below for more information.
$1E	**LONG**	*os_dosdate*: **GEMDOS** format date of system build.

$20	**LONG**	*p_root*: Pointer to a system variable containing the address of the **GEMDOS** memory pool structure. This entry is available as of **TOS** 1.2. The location pointed to by this value should never be modified by an application.
$24	**LONG**	*p_kbshift*: Pointer to a system variable which contains the address of the system keyboard shift state variable. See below for more information. This entry is available as of **TOS** 1.02. This location should never be modified by an application.
$28	**LONG**	*p_run*: Pointer to a system variable which contains the address of the currently executing **GEMDOS** process. See below for more information. This entry is available as of **TOS** 1.02. The information pointed to by this variable should never be modified by an application.
$2C	**LONG**	*p_rsv2*: Reserved

Some versions of AHDI (the Atari Hard Disk Interface) contain a bug which copies the system header to RAM and then corrupts some portions of it. The following 'C' structure definition defines the **OSHEADER** structure. The function GetROMSysbase() can be used to return an **OSHEADER** pointer to the code in ROM. GetROMSysbase() will execute properly in either user or supervisor mode.

```
typedef struct _osheader
{
       UWORD     os_entry;
       UWORD     os_version;
       VOID     *reseth;
       struct _osheader *os_beg;
       char     *os_end;
       char     *os_rsv1;
       char     *os_magic;
       LONG     os_date;
       UWORD    os_conf;
       UWORD    os_dosdate;

       /* Available as of TOS 1.02 */
       char     **p_root;
       char     **p_kbshift;
       char     **p_run;
       char     *p_rsv2;
} OSHEADER;

#define _sysbase        ((OSHEADER **)0x4F2)

OSHEADER *
GetROMSysbase( VOID )
{
       OSHEADER *osret;
       char *savesp = (Super(SUP_INQUIRE) ? NULL : Super(SUP_SET));

       osret = (*_sysbase)->os_beg;

       if( savesp )
           Super( savesp );

       return osret;
}
```

OS Configuration Bits

os_conf contains the country code and video sync mode that the operating system was compiled for. Bit #0 of this variable is 0 to indicate NTSC video mode or 1 to indicate PAL. The remaining bits, when shifted right by one bit, yield the country code as follows:

os_conf >> 1	Country
0	USA
1	Germany
2	France
3	United Kingdom
4	Spain
5	Italy
6	Sweden
7	Switzerland (French)
8	Switzerland (German)
9	Turkey
10	Finland
11	Norway
12	Denmark
13	Saudi Arabia
14	Holland
15	Czechoslovakia
16	Hungary
127	All countries are supported. As of **TOS** 4.0 the OS is compiled with text for all languages and switches between them based on the country code stored in non-volatile RAM. Use the '_AKP' cookie to determine the actual language in use.

GEM Memory Usage Parameter Block

The pointer at offset $14 in the OS header points to the **GEM** Memory Usage Parameter Block which is defined as follows:

```
typedef struct
{
        /* $87654321 if GEM present */
        LONG gem_magic;

        /* End address of OS RAM usage */
        LONG gem_end;

        /* Execution address of GEM */
        LONG gem_entry;
} MUPB;
```

GEM is only launched at system startup if *gem_magic* is $87654321. The **XBIOS** call **Puntaes()** also uses this information to restart the operating system after clearing **GEM** (only if disk-based). It verifies that *gem_magic* was valid and that **GEM** was in RAM, then it modifies *gem_magic* and restarts the operating system.

Keyboard Shift State Variable

The OS header entry *p_kbshift* provides a method of reading the state of the keyboard shift state variables more quickly than with **Kbshift()**. This header entry did not exist in **TOS** 1.0. The following code provides an acceptable method for accessing this variable in all **TOS** versions:

```
#define Kbstate    *p_kbshift

char *p_kbshift;

VOID
init_kbshift( VOID )
{
       /* See above for GetROMSysbase() definition. */
       OSHEADER *os = GetROMSysbase();

       if ( os->os_version == 0x0100)
          p_kbshift = (char *)0xE1BL;
       else
          p_kbshift = *(char **)os->p_kbshift;
}
```

Currently Running Process

The OS header entry *_p_run* is used to locate the address of the basepage of the currently running process. This entry has only existed as of **TOS** 1.02 and should never be modified. The following routine returns the address of the basepage of the currently running process in all versions of **TOS**:

```
#define SPAIN      4
typedef long PID

PID *
get_run()
{
       OSHEADER *os = GetROMSysbase();

       if(os->os_version < 0x0102)
       {
          if(( os->os_conf >> 1 ) == SPAIN)
               return (PID *)0x873C;
          else
               return (PID *)0x602C;
       }
       else
          return (PID *)(os->p_run);
}
```

The Cookie Jar

Overview

The 'Cookie Jar' is a structure in memory containing entries called 'cookies' which are placed in the 'jar' by the operating system or Terminate and Stay Resident (TSR) applications. Applications can test for the presence of a cookie to determine the presence of a hardware device or system feature.

The location of the cookie jar is determined by the address contained in the system variable *_p_cookies* ($5A0). If no cookie jar has been allocated yet, this entry will contain **NULL** (0).

Structure

The variable *_p_cookies* points to multiple **COOKIE** structures as defined below:

```
typedef struct
{
    LONG cookie;
    LONG value;
} COOKIE;
```

The structure member *cookie* contains a value that hopefully uniquely identifies the cookie. *cookie* values are 4-byte packed longword identifiers (often a 4 letter ASCII code word). Entries with the high byte equal to $5F, the underscore character, are reserved for use by Atari.

The structure member *value* may contain any value meaningful to an application or no value at all. In some cases a cookie won't have a meaningful value and its presence simply signals the existence of another process or system feature. TSR's often use *value* to store a pointer to an internal structure. The operating system uses cookies to signal the availability of hardware devices or system features.

The end of the cookie jar is signaled with a final entry with the value for *cookie* equaling **NULL**. The *value* entry for this final cookie contains the number of entries possible without reallocating the jar.

Searching for a Cookie

The following code may be used to find a cookie in the cookie jar. It returns 0 if an error occurred or 1 if successful. If *p_value* is non-**NULL** on entry, the address it points to will be filled in with the value of the cookie.

```
WORD
getcookie( target, p_value )
LONG target;
LONG *p_value;
{
      char *oldssp;
      COOKIE *cookie_ptr;

      oldssp = (Super(SUP_INQUIRE) ? NULL : Super(1L));
```

```
        cookie_ptr = *(COOKIE **)0x5A0;

        if(oldssp)
            Super( oldssp );

        if(cookie_ptr != NULL)
        {
            do
            {
                if(cookie_ptr->cookie == target)
                {
                    if(p_value != NULL)
                        *p_value = cookie_ptr->value;

                    return 1;
                }
            } while((cookie_ptr++)->cookie != 0L);
        }

        return 0;
}
```

Placing a Cookie

Only TSR programs should place cookies in the cookie jar. The cookie these programs place should either signal a function provided by the TSR or the presence of an expansion device. A CPX, desk accessory, or standard application should not place cookies in the jar.

To place a cookie, the TSR must first locate the current location of the cookie jar. It is possible that a cookie jar does not exist (*_p_cookies* == 0). In that case, a new jar should be allocated.

In most instances, the cookie jar should be allocated in increments of 8 slots (though it is not a requirement). In addition, if the process installs a new cookie jar in a **TOS** version lower than 1.06 it is also the processes responsibility to remove it upon a warm reset. Calling the following code after installing the cookie jar for the first time will ensure that the cookie jar pointer is properly reset on a warm boot.

```
RESMAGIC      equ     $31415926
_resvalid     equ     $426
_resvector        equ     $42A
_p_cookies        equ     $5A0

                  .globl  _unjar

_unjar:           move.l  _resvalid,valsave
                  move.l  _resvector,vecsave
                  move.l  #reshand,_resvector
                  move.l  #RESMAGIC,_resvalid
                  rts

reshand:          clr.l   _p_cookies
                  move.l  vecsave,_resvector
                  move.l  valsave,_resvalid
                  jmp     (a6)

                  .bss
```

```
vecsave:          .ds.l       1
valsave           .ds.l       1
```

After determining the location of the cookie jar, the application should search for the first empty slot in the jar by looking for a **NULL** in the *cookie* field of a slot. Next, the application must determine if this is the last slot in the jar by comparing the entry in the *value* field of the current cookie to the number of the actual slot you are comparing. For instance, if you have found **NULL** as the value for *cookie* in slot 16 and *value* is equal to 16, the jar is full and must be reallocated.

If the slot found is not the last one, the application can simply copy the current slot to the next slot and insert its own cookie.

If the jar must be reallocated, you should allocate enough memory to increase the size of the cookie jar, copy the old entries to the new jar, insert your entry as the last cookie in the jar, and finally terminate the jar with a cookie containing a **NULL** and the new number of slots you have allocated.

Though not mentioned previously, it is also advisable to ensure that your cookie isn't already in the jar before placing it to avoid two cookies for multiple executions of the same application to appear.

System Cookies

As of **TOS** 1.06, the operating system will place several cookies in the cookie jar to inform applications of certain operating system and hardware capabilities as follows:

<table>
<tr><th>cookie</th><th>value</th></tr>
<tr><td>_CPU</td><td>The low WORD of the CPU cookie contains a number representing the processor installed in the system as follows:
<table><tr><th>Value</th><th>Processor</th></tr><tr><td>0</td><td>68000</td></tr><tr><td>10</td><td>68010</td></tr><tr><td>20</td><td>68020</td></tr><tr><td>30</td><td>68030</td></tr></table></td></tr>
<tr><td>_VDO</td><td>This cookie represents the revision of the video shifter present. The low WORD represents the minor revision number and the high WORD represents the major revision number. Currently valid values are:
<table><tr><th>Major</th><th>Minor</th><th>Shifter</th></tr><tr><td>0</td><td>0</td><td>ST</td></tr><tr><td>1</td><td>0</td><td>STe</td></tr><tr><td>2</td><td>0</td><td>TT030</td></tr><tr><td>3</td><td>0</td><td>Falcon030</td></tr></table></td></tr>
</table>

_FPU

This cookie identifies the presence of floating-point math capabilities in the system. A non-zero low **WORD** indicates the presence of software floating point support (no specific values have yet been assigned). The high **WORD** indicates the type of coprocessor currently connected to the system as follows:

Value	Meaning
0	No FPU is installed.
1	SFP004
2	68881 or 68882
3	68881 or 68882 and SFP004
4	68881
5	68881 and SFP004
6	68882
7	68882 and SFP004
8	68040 Internal
9	68040 Internal and SFP004

_FDC

This cookie indicates the capability of the currently connected floppy drive. The lowest three bytes is a code indicating the origin of the unit ('ATC' is an Atari unit). The upper byte is a value indicating the highest density floppy present as follows:

Value	Density
0	360 Kb/ 720 Kb
1	1.44 Mb
2	2.88 Mb

_SND

This cookie contains a bitmap of sound features available to the system as follows:

Bit	Feature
0	GI Sound Chip (PSG)
1	Stereo 8-bit Playback
2	DMA Record (w/**XBIOS**)
3	16-bit CODEC
4	DSP

_MCH

This cookie indicates the machine type with the major revision number in the high **WORD** and the minor revision number in the low **WORD** as follows:

Major	Minor	Shifter
0	0	ST
1	0	STe
1	8	ST Book
1	16	Mega STe
2	0	TT030
3	0	Falcon030

_SWI

On machines that contain internal configuration dip switches, this value specifies their positions as a bitmap. Dip switches are generally used to indicate the presence of additional hardware which will be represented by other cookies.

_FRB

This cookie is present when alternative RAM is present. It points to a 64k buffer that may be used by DMA device drivers to transfer memory between alternative RAM and ST RAM for DMA operations.

_FLK

The presence of this cookie indicates that file and record locking extensions to **GEMDOS** exist. The *value* field is a version number currently undefined.

_NET

This cookie indicates the presence of networking software. The cookie value points to a structure which gives manufacturer and version information as follows:

```
struct netinfo
{
       LONG publisher;
       LONG version;
};
```

_IDT

This cookie defines the currently configured date and time format, Bits #0-7 contain the ASCII code of the date separator. Bits #8-11 contain a value indicating the date display format as follows:

Value	Meaning
0	MM-DD-YY
1	DD-MM-YY
2	YY-MM-DD
3	YY-DD-MM

Bits #12-15 contain a value indicating the time format as follows:

Value	Meaning
0	12 hour
1	24 hour

Note: The value of this cookie does not affect any of the internal time functions. It is intended for informational use by applications only.

_AKP

This cookie indicates the presence of an Advanced Keyboard Processor. The high word of this cookie is currently reserved. The low word indicates the language currently used by **TOS** for keyboard interpretation and alerts. See the explanation for the country code in the OS header earlier in this chapter for valid values.

If this cookie is present on TOS 5.0 and higher then the system supports soft-loaded keyboard tables.

FSMC

This cookie indicates the presence of **FSM** or **SpeedoGDOS**. Its *value* field is a pointer to a structure as follows:

```
typedef struct
{
    LONG    gdos_type;
    UWORD   version;
    WORD    quality;
} GDOS_INFO;
```

The *gdos_type* field determines the variety of **GDOS**. '**_FSM**' represents Imagen font-based **FSM** whereas '**_SPD**' represents Bitstream font-based **FSM**. *version* specifies the current GDOS version.

quality determines the output quality of **v_updwk()**. The default setting is **QUAL_DEFAULT** (0xFFFF) which causes the driver to use the setting last set in the driver configuration accessory or CPX. This default setting may be overridden by placing a value of **QUAL_DRAFT** (0x0000) or **QUAL_FINAL** (0x0001) at this location. The quality setting should be restored to **QUAL_DEFAULT** at the end of each print job.

SAM\0	This cookie indicates the presence of System Audio Manager and the **XBIOS** extensions it provides. The *value* field is currently reserved for internal use.
MiNT	This cookie indicates the presence of **MiNT** (**MultiTOS**) and its *value* field is the current version number (ex: **MiNT** 1.02 has a *value* field of 0x00000102).

BIOS Devices

The **BIOS** provides access to six default devices (numbered 0–5). In addition, **TOS** 2.00 provides the ability to add extra devices with the **XBIOS Bconmap()** function (see the **XBIOS** overview for more information). Device assignments higher than device five are dependent upon the machine and any third-party enhancements. The following list indicates the device assignments which remain constant:

Name	Device Number	GEMDOS Filename	Meaning
DEV_PRINTER	0	PRN:	Centronics Parallel Port
DEV_AUX	1	AUX:	Default Serial Device (this device number could actually refer to any serial device connected to the system depending on which was mapped with **Bconmap()**)
DEV_CON	2	CON:	Console (screen device)
DEV_MIDI	3	N/A	MIDI Ports
DEV_IKBD	4	N/A	Intelligent Keyboard Controller
DEV_RAW	5	N/A	Console (no interpretation)

The Console Device

Two methods are provided for outputting characters to the screen. Output via **BIOS** device #2 subjects character codes to interpretation. Codes such as a carriage return (ASCII 13), line feed (ASCII 10), TAB (ASCII 9), CTRL-G (ASCII 7), and ESCAPE (ASCII 27) are interpreted as special cases and handled specially.

Output via **BIOS** device #5 causes all characters to be output literally to the screen without interpretation.

The VT-52 Emulator

The Atari console device contains emulation code compatible with the VT-52 standard. Special escapes may be used to manipulate the cursor and create text effects.

To send an escape sequence, one of the following codes (and possibly additional characters) must be sent following the ESCAPE character (ASCII 27):

Escape	Code	Effect
A	65	Move the cursor up one line. If the cursor is on the top line this does nothing.
B	66	Move the cursor down one line. If the cursor is on the bottom line this does nothing.

C	67	Move the cursor right one line. If the cursor is on the far right of the screen this does nothing.
D	68	Move the cursor left one line. If the cursor is on the far left of the screen this does nothing.
E	69	Clear the screen and place the cursor at the upper-left corner.
H	72	Move the cursor to the upper-left corner of the screen.
I	73	Move the cursor up one line. If the cursor is on the top line, the screen scrolls down one line.
J	74	Erase the screen downwards from the current position of the cursor.
K	75	Clear the current line to the right from the cursor position.
L	76	Insert a line by scrolling all lines at the cursor position down one line.
M	77	Delete the current line and scroll lines below the cursor position up one line.
Y	89	Position the cursor at the coordinates given by the following two codes. The screen starts with coordinates (32, 32) at the upper-left of the screen. Coordinates should be presented in reverse order, Y and then X.
b	98	This code is followed by a character from which the lowest four bits determine a new text foreground color.
c	99	This code is followed by a character from which the lowest four bits determine a new text background color.
d	100	Erase the screen from the upper-left to the current cursor position.
e	101	Enable the cursor.
f	102	Disable the cursor.
j	106	Save the current cursor position. (Only implemented as of **TOS** 1.02)
k	107	Restore the current cursor position. (Only implemented as of **TOS** 1.02)
l	108	Erase the current line and place the cursor at the far left.
o	111	Erase the current line from the far left to the current cursor position.
p	112	Enable inverse video.
q	113	Disable inverse video.
v	118	Enable line wrap.
w	119	Disable line wrap.

Media Change

The **BIOS** function **Mediach()** returns the current media-change status of the drive specified. This state is used to determine if a disk has been changed in removable media drives (floppies, removable hard drives, etc.

The **Getbpb()** incorrectly resets the media change state. Failure to properly reset this state after calling **Getbpb()** can cause data loss. The function **_mediach()**, shown below, forces the **Mediach()** function to return a 'definitely changed' state and should always be called after calling **Getbpb()** on removable media drives.

```
/*
 * _mediach(): force the media 'changed' state on a removable drive.
 *
 * Usage: errcode = _mediach( devno )      - returns 1 if an error occurs
 *
 * Inputs: devno - (0 = 'A:', 1 = 'B:', etc...)
 *
```

```
 */

            .globl   _mediach

_mediach:
            move.w   4(sp),d0
            move.w   d0,mydev
            add.b    #'A',d0
            move.b   d0,fspec     ; Set drive spec for search
loop:
            clr.l    -(sp)        ; Get supervisor mode, leave old SSP
            move.w   #$20,-(sp)   ; and "Super" function code on stack.
            trap     #1
            addq.l   #6,sp
            move.l   d0,-(sp)
            move.w   #$20,-(sp)

            move.l   $472,oldgetbpb
            move.l   $47e,oldmediach
            move.l   $476,oldrwabs

            move.l   #newgetbpb,$472
            move.l   #newmediach,$47e
            move.l   #newrwabs,$476

            ; Fopen a file on that drive
            move.w   #0,-(sp)
            move.l   #fspec,-(sp)
            move.w   #$3d,-(sp)
            trap     #1
            addq.l   #8,sp

            ; Fclose the handle
            tst.l    d0
            bmi.s    noclose

            move.w   d0,-(sp)
            move.w   #$3e,-(sp)
            trap     #1
            addq.l   #4,sp

noclose:
            moveq    #0,d7
            cmp.l    #newgetbpb,$472      ; still installed?
            bne.s    done

            move.l   oldgetbpb,$472       ; Error, restore vectors.
            move.l   oldmediach,$47e
            move.l   oldrwabs,$476

            trap     #1                   ; go back to user mode
            addq.l   #6,sp                ; restore sp

            moveq.l  #1,d0                ; 1 = Error
            rts
done:
            trap     #1                   ; go back to user mode
            addq.l   #6,sp                ; from stack left above

            clr.l    d0                   ; No Error
```

```
            rts

/*
 * New Getbpb()...if it's the target device, uninstall vectors.
 * In any case, call normal Getbpb().
 */

newgetbpb:
            move.w   mydev,d0
            cmp.w    4(sp),d0
            bne.s    dooldg

            move.l   oldgetbpb,$472   ; Got target device so uninstall.
            move.l   oldmediach,$47e
            move.l   oldrwabs,$476
dooldg:     move.l   oldgetbpb,a0     ; Go to real Getbpb()
            jmp      (a0)

/*
 * New Mediach()...if it's the target device, return 2. Else call old.
 */

newmediach:
            move.w   mydev,d0
            cmp.w    4(sp),d0
            bne.s    dooldm
            moveq.l  #2,d0            ; Target device, return 2

            rts

dooldm:
            move.l   oldmediach,a0        ; Call old
            jmp      (a0)

/*
 * New Rwabs()...if it's the target device, return E_CHG (-14)
 */

newrwabs:
            move.w   mydev,d0
            cmp.w    4(sp),d0
            bne.s    dooldr
            moveq.l  #-14,d0
            rts

dooldr:
            move.l   oldrwabs,a0
            jmp      (a0)

            .data

fspec:      dc.b     "X:\\X",0
mydev:      ds.w     1
oldgetbpb:  ds.l     1
oldmediach: ds.l     1
oldrwabs:   ds.l     1

            .end
```

BIOS Vectors

Reset Vector

Shortly after a warm boot the OS will jump to the address contained in the system variable *resvector* ($42A) if the value in the system variable *resvalid* ($426) contains the magic number $31415926. The OS will supply a return address to this code segment in register A6 but the subroutine must not utilize the stack as neither stack pointer will be valid.

If your process needs to do cleanup in the event of a warm reset (see “Placing a Cookie” earlier in this chapter) the following code installs a user routine to accomplish this.

```
_resvalid     equ      $426
_resvector        equ      $42A
RESMAGIC      equ      $31415926

              .text

installres:
              move.l   _resvalid,oldvalid
              move.l   _resvector,oldvector
              move.l   #myresvec,_resvector
              move.l   #RESMAGIC,_resvalid
              rts
myresvec:
              *
              * Insert user code here
              *
              move.l   oldvector,_resvector
              move.l   oldvalid,_resvalid
              jmp      (a6)

              .bss

oldvector:        ds.l     1
oldvalid:     ds.l     1

              .end
```

System Bell Vector

As of **TOS** 1.06, the OS jumps through the address contained in the system variable *bell_hook* ($5AC) to ring the system bell. It is possible for a custom routine to hook into this vector to alter the bell sound. The user routine may modify registers D0-D2/A0-A2 and may chain to the old bell handler if desired. It is also safe to make **BIOS** and **XBIOS** calls following the procedure for calling from an interrupt (when not running under **MultiTOS**). The routine should either jump to the old handler or execute an RTS statement.

System Keyclick Vector

Similar to the system bell vector, another vector is called each time a keyclick sound is generated. This vector is stored in system variable *kcl_hook* ($5B0) and is entered with the keycode (not the ASCII code) of the key struck in the low byte of D0. Registers D1-D2/A0-A2 may be modified, however, all other registers including D0 must be maintained. The replacement handler may either chain to a new handler or RTS.

Deferred Vertical Blank Handlers

Applications may install custom routines which are called during every vertical blank (approx. 50-72 times per second). The OS performs several operations during the vertical blank as follows:

- The system variable *_frclock* is incremented.
- The system variable *vblsem* is tested. If 0, the vertical blank handler exits immediately.
- All registers are saved.
- The system variable *_vbclock* is incremented.
- If the system is currently in a high resolution video mode and a low-resolution monitor is detected, the video resolution is adjusted and the vector found at system variable *swv_vec* is called.
- The text cursor blink routine is called.
- If a new palette has been selected since the last vertical blank, it is loaded.
- If a new screen base address has been selected since the last vertical blank, it is selected.
- Each of the “deferred” vertical blank routine handlers is called.
- If the system variable *prt_cnt* is greater than -1, the vector at system variable *scr_dump* is called.
- Saved registers are restored and processing continues.

To install a routine to be called as a “deferred” vertical blank handler, you must inspect the list of handler vectors at *vblqueue* for a **NULL** slot, replace it with your vector and initialize the next slot to **NULL**. The system variable *nvbls* indicates the number of slots pointed to by

vblqueue. If the vertical blank handler list is filled, you may allocate a new area, copy the old list of handlers with your handler, and update the pointer *vblqueue* and *nvbls*.

The XBRA Protocol

Many applications that add functionality to the system do so by 'hooking' themselves into one or more interrupt or pass-through vectors (usually with **Setexc()**). Most vector handlers work by executing the relevant code when the interrupt is called and then calling the original vector handler. When several applications handle one vector, a vector 'chain' is created. This chain makes it difficult for debuggers or the process itself to 'unhook' itself from the chain.

The XBRA protocol was designed so that processes that wish to be able to unhook themselves may and so that debuggers can trace the 'chain' of vector handlers. Following the protocal is simple. Prior to the first instruction of the vector handler, insert three longwords into the application as follows:

- The longword 'XBRA' 0x58425241.
- Another longword containing the application 'cookie' ID (this is the same as that put into the cookie jar if applicable).
- A longword into which should be placed the address of the original handler.

The following code example shows how to correctly use the XBRA protocol in a routine designed to supplement the 680x0 TRAP #1 vector (**GEMDOS**):

```
instl_trap1:
             move.l       #my_trap1,-(sp)
             move.w       #VEC_GEMDOS,-(sp)
             move.w       #Setexc,-(sp)
             trap         #13
             addq.l       #8,sp
             move.l       d0,old_handler
             rts

             DC.L         'XBRA'
             DC.L         'SDS1'   ; Put your cookie here
old_handler  DC.L         0

my_trap1:
             movem.l      d2-d7/a2-a6,-(sp)

             ;
             ; Your TRAP #1 handler goes here.
             ;

             movem.l      (sp)+,d2-d7/a2-a6
             move.l       old_handler,-(sp)     ; Fake a
       return
             rts                                ; to old code.
```

The following ‘C’ function is an example of how to use the XBRA protocol to unhook a vector handler from the XBRA chain. This function will only work if all installed vector handlers follow the XBRA protocol. It takes a **Setexc()** vector number and an XBRA application id cookie as a parameter. It returns the address of the routine that was unhooked or 0L if unsuccessful.

```
typedef struct xbra
{
       LONG    xbra_id;
       LONG    app_id;
       VOID    (*oldvec)();
} XBRA;

LONG
unhook_xbra( WORD vecnum, LONG app_id )
{
       XBRA *rx;
       LONG vecadr, *stepadr, lret = 0L;
       char *savessp;

       vecadr = Setexc( vecnum, VEC_INQUIRE );
       rx = (XBRA *)(vecadr - sizeof( XBRA ));

       /* Set supervisor mode for search just in case. */
       savessp = Super( SUP_SET );

       /* Special Case: Vector to remove is first in chain. */
       if( rx->xbra_id == ‘XBRA’ && rx->app_id == app_id )
       {
          Setexc( vecnum, rx->oldvec );
          return vecadr;
       }

       stepadr = (LONG *)&rx->oldvec;
       rx = (XBRA *)((LONG)rx->oldvec - sizeof( XBRA ));
       while( rx->xbra_id == ‘XBRA’ )
       {
          if( rx->app_id == app_id )
          {
              *stepadr = lret = (LONG)rx->oldvec;
              break;
          }

          stepadr = (LONG *)&rx->oldvec;
          rx = (XBRA *)((LONG)rx->oldvec - sizeof( XBRA ));
       }

       Super( savessp );
       return lret;
}
```

BIOS Function Calling Procedure

BIOS system functions are called via the TRAP #13 exception. Function arguments are pushed onto the current stack (user or supervisor) in reverse order followed by the function opcode. The calling application is responsible for correctly resetting the stack pointer after the call.

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

The following example for **Bconout()** illustrates calling the **BIOS** from assembly language:

```
move.w      #char,-(sp)
move.w      #dev,-(sp)
move.w      #$03,-(sp)
trap        #13
addq.l      #6,sp
```

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

```
_bios:
       ; Save the return code from the stack
       move.l  (sp)+,trp13ret
       trap    #13
       move.l  trp13ret,-(sp)
       rts

       .bss
trp13ret:
       .ds.l       1
```

With the above code, you could easily design a 'C' macro to add **BIOS** calls to your compiler as in the following example for **Bconout()**:

```
#define Bconout( a )   bios( 0x02, a )
```

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

Calling the BIOS from an Interrupt

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

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

myhandler:
                sub.l   #savamt,savptr
```

```
; BIOS calls may be performed here
add.l    #savamt,savptr
rte      ; (or rts?)
```

This method is not valid under **MultiTOS**.