STE TOS Release Notes

January 12, 1990

Atari Corporation
1196 Borregas Avenue
Sunnyvale, CA 94086

COPYRIGHT

DISCLAIMER

ATARI CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Atari Corporation reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Atari Corporation to notify any person of such revision or changes

TRADEMARKS

Atari is a registered trademark of Atari Corporation. SLM804, ST, and TOS are trademarks of Atari Corporation.

This document was produced entirely with Microsoft Write, an Atari Mega 4 computer, and an Atari SLM804 laser printer.

Table of Contents

Introduction

Welcome to the STE TOS Release Notes!

This document describes the changes made in STE TOS. It also describes the **Cookie Jar**, a feature implemented in STE TOS that can be easily retrofitted to other versions, as well.

There is a section for each affected layer of TOS (Desktop, AES, and so on), and one for the Cookie Jar. We suggest you look through the Table of Contents, then read through the document in the order of your interests.

As always, we welcome your comments.

Desktop Changes

Show File:

- Error messages requiring keyboard input at the start of a Show File were removed. You now get an alert instead of a cleared screen with an error message.

- Mouse input was added to Show File. Pressing the left mouse button during show file is the equivalent of hitting the space bar, and pressing the right mouse button aborts (like Q, q, ^C or Undo).

AES Changes

Autoboot Applications: - The desktop background pattern is now set correctly before an autoboot application is launched. Previously, it would show a dither pattern in color resolutions, rather than the normal solid color desktop.

---

rsrc_load(),
shel_find(): - The `shel_name()` bug in Rainbow TOS, where the AES is sometimes unable to find a filename passed to `shel_find()`, was corrected in STE TOS.

VDI Changes

Color Mapping:

- `vq_color()` and `vs_color()` now have finer granularity in color representation, reflecting the change in the hardware.

  Note that the least significant bit of the STE intensity value is the most significant bit of the STE hardware register value. This is for ST compatibility to provide the closest mapping to the correct STE intensity for ST software that doesn't use 4 bit color values. Programs which correctly use VDI calls will get correctly scaled color values in any case. (One of the many advantages of writing your programs to be device independent!)

Color Palette Size:

- The palette size returned by `v_opnwk()` and `v_opnvwk()` in the workstation structure (`work_out[39]`) is now 4096, reflecting the change in the hardware.

Underlined Text:

- In some cases, the underline of underlined text would fall outside of the character cell. This would cause lines of text following underlined lines of text to overwrite and thus erase underlines. A check was added to ensure that the underline falls within the character cell so that it is not erased by subsequent lines of text.

BIOS/XBIOS Changes

Color:

- `Setpalette()` and `Setcolor()` support 4 bit color values.

  Note that the least significant bit of the STE intensity value is the most significant bit of the STE hardware register value. This is for ST compatibility to provide the closest mapping to the correct STE intensity for ST software that doesn't use 4 bit color values. Programs which correctly use VDI calls will get correctly scaled color values in any case. (One of the many advantages of writing your programs to be device independent!)

Console Bell and Keyclick:

- Hooks are provided for changing the keyclick and bell sounds. The ROM routines may be replaced by a TSR routine by merely setting the appropriate variable to the address of the appropriate routine in the TSR.

  `bell_hook` is a system variable at `$5ac`. The handler there gets a `jsr` when the bell is to sound. The enable bit in `conterm` has already been checked at that point - the bell really should sound. It may change d0-d2/a0-a2 but must preserve all other registers. It can take as long as it likes, and use BIOS calls. It can chain to the old value of the bell handler.

  `kcl_hook`, at `$5b0`, is a similar hook for keyclick. It can change the same regs, and it shouldn't take very long. If you want an elaborate sound, set something up to be played at interrupt level.

  Both handlers run in Supervisor mode. Each should return with `rts`.

Hard Disk Bootup Delay:

- A delay has been placed in the boot sequence at the time hard disk boot is attempted. This gives hard disks time to get up to speed before the OS starts trying to talk to them. The delay is ten seconds for an acknowledge to the first command byte, and forty seconds for data. This delay only occurs on powerup, not on warmboot.

  This is a convenience feature only, so you can power up your whole system at the same time, and may or may not work with all hard disk/controller/host adapter configurations. This feature does work with the Atari hard disks on which it was tested.

Physbase():

- Physbase() may return values that are not 256-byte aligned, since word-aligned values are now supported. Also see Setscreen().

Rsconf() Bug:

- The bug in Rsconf() in Rainbow TOS, where any attempt to set RTS/CTS flow control instead results in NO flow control, has been corrected in STE TOS.

Setscreen():

- Setscreen() supports word aligned arguments for physbase. Previously 256-byte aligned values were required. You can no longer expect Setscreen() to round down to a 256-byte boundary. Also see Physbase().

Sound:

- By default, DMA and GI sound are both mixed through the speaker and the external sound channels. There is no other BIOS support for DMA sound.

Timeouts:

- All BIOS timeouts now use the 200Hz timer rather than instruction loops. This means that if you disable the 200Hz timer, you may have problems with devices requiring timeouts, such as floppy I/O and DMA. Accesses to these devices will never time out; if no timeout errors occur, these devices will work fine.

The Cookie Jar

Starting after Rainbow TOS, TOS contains something we call the Cookie Jar. (It can be "retrofitted" into older TOSes, as well.) The name comes from the fact that certain, hopefully unique, numbers used to identify something special are called "cookies," and the Cookie Jar is a special place for storing them. Cookies are placed in the cookie jar by the system itself, and by programs which, in effect, add some utility to the system, such as resident device drivers or OS call handlers. Collectively, these programs are called TSRs, for Terminate and Stay Resident utilities. (See "Desk Accessories and Applications" for the special problems they present.)

The pointer to the Cookie Jar is found at `_p_cookies`, a new system variable at `$5a0`. If no Cookie Jar exists (e.g. on old TOS) this pointer will be zero.

Even in old TOSes, a program can install a cookie jar. See the section called "Cookie Jars and Old ROMs" below.

The Cookie Jar contains pairs of longwords. The first longword of each pair is the cookie; it's a (hopefully) unique number which identifies the m~~ning of this entry. The second longword is any value at all, and provides additional information relating to that cookie. Think of entries as environment variables, with a name and a value.

The first longword (the cookie) may also be any value, but when choosing values you should strive for something both descriptive and unique. Many good choices spell out meaningful words when dumped as four ASCII characters. **Please don't use any variant of the word "cookie."** It's just too obvious, it doesn't describe the entry at all, and besides, it's like calling a variable in your program "var."

Some cookies are boolean; that is, either the cookie is there or it isn't, meaning the associated TSR is either present or absent, and no other information is necessary.

The opposite end of the spectrum is an entry whose value is a pointer to a structure describing something in detail, such as variables which control the operation of the TSR or report its status.

In between are the entries which describe something useful in the second longword of the entry (such as the revision number of the TSR), but require no further information.

Any search of the cookie jar must end when it encounters an entry whose first longword is zero. This marks the end of the list of cookies. The second longword of that entry contains the size of the space which is allocated for the cookie jar, in entries (eight bytes each). Adding an entry means copying the zero entry and its value to the next slot, and placing your cookie and value where the zero entry was before. However, there might not be room for this: if the zero entry takes up the last allocated slot, you will have to reallocate the cookie jar.

Reallocating the Cookie Jar

If you need to add an entry to the jar, but the zero cookie is using the last slot in the jar, you have to allocate a new cookie jar. The new one must be at least as big as the old one, plus room to add your new cookie. It will pay, however, to allocate even more room than that, because if you allocate only one more slot and fill it, the next TSR will just have to reallocate the cookie jar again. Allocating the cookie jar in increments of eight slots (8*8=64 bytes) is probably reasonable.

To move the cookie jar, allocate the new space, copy the old jar to the new one, change `_p_cookies` to point to the new jar, and update the zero entry at the end with the new value for the size of the jar. Remember, the value of the zero entry is the size of the allocated space in entries, not bytes.

The old value of `_p_cookies` doesn't matter; there's no reason to save it, and it can't be freed or otherwise recovered.

Do not rely on the location of the cookie jar or a specific cookie in the jar -- some later program might reallocate the jar, or even reorder the entries.

The crucial aspect of reallocating the cookie jar is that the memory it gets put in must never be freed or reused. That's why the cookie jar can only be expanded by resident utilities and the system itself, not transient programs, which terminate, and not accessories, which are freed by a resolution change.

Desk Accessories and Applications

The cookie jar is not a generalized interprocess communication facility. Desk accessories and applications (such as shells) should not place cookies in the cookie jar. In the first place, they can't reallocate the cookie jar because the memory they own can be freed: desk accessories go away if the resolution is changed from the Desktop, and applications go away when they terminate. In the second place, there are other ways of communicating with accessories and applications: the AES has message-passing, and shells can place information in the environment string.

Using the Cookie Jar

TSRs place cookies in the jar so other programs can tell that they're installed, and get other information like a pointer to a structure containing status and control variables, or even procedure addresses.

The absence of a TSR's cookie from the cookie jar means that TSR hasn't installed itself (or has somehow been removed.) The absence of one of the cookies that the system puts in the cookie jar means you're on an ST with ROMs which don't initialize the cookie jar, and you should behave accordingly. You can presume that you are on a 520ST, 1040ST, or Mega ST using TOS 1.4 or earlier. TOS 1.6, in the STe, is the first version of TOS with a BIOS which installs a cookie jar.

Programs can scan the cookie jar for the presence of a cookie using a routine like `getcookie()`, found at the end of this document.

Cookie Jars and Old ROMs

Old ROMs clear the pointer at `$5a0` at cold boot, which is why a zero value there means there is no cookie jar. A TSR which knows about cookie jars can create the cookie jar, though, by allocating some space (in the area that will stay resident, of course), setting up the zero entry, and placing a pointer to that space at `$5a0`. Other TSRs which know about the cookie jar will see this one, and operate normally.

A program which **creates** a cookie jar (because the BIOS didn't do it) should also arrange to **remove** it if the machine is reset, because a warm boot with old ROMs does not zero the cookie jar pointer. This can be done with a procedure like `unjar()`, found at the end of this document. If your program expands an existing cookie jar it need not use `unjar()`; only the program which **creates** the cookie jar needs to remove it.

Cookies Installed by the BIOS

Cookies which the BIOS installs have an underscore (`$5f`, '_') in the high-order byte, to avoid conflict with other cookies. Inasmuch as we can specify what cookies people should use, please consider all names with an underscore in the first byte as "reserved for use by Atari."

As a rule, things which you can easily determine in existing, ROM-independent ways have not been assigned cookies, but things which are harder to find out have been. The system-installed cookies are:

Cookie	Value
_CPU	0, 10, 20, 30 (decimal), meaning 68000, 68010, etc
_VDO	the major/minor part number of the video shifter installed.
_SND	a bitmap of sound hardware types available.
_MCH	the machine type, describing the machine in general.
_SWI	the value of configuration switches, if available.
_FRB	a pointer to a 64K buffer for ACSI DMA transfers.

The value for _VDO is a major and minor part number for the video shifter in the machine. The low-order word of the value is the minor part number, to make fine distinctions, and for now they're all zero. The high-order word of the value is the major part number. Their characteristics are described elsewhere. As a rule, check only the high word to find out what kind of video hardware is in the machine. The low word will make finer distinctions should that become necessary, but all parts with the same major number will be compatible with one another (indeed, nearly identical).

_VDO COOKIE (high word)

Value	Meaning
0	ST
1	STE
2	TT

The value for _SND is a bitmap of sound hardware available. Some of the bits are for future expansion. External hardware and drivers (ST-Replay, etc.) should install their own cookies: the _SND cookie describes only the sound hardware which the BIOS knows about.

_SND COOKIE

Bit	Meaning
0	GI/Yamaha sound chip (as in STs)
1	stereo DMA sound (as in STEs and TTs)

The value for _MCH is simply an indication of what machine you're on. It describes the "rest" of the machine, such as the presence and type of real-time clock, DMA channels, etc. Any attribute described by a cookie takes precedence over the general attributes lumped into the overall machine description.

As with _VDO, the _MCH value is defined as two words, with a major and minor number for each machine type. The minor number is currently always zero, until we come up with something which needs it. If you want to know about the machine in general, check only the major number (the high word of the value).

_MCH COOKIE (high word)

Value	Meaning
0	520ST, 1040ST, Mega
1	STE
2	TT

The value of the _SWI cookie is the value of the configuration switches, if any. Only STE and TT have configuration switches. They are normally found at the same place, except that some TTs have them in a different place. That's one reason there's a cookie for them. Another reason is that the BIOS had to probe them anyway, to determine the presence of the sound output electronics for that bit in _SND. The meaning of each configuration switch is defined elsewhere.

The value of the _FRB ("Fast Ram Buffer") cookie, if present, is the address of a 64K buffer intended for use by ACSI DMA device drivers. Single-purpose RAM on the Atari TT (sometimes called "Fast RAM" because it is not shared with the video logic, and can be accessed in burst mode) is not accessible to the ACSI DMA controller. Therefore, requests for transfers to or from this RAM must use dual-purpose RAM as a staging area. This 64K buffer is provided for use by all ACSI DMA drivers, so they don't all have to allocate their own buffers. Naturally, this mechanism implies that drivers are monolithic: once a transfer is started, no other transfer can start, because both would try to use the same FRB. This is normally not a problem, since the ACSI DMA channel is also monolithic in this way. (Access is controlled with the `flock` system variable.) If the _FRB cookie is absent, it is because there is no single-purpose RAM.

Finding Cookies and Values

getcookie(): C Procedure to Find Cookies and Values

```
/*
 * int
 * getcookie(cookie, p_value)
 * long cookie;
 * long *p_value;
 *
 * Returns zero if the 'cookie' is not found in the cookie jar.
 * Returns nonzero if the 'cookie' is found, and places its value
 * in the longword pointed to by p_value.  If p_value is NULL,
 * doesn't put the value anywhere, but still returns the error code.
 *
 * Note that getcookie takes the cookie itself as its first argument,
 * so calls look like this:
 *
 *    long value;
 *    if (getcookie(0x5f56444f,&value)) { succeed; }
 *
 * The code assumes that Super() and NULL have been defined appropriately
 * compiler; since Super is so strange, you might get warnings about
 * type conversions, but this should still compile correctly.  The
 * cookie jar pointer is in user-mode-protected RAM, but the cookie
 * jar itself won't be.
 */

struct cookie {
     long c;
     long v;
};

int
getcookie(target,p_value)
long target;
long *p_value;
{
    long oldssp;
    struct cookie *cookie_ptr;

    /* get super mode if not already */
    if (Super(1L) == 0) oldssp = Super(0L);
    else oldssp = 0;

    cookie_ptr = *(struct cookie **)0x5a0;

    /* back to user mode if necessary */
    if (oldssp) Super(oldssp);
     if (cookie_ptr != NULL) {
      /* Use do/while here so we can match the zero entry itself */
      do {
          if (cookie_ptr->c == target) {

            /* found it */
            if (p_value != NULL) *p_value = cookie_ptr->v;

            /* return nonzero for success */
            return 1;
          }
      } while ((cookie_ptr++)->c != 0);
```

```
    }

    /* failed to find it (or no cookie jar at all!) */

    /* return 0 (failed) */
    return 0;
}
```

Clearing the Cookie Pointer on Warm Boot

```
_unjar: Procedure to clear cookie pointer on warm boot

* Procedure to install a new reset handler which clears _p_cookies
* on warm boots, for TOS ROMs which don't do that themselves.
* reshand and the save variables must be in the resident part of a TSR.
* This procedure must be executed in Supervisor mode.
* Reshand itself will run in Supervisor mode, too.

RESMAGIC    equ         $31415926
_resvalid   equ         $426
_resvector  equ         $42a
_p_cookies  equ         $5a0

            .globl      _unjar
_unjar:     move.l      _resvalid,valsave       ; save old valid
            move.l      _resvector,vecsave      ; ... and vector
            move.l      #reshand,_resvector     ; install new vector
            move.l      #RESMAGIC,_resvalid     ; ... and validate it
            rts

reshand:
            clr.l       _p_cookies              ; clobber cookie jar pointer
            move.l      vecsave,_resvector      ; restore old vector
            move.l      valsave,_resvalid       ; restore old valid
            jmp         (a6)                    ; return to ROMs

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