TT030 TOS Release Notes

October 8, 1990

Atari Corporation
1196 Borregas Avenue
Sunnyvale, CA 94086

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

Table of Contents

AES Changes

AES Version Number: The AES version number returned in globl[0] after appl_init is now 0x0300 (version 3.0).

Critical Error Handler: No context switching will happen in the AES critical error handler. Previously, this could have caused recursive GEMDOS calls if a timer event was waiting at the time a critical error happened. Now all processes are blocked until the user dismisses the critical error alert box. Remember that GEM programs which take the critical error handler must not make any AES or GEMDOS calls while handling a critical error. No GEMDOS calls because GEMDOS is not recursive at all, and it could have been a GEMDOS call which triggered the critical error. No AES calls because an application making an AES call will cause a context switch, and the AES process switched to might make a GEMDOS call.

Desk Accessories: On startup and after a change of resolution, the AES now creates a GEMDOS process for the code that starts up desk accessories and launches the desktop. As a result, *all* memory which is allocated by accessories and the Desktop will be freed on a resolution change. Previously, it was possible for a desk accessory to Malloc memory which did not get freed because the AES had no way of knowing about that memory. Now, the Desktop (as a GEMDOS process) actually terminates on a resolution change, and GEMDOS frees all the memory allocated to it.

Any appl_exit call will now block the calling process until all desk accessories are waiting for a message event. As a result, under this AES version, DAs can safely Malloc memory. Memory allocated by a DA is actually owned by the GEMDOS process that is running when the application makes the Malloc call. When that process terminates, GEMDOS will free the memory. Previously, the AES was not always getting AC_CLOSE messages to DAs until after a program had already terminated, and all of its resources freed. Now, a DA can be assured that it will receive the AC_CLOSE message *before* the application terminates, so any memory that it has allocated will not already be invalidated at AC_CLOSE time.

This improved AC_CLOSE handling helps with VDI workstations opened by desk accessories as well. When a workstation is opened, the VDI makes a GEMDOS Malloc call to allocate memory for an application's workstation. Previously, any workstation opened by a desk accessory would be freed when its parent application terminated, and possibly before the DA got its AC_CLOSE message and was able to close the workstation.

Menu/Alert Screen Buffer:

The menu/alert screen buffer is now 1/2 the size of the screen memory.

Mouse, Keyboard, and Screen Ownership:

Mouse, keyboard, and screen control ownership are now unconditionally changed to the previous process after a process is run by the shell library. This shouldn't affect any applications except the Desktop, but it does make error recovery from errant programs more reliable.

Window Colors

Two new wind_set calls are provided in the Atari GEM AES version 3.0 to set the colors of window elements. See the GEM AES manual section 11.3.6, page 11-21 for information on the wind_set call. The parameters for setting window element colors are as follows:

wind_set(WORD handle, WORD field, WORD element, WORD tcolor, WORD bcolor);

The "field" values for these new calls are 18 (WF_COLOR), to set window element colors by window handle, and 19 (WF_DCOLOR), to set default window element colors. When using WF_COLOR, handle is the AES handle of the window whose colors are to be set. When using WF_DCOLOR, the handle parameter is ignored.

Applications should normally *not* use WF_DCOLOR to set the default colors, because users will use the Window Colors control panel extension to set the default window colors they want to see. Instead, applications should use WF_COLOR to set window colors for their own windows.

The element parameter determines what part of the window (close box, name, horizontal slider, etc.) is set by the call. Window element numbers are defined in WCOLOR.H. These numbers should not be confused with the bits which define window parts for wind_create and wind_calc - they are different. Also remember that not all elements are visible on all windows. We could spend several pages describing what elements are visible under what circumstances, but the result would be more confusing than enlightening. The best way to find out what elements are visible in a given window is to experiment with different color settings.

The tcolor parameter defines the element color when the window is topped; bcolor defines the color when the window is in the background. A value of -1 for tcolor or bcolor means "don't change."

Because AES windows consist of a group of normal AES objects, what you actually set with these calls is the "object color WORD" for the window element you are setting. Thus, you set not only the text, border, and fill colors for the window element, but also the fill pattern and text mode (replace/transparent). See the GEM AES manual section 6.3.7.4, page 6-16 for more information on the object color WORD. WCOLOR.H provides C macros for converting color indices into a nybble-packed color word.

Applications which use wind_set to set window element colors should allow users to choose color sets. It is recommended that an interface similar to the Window Colors control panel extension be used for application based window color settings. As appropriate, provide controls that allow the user to select colors for elements or groups of elements which will be visible in the application's windows. Also provide reasonable default color sets which use the standard GEM palette. Keep in mind that a user may have set up a custom palette that renders your default color sets ugly or unusable.

WCOLOR.H

```
/* WCOLOR.H
 * definitions for new wind_set call WF_COLOR and WF_DCOLOR
 * 900614 kbad
 */

#define Color2Border( c )       ( (c) << 12 )
#define Color2Text( c )   ( (c) << 8 )
#define Color2Fill( c )    ( c )

#define ColorWord( borderColor, textColor, fillColor )\
        ( Color2Border(borderColor) |\
        Color2Text(textColor) |\
        Color2Fill(fillColor) )

#define WF_COLOR 18   /* set element color words by handle */
#define WF_DCOLOR 19 /* set default element color words  */

/*

New wind_set call for setting window element colors:

wind_set( WORD w_id, WORD field, WORD element, WORD tcolor, WORD bcolor);

        field: WF_COLOR set object color words for window element by handle
        field: WF_DCOLOR set default object color words for window element
                (w_id parameter is ignored for WF_DCOLOR)
       element: part of window to set, defined below
       tcolor: object color word used when window is topped (-1: ignore)
       bcolor: object color word used when window is not topped (-1: ignore)


*/

/* Window elements          object type     description             */
#define W_BOX       0   /*  IBOX            window parent object    */
#define W_TITLE     1   /*  BOX             parent of closer, name, fuller  */
#define W_CLOSER    2   /*  BOXCHAR         close box               */
#define W_NAME      3   /*  BOXTEXT         mover bar               */
#define W_FULLER    4   /*  BOXCHAR         full box                */
#define W_INFO      5   /*  BOXTEXT         info line               */
```

```
#define W_DATA      6  /*  IBOX     holds remaining window elements */
#define W_WORK      7  /*  IBOX     application work area            */
#define W_SIZER     8  /*  BOXCHAR  sizer box                       */
#define W_VBAR      9  /*  BOX      holds vertical slider elements  */
#define W_UPARROW   10 /*  BOXCHAR  vert. slider up arrow           */
#define W_DNARROW   11 /*  BOXCHAR  vert. slider down arrow         */
#define W_VSLIDE    12 /*  BOX      vert. slider background         */
#define W_VELEV     13 /*  BOX      vert. slider thumb/elevator     */
#define W_HBAR      14 /*  BOX      holds horizontal slider elements*/
#define W_LFARROW   15 /*  BOXCHAR  horz. slider left arrow         */
#define W_RTARROW   16 /*  BOXCHAR  horz. slider right arrow        */
#define W_HSLIDE    17 /*  BOX      horz. slider background         */
#define W_HELEV     18 /*  BOX      horz. slider thumb/elevator     */
```

_FPU Cookie:

The _FPU cookie describes what hardware and software floating-point support is installed in the machine. The low word of the cookie's value describes software floating-point support; currently, this is always zero, but Atari may assign values in the future for software floating-point support packages. Notably, the 68040 requires software to support some of the instructions that the 68881/68882 execute on-chip.

The high word describes the hardware floating-point installed in your system. Note that the "unsure" cases result because the BIOS probes for the 6888x without determining which FPU you have. If some software cares, it can probe, and reset the cookie's value accordingly. The BIOS *always* installs an _FPU cookie, even with zero value, so floating-point support software can change the low word when it installs itself.

_FPU Cookie (high word)

Value	Meaning
0	No hardware FPU detected
1	SFP004 or compatible: 68881 as peripheral
2	68881 or 68882, unsure which, as coprocessor
3	68881 or 68882 plus SFP004
4	68881 for sure
5	68881 plus SFP004
6	68882 for sure
7	68882 plus SFP004
8	68040's internal floating-point support
9	68040 plus SFP004

Note: If your software requires Line-F floating point support, check this cookie for a non-zero value in either the high or low word.

Malloc:

Malloc(0L) returns zero. It used to return a pointer, but since the call doesn't allocate any memory, the pointer pointed to memory that nobody owned, and that pointer could not legally be used. Now a Malloc request for zero bytes is considered an error, and Malloc returns zero.

Sversion:

Sversion returns the GEMDOS version number. The return value has the "major" revision number in the *low* byte and the "major" revision number in the ***high*** byte.

Version	Major	Minor	Returned by Sversion
Mega TOS 1.02	00	13	0x1300
Rainbow TOS 1.04	00	15	0x1500
STE TOS 1.06	00	15	0x1500
STE TOS 1.62	00	17	0x1700
TT TOS 3.01	00	19	0x1900

TOS Version Number:

The TOS version number for the first release of TT TOS is TOS 3.01.

Mxalloc 0x44 Allocate memory (with preference)

```
LONG    Mxalloc(amount, mode)
LONG    amount;
WORD    mode;
```

This call works like Malloc(), but takes an extra parameter: a flag telling where to get the memory.

Mode	Meaning
0	ST RAM only
1	alternative RAM only
2	either, ST RAM preferred
3	either, alternative RAM preferred

If amount is -1L, the size of the largest single block of the type specified by mode is returned. In that case, mode values 2 and 3 are identical, and the size of the largest block of either type is returned.

If amount is not -1L, a block amount bytes long is allocated to the calling process and a pointer to it is returned. If mode is 0 or 1, the block will come from the kind of memory specified. If mode is 2 or 3, the "preferred" type of memory is checked first for a large-enough block, then the other type is checked.

If alternate RAM is eligible to satisfy a request, but there isn't enough of it available, the request will come from ST RAM. If there isn't enough of *that,* the request fails.

It should be clear that the Malloc() call devolves into a Mxalloc() call with a mode value of 0 or 3, depending on the state of the Malloc-eligibility bit in the program's header.

Maddalt 0x14 Inform GEMDOS of "alternative" memory

LONG Maddalt(start,size)

This call causes GEMDOS to become aware of memory that it can use for loading processes and satisfying Malloc calls. It can be used to inform GEMDOS of memory that the BIOS memory-sizing algorithm did not tell GEMDOS about initially.

The arguments start and size must describe a contiguous block of memory, and once this call is made that memory "belongs" to GEMDOS. There is no way to "take it back," and no program should use this memory except through GEMDOS's Malloc and Pexec calls.

The memory so added is "alternative" memory; that is, it is only eligible for program loading if the Alternative RAM Load bit is set in the header of the program being loaded, and it is only eligible for satisfying Malloc calls if the Alternative RAM Malloc is set, or if Mxalloc modes 1 or 3 is used.

Maddalt returns 0 for success, or an error code for failure.

This call should only be made once for a given block of memory. The best way to do this is to run a program in your AUTO folder that makes the Maddalt call. This would only be appropriate if there is memory in your system that you want to use for programs and Malloc calls, and which the BIOS doesn't "find" and tell GEMDOS about at boot time.

New XBIOS Graphics Calls for the TT030

Warning: Use of these calls is restricted to the TT only. For compatibility with other Atari computers use of these calls should be limited to those applications which will require porting due to other incompatibility.

EgetPalette 0x85 Get Look Up Table registers

```
VOID    EgetPalette(colorNum, count, palettePtr)
WORD    colorNum, count;
LONG    palettePtr;
```

Copy the contents of of a contiguous set of TT hardware color Look Up Table (LUT) registers starting with register colorNum into the area pointed to by palettePtr. count words are transferred into the area. palettePtr must fall on a word boundary.

EgetShift 0x81 Get current shift mode value

```
WORD    EgetShift()
```

Return the current shift mode register value. See Esetshift for details.

EsetBank 0x82 Set color Look Up bank

```
WORD    EsetBank(bankNum)
WORD    bankNum;
```

Set bank number (0–15) for active TT color Look Up Table (LUT). This call also maps the current bank's colors to the old ST color Look Up Table. The bank is set immediately. Function always returns old bankNum. If bankNum is negative, the hardware register is not altered.

EsetColor 0x83 Set color entry

```
WORD    EsetColor(colorNum, color)
WORD    colorNum, color;
```

Set the absolute entry colorNum in the TT color Look Up Table (LUT) to the given color. color is set immediately. Always returns the old color. If color is negative, the hardware register is not altered.

EsetGray 0x86 Set/clear gray mode

```
WORD    EsetGray(switch)
WORD    switch;
```

Set the manner in which the color Look Up Table (LUT) data is interpreted by the display hardware. A switch value of zero directs the display hardware to interpret the LUT data as color, 4 bits for each of the three components. With a non-zero switch value, the upper byte of the LUT entry is ignored and the lower byte alone represents one of 256 gray levels. Always returns the old switch value (a non-zero value means switch is set). On input if switch is set to a negative value, the hardware register is not altered.

EsetPalette 0x84 Set palette registers

```
VOID    EsetPalette(colorNum, count, palettePtr)
WORD    colorNum, count;
LONG    palettePtr;
```

Set the contents of a contiguous set of TT hardware palette registers with the words pointed to by palettePtr. palettePtr must fall on a word boundary. The set of registers loaded begins with Look Up Table (LUT) register colorNum and extends for count words. The function sets the palette immediately.

EsetShift 0x80 Set shift mode register

```
WORD        EsetShift(shftMode)
WORD        shftMode
```

Set the TT shift mode register to shftMode. Return old shift mode register value.

Bit Assignments

```
S - - G - M M M - - - - B B B B
```

```
S = Smear Mode
G = Gray-Mode
MMM = Mode:  000    320x200x4
             001    640x200x2
             010    640x400x1
             100    640x480x4
             110    1280x960x1
             111    320x480x8
BBBB = Bank
```

For more information on Gray-Mode see: EsetGray. Note the values returned by Getrez() correspond to the Mode information given here. Also, it is easier to change individual elements of this register using the specialized calls below. They should be used whenever possible.

EsetSmear 0x87 Set/clear video smear mode

```
WORD    EsetSmear(switch)
WORD    switch;
```

Set the video smear mode. A switch value of zero indicates normal display mode while a non-zero value instructs the display hardware to repeat (smear) the last non-zero color encountered whenever zero values are retrieved. Function always returns the old switch value (a non-zero value means switch is set). On input, if switch is set to a negative value, the hardware register is not altered.

Bconmap 0x2c Change mapping of device 1

```
LONG    Bconmap(devno)
WORD    devno;
```

This call maps devno in as Bcon* device number 1. It returns the old mapping. If devno is -1, there's no change; the current mapping is simply returned. If devno is -2, a pointer is returned (see below). Legal values are -1 (for no change), -2 (to return the pointer), and values 6 and up.

You can tell you're on a system which doesn't support Bconmap by making the call and checking the return: if the return value is 44 (0x2c, the same as the XBIOS call number) then Bconmap is not available.

In addition to the above, if devno is -2, a pointer to the device mapping structure is returned. This is used by programs which need to install new mappable handlers. It also contains the number of mappable devices; the highest legal devno value for Bcon calls (including Bconmap itself) is that number plus 5.

Illegal values (0-5, or higher than the highest legal value, or negative but not -1 or -2) don't change anything, and return 0.

The mapping is accomplished by writing into the (published) vector table in low memory. In addition, new pointers are used to make Iorec and Rsconf indirect. Therefore, programs which use the vector table in low RAM and/or Iorec and Rsconf will see the "currently-mapped" device when they make Bcon calls with devno=1, and when they make Iorec and Rsconf calls.

Bconmap-aware programs can use the higher devno values to get at a specific port, no matter what the current mapping is. They still need to use Bconmap to "map in" the desired port before making lorec and Rsconf calls.

DMAread 0x2a Read sectors from device

```
LONG    DMAread(sector,count,buffer,devno)
LONG    sector;
WORD    count;
VOID    *buffer;
WORD    devno;
```

Reads sectors from the device into memory. Works for ACSI and SCSI devices. For SCSI, does not actually use DMA: handshakes the bytes across.

Device numbers are:

devno	Device
$0-$7	ACSI devices $0-$7
$8-$f	SCSI devices $8-$f
other	reserved for future use

Returns a BIOS error code. This call assumes the memory at buffer can actually be accessed by the bus the device is on. Therefore, DMAread from an ACSI device into alternative RAM won't work.

DMAwrite 0x2b Write sectors to device

```
LONG    DMAwrite(sector,count,buffer,devno)
LONG    sector;
WORD    count;
VOID    *buffer;
WORD    devno;
```

Writes sectors from memory to a device. Works for ACSI and SCSI devices. For SCSI, does not actually use DMA: handshakes the bytes across.

Device numbers are:

devno	Device
$0-$7	ACSI devices $0-$7
$8-$f	SCSI devices $8-$f
other	reserved for future use

Returns a BIOS error code. This call assumes the memory at **buffer** can actually be accessed by the bus the device is on.

NVMaccess 0x2e Access Non-Volatile Memory

```
WORD    NVMaccess(op,start,count,buffer)
WORD    op, start, count;
BYTE    *buffer;
```

This call manages the non-volatile memory (NVM) in the TT's real-time clock chip. There are 50 bytes of memory there, of which two are reserved at the end as a check on the rest of the data. This call validates the check data on reads, recomputes the check data on writes, and initializes the check data if you want.

Opcode	Meaning
0	READ: copy data from NVM to the buffer.
1	WRITE: copy data from the buffer to NVM.
2	INIT: zero the NVM and initialize the check data.

start specifies the first location to read or write; count specifies the number of bytes to transfer.

Returns zero for success, EBADRQ (-5) for a range error in the arguments, and EGENRL (-12) if the NVM check data isn't consistent before a read or write. In the case of a read the data is transferred anyway.

NVM usage is to be dictated by Atari. We will take suggestions and applications for assignments of bytes, but using bytes or values whose meanings are not published by us assures trouble in the future.

Bconmap Discussion

Bconmap() makes the new serial ports on a TT030 accessible to programs which were written when there was just one serial port.

There are new Bconin/out/stat/ostat device numbers on a TT:

devno	Meaning
0	PRN
1	currently-mapped serial port (see below)
2	CON
3	MIDI
4	IKBD
5	RAW
6	ST-compatible serial port (called Modem 1; default).
7	SCC Channel B (Modem 2 on the back of a TT).
8	TTMFP serial port (3-wire, Serial 1).
9	SCC Channel A (full handshake, Serial 2).

Bcon calls on device 1 (normally AUX) might actually refer to any of these devices, or to a user-installed device (with an even higher devno). You use Bconmap to change the mapping of device 1. Bconmap also changes the mapping of Rsconf() calls, and of lorec calls with lorec device number 0.

The port assignments above are for TT only; other machines with "extra" serial ports will have other assignments. Port 6 is always going to be the ST-compatible one, though. Other devices may be installed by drivers at boot time (or even later).

Installing New Bconmap Device Drivers

Bconmap(-2) returns a pointer to a structure. The structure looks like this:

```
struct bconmap {
        LONG *maptab;          /* ptr to map table (see below)   */
        WORD maptabsize;       /* number of lines in the table   */
};
```

The map table contains a line for each device. Each line contains pointers to the Bconstat, Bconin, Bcostat, and Bconout routines, plus a pointer to the Rsconf routine, plus a pointer to the Iorec for that device. The table's size (the number of devices) is in maptabsize. Maptabsize is used by all Bcon calls to range-check the device number. The highest legal value for Bcon calls (including Bconmap) is this number plus 5.

A Bconmappable driver must have Bconstat, Bconin, Bcostat, and Bconout entry points, plus an iorec, plus an Rsconf function pointer. You install it by copying the table pointed to by maptab into a larger area and adding your driver's entry (five procedure pointers and an iorec pointer), then changing maptab and maptabsize.

In the unlikely event that your program finds itself installing the 38th device, that is, the one which would have Bconmap number 44 (decimal), you should actually allocate TWO new rows for maptab, install your device in the SECOND one, and increment maptabsize by two. Otherwise, a current mapping of device 44 would be indistinguishable from the case where Bconmap was not available at all. No programs should be told to use device number 44.

Bconmap and Rsconf

Rsconf has been mis-documented for some time. It actually returns a longword value. That longword is four bytes stuck together. Those four bytes are the UCR, RSR, and TSR registers of the MFP, plus a useless byte. The UCR register is in the high byte of the returned longword, followed by the RSR, then the TSR, and the useless byte in the low-order byte. These bytes are the values of those registers BEFORE the changes dictated by the arguments to Rsconf.

In addition, ever since Rainbow TOS, Rsconf(-2,-1,-1,-1,-1,-1) returns the last baud-rate value set with Rsconf. If the first argument is -2, all the other arguments are ignored.

In the world of Bconmap, the Rsconf arguments have to be interpreted slightly differently. Not every device is a 68901 MFP any more. For the new devices, the bits which make sense are used, the others discarded.

Programmers should use Rsconf(-1,-1,-1,-1,-1,-1) to read the current values, then change the bits they want to change and call Rsconf again with the new values; changing bits in registers not listed below is now considered illegal. (Consider a REAL MFP: TSR contains, among other things, the transmitter enable bit; if you write 0x08 to cause a break, you will be disabling the transmitter!)

The bits in the Rsconf args and return value which Bconmappable devices must emulate are as follows:

UCR: bits 6-5: word-length (00=8, 01=7, 10=6, 11=5)
bits 4-3: stop bits: (01=1, 10=1.5, 11=2; 00 is invalid)
bit 2: parity (0=no, 1=yes)
bit 1: parity (0=odd, 1=even, meaningful only if bit 2 is 1)

RSR: none

TSR: bit 3: break (sends break while 1)

SCR: none

Programs which use synchronous modes probably talk directly to their hardware, so it doesn't make much sense to "emulate" that here. If a legal value is inconvenient (such as 1.5 stop bits with hardware which doesn't support it, or a baud rate you can't support) you can ignore it: users will get used to the restrictions imposed by your device and driver.

Serial 2 vs. LAN Connector

SCC channel A is shared between the DB9 on the back of the TT030 (labelled Serial 2) and the round LAN connector on the left side. Initially, it is programmed to use the DB9 connector. The selection can be made as follows:

```
Select LAN connector:   Offgibit(0x7f);  /* clear bit 7 (only)   */
Select DB9 connector:   Ongibit(0x80);   /* set bit 7 (only)     */
```

The Two Kinds of RAM

This section discusses the concept of "alternative RAM" in general first, and gets to the specifics as they relate to the TT030 later.

In the TT030 and other ST-like machines planned for the future, there are two general kinds of RAM: there is ST RAM, which is ST-compatible, and there is "alternative RAM," which is not. Exactly how it is not varies by machine and type of RAM. Primarily, the video chip can only display screen data from ST RAM, and the DMA sound chip can only play data stored in ST RAM. Secondarily, other chips which access memory, like ACSI DMA (for ST hard disks and other devices) and SCSI DMA (for SCSI devices), may not be able to get at alternative memory directly. This affects most programs not at all, since they use BIOS and GEMDOS calls to accomplish this kind of transfer, and the device driver is responsible for getting the data from here to there transparently, no matter where "here" and "there" are.

The "rules for eligibility" for a program running in alternate RAM are:

(1) It must not try to set the screen base address in alternative RAM, or play DMA sound from there.

(2) It must not try to make a device driver do DMA from or to there, unless the device driver knows about the differences between ST RAM and alternative RAM.

(3) It must not try to do DMA itself from or to there (only specialized device drivers do this).

The second point is a bit sticky: it refers to the fact that existing DMA device drivers don't know about the restrictions on alternative RAM.

Since programs written before there was any concept of alternative RAM don't know if they break the rules or not, you, the user, must inform GEMDOS as to whether a program is eligible to use alternative RAM, or must use ST RAM. As a finer distinction, you can select the eligibility for program loading and Malloc() calls separately. A program which Mallocs a screen buffer might still be eligible to load into alternative RAM, but its Malloc() calls must be satisfied from ST RAM.

The Specifics of the Two Kinds of RAM

As of Rainbow TOS , one of the reserved longwords in the header of executable files (PRG, TTP, TOS) acquired a meaning: the bits there control the way GEMDOS treats that program. (The least-significant bit of that longword (bit 0), when set, means GEMDOS need not clear all of RAM when loading that program, only the program's declared BSS. This makes programs load faster.)

The next two bits have been assigned meanings relating to alternative RAM. Bit 1, when clear, means that the program must be loaded into ST RAM; bit 2, when clear, means that Malloc calls by that program must be satisfied using ST RAM.

When one of these bits is set, the corresponding operation (program load, Malloc call) may be satisfied from "alternative" RAM. In general, alternative is considered preferable to ST RAM. If a program doesn't break any of the rules for eligibility in alternative RAM, it is desirable to set those bits in its header.

Bit	Meaning
0	Only clear BSS
1	Alternative RAM Load
2	Alternative RAM Malloc

If TT RAM is eligible to satisfy a request, but there isn't enough of it available, the request will come from ST RAM. If there isn't enough of *that,* the request fails.

For loading programs, "enough" RAM is a relative thing. For one program, it's more important to run fast than it is to have a lot of memory, so "enough" RAM is, say, 256K more than its own declared requirements (text+data+bss). For another, having lots of RAM is more important, even if it means not running as fast as possible.

A new field in the program's header, called the TPAsize field, reflects the memory requirements of the program. If the program's "program-load" bit is clear (meaning it must load in ST RAM) this field is ignored, and the program is loaded into ST RAM. If the program can be loaded in alternative RAM, and there's more alternative RAM than ST RAM, the field is ignored and the program is loaded into alternative RAM. The field is only checked if alternative RAM is eligible for loading the program, and there is more ST RAM available. In that case, the TPAsize field tells how much alternative RAM is "enough." If there is "enough" alternative RAM, the program loads there; if not, the program loads in ST RAM.

The TPAsize field tells, in 128K steps, how much alternative RAM is enough. The amount is added to the declared size (text+data+bss) of the program. If there is less than this amount available, the program gets loaded into ST RAM. The field is four bits wide, and is in the high four bits of the program-flags longword. The amount is the field's value times 128K, plus 128K. Therefore a value of zero, which is what all programs have currently, means 128K. The value can go up to 15, meaning 2MB.

Example

Setup: A program's alternative-RAM load bit is set. Its TPAsize field is set for 512K. Its text+data+bss size is 110K.

Result: If there is more alternative RAM than ST RAM, the program loads into alternative RAM.

If there is more ST RAM, the TPAsize field is taken into account. If there is 622K of alternative RAM available, or more, the program loads into alternative RAM. If not, it loads into ST RAM.

In this example, it's possible that there isn't 622K of ST RAM available either. If there is more than 110K, though, the program will still be loaded and run; the TPAsize field is not considered an absolute minimum for the program to load. 110K is the program's declared text+data+bss size: that, plus space for a small initial stack, is the absolute minimum.

Remember, TPAsize does not reflect the maximum or minimum size of the TPA the program will ultimately get. It's just the tiebreaker in the case where there is more ST RAM than alternative RAM, and GEMDOS has to know where to put the program.

What Does It All Mean?

There are two common memory models for programs on the Atari ST. One has the user or library declare a "stack size" at compile time or link time. The runtime startup moves the stack pointer to the end of the program plus the stack size and uses Mshrink to give the rest of the TPA back to GEMDOS. Then, as the program calls the library malloc(), it uses Malloc to get memory back from the OS. For this kind of program, the TPAsize field should represent at least as much space as the "stack size" the startup will use. MWC, GCC, Turbo C, and lots of other environments use this memory model.

The other memory model keeps some amount of memory, and that memory is used as a "heap" - the stack grows down from the top of it, and library malloc() calls use memory up from the bottom. For this kind of program, the TPAsize field should be the minimum reasonable size of the stack+heap space. Alcyon C uses this memory model.

You may wonder why these fields are part of the program header, and not controlled by, say, new GEMDOS calls, or new parameters to Pexec. The reason is that they are properly part of the program: a program's alternative-RAM characteristics and memory requirements are inherent in its behavior. They're not based on its parent's behavior, and its parent should not have to know about them in order for GEMDOS to make intelligent decisions.

Since the information is in the program's header, it can be changed by an outside utiliity without special knowledge of the program's structure. If you can see that a program doesn't do screen-flipping or talk to the DMA chip directly, it can probably be run in alternative RAM, and you could set its flags appropriately.

After Your Program Loads

The bit in the program header which controls the eligibility of alternative RAM for Malloc calls is intended for compatibility, so existing programs which have no knowledge of alternative RAM can get the benefit of the higher speed and extra capacity.

New programs, written after the publication of this information, can use a new call, Mxalloc(). This call works like Malloc(), but takes an extra parameter: a flag telling where to get the memory.

Mode	Meaning
0	ST RAM only
1	alternative RAM only
2	either, ST RAM preferred
3	either, alternative RAM preferred

If amount is -1L, the size of the largest single block of the type specified by mode is returned. In that case, mode values 2 and 3 are identical, and the size of the largest block of either type is returned.

If amount is not -1L, and a block of that size is available in the type(s) of memory specified by mode, the block is allocated and its starting address is returned.

It should be clear that the Malloc() call devolves into a Mxalloc() call with a mode value of 0 or 3, depending on the state of the Malloc-eligibility bit in the program's header.

Other Important Notes

The Line-A graphics interface is maintained for backward compatibility with existing ST programs only. It should not be used for new programs. It will not keep pace with future hardware or software improvements. The VDI should be used.