Pexec Cookbook

Third Edition
6 September 1991

Atari Corporation
1196 Borregas Avenue
Sunnyvale, CA 94086

COPYRIGHT

Copyright 1991 by Atari Corporation; all rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of Atari Corporation, 1196 Borregas Ave., Sunnyvale, CA 94086.

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. SLM, ST, and TOS are trademarks of Atari Corporation.

This document was produced entirely with Atari Computers, Atari SLM Laser Printers, and Microsoft Write.

Table of Contents

INTRODUCTION

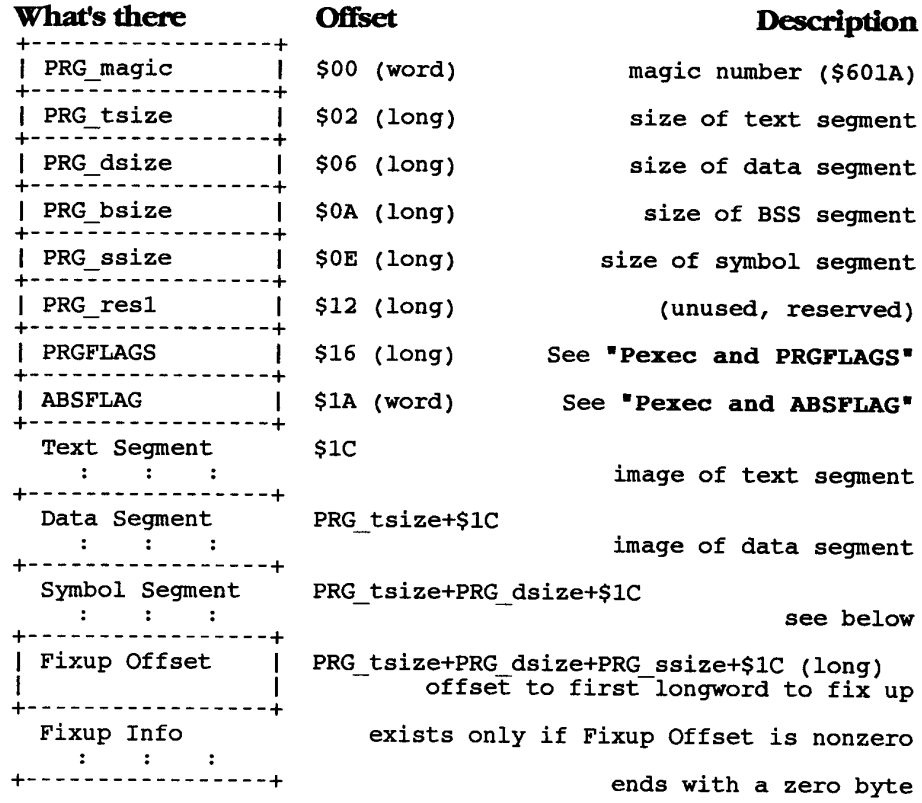
In broad terms, the things you have to know about **Pexec** are that it starts up a process, lets it execute, then returns to the caller when that process terminates. The *caller* - the process which used **Pexec** in the first place - has some responsibilities: it must make memory available to the OS for allocation to the child, it must build an argument string for the child, and it must either pass an environment to the child or let the child inherit a copy of its environment.

Pexec has several modes, not all of which are available in all versions of TOS. See **"Version? What Version?"** and **"Pexec Flavors"** for more information.

Note: Because **Desk Accessories** are not GEMDOS processes, they should not use **Pexec**. Although their files have the same internal structure as executable files, they are not treated as executable files, and the rest of this document does not apply to them.

THE FORMAT OF AN EXECUTABLE FILE

An executable file consists of a header followed by images of the text and data segments, zero or more symbol segment entries, a fixup offset, and zero or more fixup records:



Note: The Fixup Offset will not exist if ABSFLAG is nonzero, although ABSFLAG was not reliable in versions before Rainbow TOS. See **"Pexec and ABSFLAG."**

SYMBOLISM

The symbol segment is free-format and its contents depend on the tools used to generate the executable file. The Alcyon tools deposit Alcyon format symbols. Mark Williams 2.x tools generate an empty symbol segment (`PRG_ssize == 0`) and append their own symbol segment to the end of the fixup information, and Mark Williams 3.x puts the ID header and debug info in the symbol segment as well. For more information on Alcyon format symbols, consult the GEMDOS manual section on executable files.

FIXUP INFORMATION

Executable files may be loaded into memory on any word boundary. GEMDOS will fix up longwords in the text or data segments by adding the base of the text segment to the values already in the longwords. The fixup information specifies which longwords need to be fixed up before the program can be executed.

The fixup information starts with a longword containing the byte offset to the first location to fix up. If the offset is 0 then there are no fixups. See **"Pexec and ABSFLAG."**

Following the initial longword offset are a series of *relocation bytes*. These bytes specify offsets to further fixups. The bytes may have the following values:

Value	**Meaning**
0	End of relocation list;
1	Advance 254 bytes;
2..254 (even)	Advance 'N' bytes and fixup the longword there.

We assume a location pointer that is initially set to the longword fixup offset:

A byte of 0 indicates the end of the relocation list. A byte of 1 means to advance the location pointer by 254 bytes and examine the next relocation byte. A byte of 2 or more means to add the byte to the location pointer and add the address of the text segment to the longword at the address the location pointer points to. This may sound confusing, but it's really quite simple and a program to process these relocation bytes is easy to write.

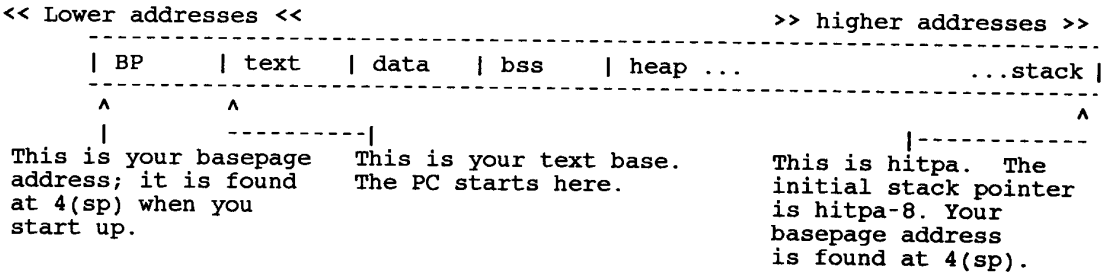
PEXEC AND ABSFLAG

The word at offset $1A in the program header is ABSFLAG; when nonzero there are no fixups.

Using a nonzero ABSFLAG to indicate that a PRG file has no fixups is not recommended, since old versions of TOS (TOS 1.0 and Mega TOS) did not handle this case correctly. A better way to represent a PRG file with no fixups is to leave ABSFLAG in the header as zero, and place a zero longword as the "offset to first fixup" at the start of the fixup segment. A zero value there means there are no fixups. This works on all TOS versions.

THE IMPORTANCE OF SHRINKING BEFORE PEXECING

Here is a picture of a process at the moment it is started by **Pexec** mode 0:



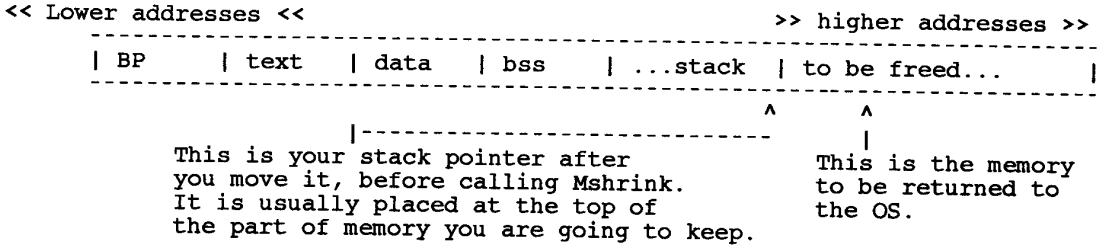
All programs are started with the largest available block of OS memory allocated to them. This block of memory in which a program runs is known as the Transient Program Area (TPA). Often (but not always!), a program's TPA is the block stretching from the end of the accessories and resident utilities to the beginning of screen memory. The point is that your program may have been allocated *all* of free memory.

The "heap" is the area between the end of your declared bss and your initial stack pointer. Your stack grows downward in this area. In the Alcyon C memory model, **malloc()** calls are satisfied from the heap. For most other libraries, you declare a stack size, and the C runtime startup moves your stack to (**bssend + stksiz**) and does an **Mshrink** down to that size, as illustrated below.

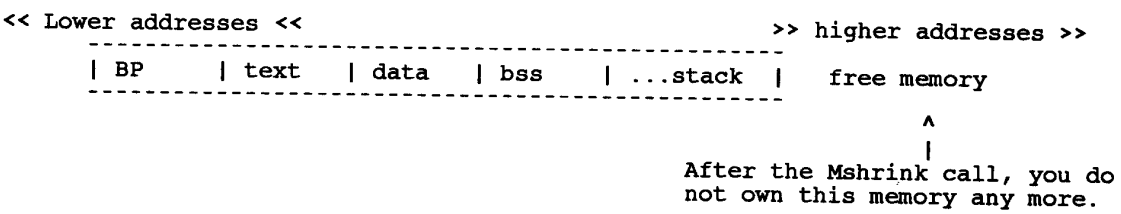
In order to make memory available for a child process, your program must *shrink* the block it owns, returning the top part to GEMDOS. The time to do this is when it starts up, or at least before using **Pexec**, and the call to use is **Mshrink**. Your program must also move its stack pointer before calling **Mshrink**, so its stack is never located outside its TPA. If it doesn't do this, its stack will be in memory it doesn't own, which is wrong.

All programs should call **Mshrink** to release some memory to TOS, even if they aren't going to **Pexec** anything. GEM VDI uses **Malloc** for its functions, AES **Mallocs** memory for file selector directories, etc. These OS functions will fail (and multi-tasking environment users will be angered) if your program selfishly keeps all of the system's memory for itself. Note that a program only needs to **Mshrink** its TPA once. After that, it can do as much **Pexec**ing as it wants.

Here is the picture just before calling **Mshrink:**



And here is the picture after calling **Mshrink:**



PEXEC AND MALLOC

It is not correct to assume much about how **Malloc** works. The only things guaranteed are (a) **Malloc(-1L)** returns the length of the largest block you can allocate, and (b) **Malloc** returns 0 if it can't allocate a block of the size you ask. Furthermore, **Pexec** is guaranteed to load you into the largest free block. But don't try to play with these guarantees too much: it is not the case that you can say **Malloc(-1L)** and know how big the TPA of a child is. For one thing, the child's environment might be allocated out of that same block of memory, but then it might not. Also, future **Mallocs** might have some overhead between blocks where linkage is kept. You just can't tell. Furthermore, it is not safe to assume that **Malloc(Malloc(-1L))** will succeed, because of the possibility of multi-tasking. In a multi-tasking environment it is also not *polite* to tyrannize all of memory like this.

PEXEC AND PRGFLAGS

Starting with the GEMDOS in Rainbow TOS (GEMDOS version 0.15), one of the reserved longwords in the header of executable files (PRG, TTP, TOS) has acquired a meaning; the bits there control the way GEMDOS treats that program. Briefly, the lower three bits of that longword have the following meanings:

Bit	Meaning
0	If set, clear only the program's declared BSS. If clear, the entire heap is cleared.
1	If set, the program is eligible to be loaded into alternative RAM. If clear, the program will only be loaded into ST RAM.
2	If set, the program's **`Malloc`** calls may be satisfied from alternative RAM. If clear, the program's **`Malloc`** calls will only be satisfied from ST RAM.

In addition, the high four bits of the PRGFLAGS value are used in deciding where to load the program.

These bits affect **`Pexec`** when using modes 0, 3, or 7. With mode 0 or 3, these values are retrieved from the longword at offset $16 in the executable file header. With mode 7, this longword is taken from the (otherwise zero) second input argument to **`Pexec`**.

For more detailed information on these "PRGFLAGS" bits, see the entry for "Program Startup" in the "GEMDOS Changes - Processes and Memory" section of the **Rainbow TOS Release Notes,** and the section titled, "The Specifics of the Two Kinds of RAM" section of the **TT030 TOS Release Notes.**

OVERVIEW OF PEXEC FLAVORS

Pexec Mode 0 (load and go)
These are the things that happen for Mode 0 (load & go). The tests for error conditions and cleanup are left out for clarity.

1. Make sure the file exists.
2. Set up env, TPA, BasePage:
    a. Allocate and copy the environment (call it env).
    b. Allocate the largest free block (call it BP).
    c. Set lowtpa, hitpa, dta, and env pointers in basepage.
    d. Do file handle and default directory inheritance.
    e. Copy command-line into BP.
3. Change the ownership of env & BP to BP.
4. Load executable file, fill in segment starts/sizes in BP.
5. Launch:
    a. Set parent pointer in BP.
    b. Set initial user stack, push initial registers to it.
    c. Set the "current process" global to BP.
    d. Start 'er up.

Pexec Mode 3 (load, don't go)
This does #1, #2, and #4. Because #3 is not done, the process that called **`Pexec`** still owns the memory.

Pexec Mode 4 (just go, original form)
This does #5 (launch) only. Because #3 is not done, the process that called **`Pexec`** still owns the memory. Thus, when the child process terminates, its env and TPA are not freed. Memory it got via **`Malloc`** *is* freed, however.

Pexec Mode 5 (create basepage)
This does #2 above (and not #3).

Pexec Mode 6 (just go, new form)
This does #3 above, then #5. Unlike mode 4, memory ownership is changed to the child process.

Pexec Mode 7 (create basepage, respecting PRGFLAGS)
This does #2 above (and not #3). Unlike mode 5, accepts PRGFLAGS as its second argument.

VERSION? WHAT VERSION?

Pexec modes 6 and 7 are not available in all GEMDOS versions. Mode 6 is not available prior to the GEMDOS version in Rainbow TOS, which was 0.15 (that is, **Sversion** returns 0x1500). Mode 7 was not available prior to the release of TT TOS, with GEMDOS version 0.19 (**Sversion** returns 0x1900). Before using these **Pexec** modes, you need to check what version of GEMDOS you are using. The only legal way to do this is to make the **Sversion** call. (You can't just check the TOS version number in the BIOS: it is possible to have one version of the BIOS and another version of GEMDOS loaded off disk.) The GEMDOS **Sversion** call returns GEMDOS's version number in a WORD, where the *high* byte of the word is the *minor* revision number, and the *low* byte is the *major* version number.

The following code fragments will tell you if **Pexec** mode 6 or 7 is available.

```
WORD pexec6_probe()
{
      WORD vers;

      vers = Sversion();

      if (((vers & 0xff) > 0) || ((vers & 0xff00) >= 0x1500))
            return YES;
      else return NO;
}


WORD pexec7_probe()
{
      WORD vers;

      vers = Sversion();

      if (((vers & 0xff) > 0) || ((vers & 0xff00) >= 0x1900))
            return YES;
      else return NO;
}
```

Note: Using a version number to determine what **Pexec** modes (or other GEMDOS features) are available is acceptable programming practice. Using a version number to determine the location of undocumented system variables is *not* acceptable. *You have been warned.*

MAKING THE CALL

When you want to exec a child, you build its filespec into one string, and its arguments into another. The argument string is a little strange: the first character of the argument string is the length of the rest of the string. This simple system call will load a program, pass in the specified argument string, and execute the program, returning the program's exit code to the caller:

```
long start_a_prog( cmd, args )
char *cmd, *args;
{
   char  buf[128];
   int   arglen;

   arglen = strlen( args );
   if ( arglen > 125 ) {
        printf( "argument string too long\n" );
        return -1;
   }
   strcpy( buf+1, args ); /* copy args to second byte */
   buf[0] = arglen;       /* set first byte to length */
   return Pexec( 0, cmd, buf, 0L );
}
```

The first zero in the **Pexec** call is the **Pexec** function code *load and go*. The **cmd** argument is the child's filespec: path (if needed), and file name with extender, null terminated. **Buf** becomes the child's command line, and the fourth argument is the environment pointer. A NULL environment pointer means "let the child inherit a copy of my environment."

RETURN FROM PEXEC

GEMDOS allows programs to return a 16-bit exit code to their parents when they terminate. This is done with the **Pterm(errcode)** call. The value in **errcode** is passed to the parent as the return value of the **Pexec** system call. The C library function **exit(errcode)** usually uses this call, as does returning a value from **main()**. If the **Pexec** fails (not enough memory, file not found, etc.) a negative code is returned, and you should deal with it accordingly. Note that error returns from **Pexec** are always negative *longs*, while return codes from the child will have zeros in the upper 16 bits.

Unfortunately, the people who wrote the startup file for the Alcyon C compiler didn't use this. The compiler calls **exit** with an error code, which calls **_exit**, but the Alcyon **_exit** always uses **Pterm0**, which returns zero as the exit code. We fixed this by rewriting GEMSTART.S, the C runtime startup code used with Alcyon C. Now, new programs return correct exit codes, but the compiler still doesn't. It is possible, though, to patch the binaries of all passes of the compiler so they do.

WHY ((3 + 6) == 0) AND ((5 + 4) != 0)

Pexec modes 3, 5, and 7 are used to set up processes to be run later, 4 and 6 to run them once they are set up. If you use mode 6 to execute a child after setting it up with mode 3, the net result is the the same as using the *load and go* mode, because the child will go away when it terminates. Mode 6 is mostly useful if you want to fiddle a bit with the child before you execute it (for example, to set breakpoints if you are a debugger). Mode 4 is useful for specially designed programs which can be run more than once. As long as a program makes no assumptions about the memory it lives in, it is a fair candidate for mode 4 execution. Mode 5 allows you to put anything you want into the child's TPA before executing it, and could be used, for example, by an executive program which builds executable programs on the fly out of common object code libraries.

USAGE EXAMPLE OF PEXEC MODE 3 AND MODE 6

The function below uses **Pexec** mode 3 to load a process without starting it. Then it uses **Pexec** mode 6 to start it. It returns the error code from **Pexec**: long negative if the initial load fails, or long positive if it succeeded and the child process returned an error code, or zero.

```
#include <osbind.h>
long doexec()
{
      long bp;
      long errcode;

      bp = Pexec(3,"MYPRG.PRG","\003arg",0L);
      if (bp < 0) {
            /* Pexec failed; return the error code.   */
            return bp;
      }
      else {
            /* if you want to do something between          */
            /* loading the process and starting the   */
            /* process, do it here.  Then...                */

            errcode = Pexec(6,0L,bp,0L);

            /* Now the process has terminated. Like   */
            /* a Pexec mode 0, the process is GONE:   */
            /* it owns no memory or other resources   */

            return errcode;
      }
}
```

LOAD ONCE, EXECUTE MANY

One thing people may want is for "load, don't go" (**Pexec** mode 3) followed by "just go" (**Pexec** mode 4) to act just like "load and go." Mode 6 instead of mode 4 does exactly this. When the child terminates, its memory image will go away (because it is tagged with the child's PID), and all will be just like load and go. You can't restart such a process (it's gone as soon as it terminates).

But let's say you use mode 3 to load a process, then mode 4 to execute it. When that process terminates, its TPA is not freed, because its parent still owns it. Now you're stuck with a process in memory which didn't get freed. Any memory the process got via **Malloc** or its own **Pexec** 3, 5, or 7 calls *is* freed, however.

You might want the process in memory which didn't get freed. Then you could start up the process as often as you wanted with **Pexec** mode 4. This probably doesn't work the way you want. There are several restrictions:

1. The child must not **Mshrink** its TPA, or must be very careful about it.
2. If the child changes the contents of its text and data segments during the first run, they'll stay in the changed state for the second - probably not what is intended.
3. The child's BSS doesn't get cleared the second time.
4. There are handle- and current-directory-inheritence problems.
5. Possibly others.

Number 1 deserves some explanation. **Pexec** step #5b (see **"Overview of Pexec Flavors"**) sets the initial user stack pointer to **BP->hitpa**, then pushes your initial stack frame (zero, then your basepage address) to it. If the child did an **Mshrink** the first time it was executed, then nobody owns this memory any more! Running **Pexec** mode 4 again violates the commandment, "Thou shalt not mess with memory thou ownest not." The parent can't fix this by writing a new value into **BP->hitpa** because it doesn't know how small the child shrunk to. The *child* can fix this by writing the new **hitpa** value into its own basepage.

There is another way to load once and execute many times. It only gets around problem 1; problems 2, 3, and 4 and possibly others exist in full force.

The other way is to use mode 3 to load the process, then **Mshrink** its TPA to just the size of its bp+text+data+bss. Each time you want to start this process, you use mode 5 to create a basepage, fill that basepage with the text/data/bss start/len fields of the old one, and use mode 6 to execute this basepage. (You can use mode 4, too, if you **Mfree** the environment and TPA afterwards.)

The virtue of this approach is that you create a new basepage for each execution of the process, so the same basepage doesn't seem to get created once, then terminate many times. Also, since this process' TPA is full-size, the **hitpa** and **lowtpa** fields are accurate.

The drawback is that the process' basepage isn't contiguous with its text, data, and BSS segments, and they in turn aren't contiguous with the TPA memory the process owns. A program's startup would probably have to be specially written to understand this before it could be used, because this is so different from the normal state of affairs under TOS.

Lastly, some people want to load many processes using mode 3, and start them up one at a time. To do this, you have to use **Pexec** mode 3 to load the process, but then you have to **Mshrink** the process's TPA so you can load the next one. If you do this, you should change the pointer in **BP->hitpa**, because that's where the initial user stack pointer gets set when the process is started (with mode 4 or 6). If you use mode 4, and you want to start the process again later, be sure the mode 4 restrictions above are followed. Even then, there may be other difficulties.

The best solution of all is to use mode 5 or 7 to create a TPA, fill it from whatever source you like (a utility-program image in RAM, for example) relocate it (see **self.s**) and start it with mode 6. Totally legal, guaranteed to work.

INHERITANCE AND ITS RESPONSIBILITIES

Children created with **Pexec** inherit several things from their parents: their current drive, their current directory on each drive, and the meanings of file handles zero through five. Each of these things refers to internal data structures, and those data structures include a "use count" which tells how many processes are "using" that structure.

In GEMDOS versions up to and including version 0.19, this inheritance is done at the time the process is created. That means there's a problem when you play games like using mode 3 to load a process, then mode 4 multiple times to run it. The use counts are incremented during the **Pexec** mode 3, then decremented when the process terminates. If you call **Pexec** mode 4 again, the use counts are decremented *again* when the process terminates, and GEMDOS's internal data structures are now inconsistent with reality: something might be marked as free (use count is zero) when in fact it is in use.

It is slightly better if you use mode 3 to load something, then use the mode 5/mode 6 pair to execute it multiple times. At least in that case you get an increment during mode 3, then another during mode 5, then a decrement during mode 6 - the result being that the use count is one *higher* than it should be. At least the structure won't be re-used out from under the original process that used it, but it does mean that tables will fill up.

A future TOS will address this issue by doing inheritance at the time the process is *started*, on the assumption that something which is created may never be started, but that anything that starts will eventually terminate. This means that the current drive, current directories, and standard handles (zero to five) will inherited at "go" time, not "load" time, so keep this in mind when coding handle-redirection routines.

COMMON PEXEC MISTAKES

Most Common Mistake #1 in attempting to do a **Pexec** is to fail to release memory owned by the parent process for use by the child. When the parent process first gains control from GEMDOS it owns all of the largest block of memory. The parent must call **Mshrink** to release memory back to the system before calling **Pexec.**

Most Common Mistake #2 is to forget to relocate the parent's stack out of the memory freed with the **Mshrink** call. Remember: an application may *not* continue to use memory it does not own.

C RUNTIME STARTUP ROUTINES

Note: The following discussion of **Mshrink** and **malloc** applies mainly to Alcyon C. Other compilers may implement **malloc** differently. For example, Mark Williams C startup code performs the **Mshrink** for you, and uses GEMDOS **Malloc** to obtain memory pools for its library **malloc** calls.

If you use Alcyon C from the developer's kit, you know that you always link with a file called GEMSTART. In GEMSTART.S, there is a lot of discussion about memory models, and then a variable you set telling how much memory you want to keep or give back to the OS. Make your choice (when in doubt, use **STACK=1**), assemble GEMSTART.S, call the result something like GEMSEXEC.O, and link the programs which **Pexec** with that file rather than the normal GEMSTART.

Your program is invoked with the address of its own basepage as the argument to a function (i.e., at 4(sp).l). The basepage structure is described in the GEMDOS manual. The interesting fields are **HITPA** (the address of the first byte *not* in your TPA), **BSSBASE** (your bss start address) and **BSSLEN** (your bss length).

Your program's stack pointer starts at **HITPA-8**, because the basepage argument and the dummy return PC on the stack take up 8 bytes. The space from **BSSBASE+BSSLEN** to your SP is the *stack+heap* space. Alcyon C library **malloc** calls use this space, moving a pointer called the *break* (in the variable **__break**, or the Alcyon C variable **_break**) up as it uses memory. The stack pointer moves down from the top as it uses memory, and if the sp and **_break** ever meet, you're out of memory. In fact, if they ever come close, within a "chicken factor" of about 1K, **malloc** will fail because it doesn't want your stack overwriting good data.

Your program keeps only the TPA it needs by calling **Mshrink**. The arguments are the address of the memory block to shrink (your program's basepage address in this case) and the new size desired. Be sure to leave enough room above your BSS for a reasonable stack, at least 2K, plus any **malloc** calls you expect to make. Let's say you're writing *make* and you want to leave about 32K for **malloc** for your dependency structures. Also, since *make* is recursive, you should leave lots of space for the stack - maybe another 16K.

SIZING UP YOUR TPA

The amount of memory that your program needs can be calculated by finding where its new top of memory needs to be:

newtop = bss base address + bss size + 16K stack + 32K heap

```
stacksize   = $4000        ; 16K
heapsize    = $8000        ; 32K

.BSS
basepage:   ds.l 1
.TEXT
   move.l   4(sp),a0       ; Get the basepage address,
   move.l   a0,basepage    ; and save it.
   move.l   $18(a0),a1     ; bss base address from basepage,
   adda.l   $1c(a0),a1     ; plus bss size,
   adda.l   #stacksize,a1  ; plus stack,
   adda.l   #heapsize,a1   ; plus heap, leaves newtop in a1
```

Since the stack pointer is at the top of your program's *current* TPA, and you're about to shrink that, your program must first move its stack:

```
   move.l  a1,sp
```

then compute its new TPA size and call **Mshrink:**

```
   suba.l   basepage,a1    ; newtop-basepage = TPA size
   move.l   a1,-(sp)       ; set up Mshrink(basepage,size)
   move.l   basepage,-(sp)
   clr.w    -(sp)
   move.w   #$4a,-(sp)     ; push function code for Mshrink
   trap     #1             ; and trap to GEMDOS
   lea      12(sp),sp      ; clean up args
```

Now that you've shrunk your program's TPA, the OS can allocate this new memory to your child and use it for other OS functions.

SELF.S

This example program is intended to be placed at the beginning of an executable file. It has several applications, but it is probably most useful for self-booting applications like video games or device drivers that can't depend on GEMDOS to load them.

The basic idea is that the entire executable file is loaded into memory, including the symbol segment (which is ignored) and relocation information. The program examines itself, determines where the relocation information is, and performs the necessary fixups. Control falls through to the code following the object file "self.o".

BSS is not cleared - it is full of symbol segment and relocation garbage, and is left as an exercise for the reader if BSS clearing is important to you.

This program is written in the MadMAC assembly language dialect, but it should work on other assemblers with minimal changes.

```
***** file "self.s"
*---- Executable file structure
.ABS
MAGIC:      ds.w    1           ; $601A magic number
TSIZE:      ds.l    1           ; size of text,
DSIZE:      ds.l    1           ; data,
BSIZE:      ds.l    1           ; BSS,
SSIZE:      ds.l    1           ; symbol segment
            ds.w    5           ; (reserved)
TSTART = *                      ; start of text

.TEXT
*---- Get control from boot ROM or GEMDOS. If the header doesn't
*---- contain $601A, assume that we've already been relocated.
     lea     *-TSTART(pc),a2    ; a2 -> PRG header
     cmp.w   #$601a,(a2)        ; correct magic#?
     bne     .exit              ; (no, don't fixup)

*- locate start of relocation information:
     lea     Start(pc),a1       ; A1 -> base of text
     move.l  a1,d1              ; D1 = base of text
     move.l  a1,a0              ; A0 = base of text
     add.l   TSIZE(a2),a1       ; A1 += tsize
     add.l   DSIZE(a2),a1       ; A1 += dsize
     add.l   SSIZE(a2),a1       ; A1 += symsize
     tst.l   (a1)               ; if (*A1 == NULL)
     beq.s   .exit              ;   then don't fix
     moveq   #0,d0              ; D0 = 0L
     add.l   (a1)+,a0           ; A0 -> first fixup
```

```
*- do relocation:
.2:  add.l   d1,(a0)          ; longword += text base
.4:  move.b  (a1)+,d0         ; get next fixup byte
     beq.s   .exit            ; if (byte == 0) break;
     cmp.b   #1,d0            ; if (byte == 1) a0 += 0xfe;
     bne.s   .3               ;
     add.w   #$00fe,a0        ; bump location pointer
     bra.s   .4               ; get next reloc byte
.3:  add.w   d0,a0            ; a0 += byte
     bra.s   .2               ; fixup, get next reloc byte

*---- Fall through to the next object file.  It had
*---- better be something prepared to handle it...
.exit:
```

PEXEC FLAVORS

Pexec mode 0 Load and go

This is the most common mode; it executes a child process like a subroutine. Returns a WORD from the child, or a negative LONG value on an error.

```
LONG Pexec(0, char *pathName, char *commandLine, char *env);
```

- **pathName** is the name of the program file to execute.
- **commandLine** is a null-terminated Pascal-style string that is copied to offset 0x80 in the child's basepage, which means that
  (1) the first character should equal the length of the string, and
  (2) the string may not exceed 125 bytes (not including the null).
- **env** is a pointer to an environment string to copy and pass to the child. An environment string is a series of null-terminated strings of the format **"VAR=value"**; the last string is followed by two zero bytes, indicating the end of the environment. If the **env** parameter is NULL, the parent's environment is copied and passed to the child.

If a **Pexec** error occurs, due to insufficient memory or some other condition, GEMDOS will return a LONG negative error number. If the **Pexec** succeeds, it will return a WORD exit code (which can not be longword negative) when the child terminates.

Pexec mode 3 Load, don't go

Used for a process that is loaded but not executed. Typically used with modes 4 and 5 to do overlays. Returns a pointer to the loaded process' basepage.

```
BASEPAGE *Pexec(3, char *pathName, char *commandLine, char
*env);
```

Parameters are the same as for mode 0, load and go.

Note: If you intend to use **Pexec(3...)** to load a process that is to be run more than once, you must ensure that:
(1) the child process does not modify its text or data segment without first initializing what it modifies - if the child does, and it is run a second time, the modified locations will not be initialized as it expects;
(2) the BSS of the child must be cleared before it is re-executed;
(3) standard file handles should be set up as they were when the child was first loaded - they are only inherited at the time of the **Pexec(3...)** call.
See also notes on memory ownership under **Pexec** modes 4 and 6.

Pexec mode 4 Just go

Executes a loaded process.
Returns values in the same manner as mode 0.

```
LONG Pexec(4, 0L, BASEPAGE *basePage, 0L);
```

The **basePage** parameter passed to GEMDOS should be the address of a process basepage which has been set up via **Pexec** 3 or 5.

If you use mode 4, you will find that when the process terminates, the memory is still around. You have to free it yourself. This is the case whether you use mode 3 or mode 5 to create the process in the first place. To free the memory, you not only have to free the basepage address, you also have to free the environment pointer. That is, do this:

```
free_process(bp)
BASEPAGE *bp;
{
     Mfree(bp->p_env);
     Mfree(bp);
}
```

Note: Use of **Pexec(4...)** is restricted by GEMDOS' concept of memory ownership. When a child is launched using this mode, memory ownership does not transfer to the child. TOS 1.4 (GEMDOS version 0x1500) adds **Pexec** mode 6 to overcome this problem. See also notes on **Pexec** mode 3 above.

Pexec mode 5 Create basepage (a.k.a. create psp)

Allocates an environment, then gets the largest block of free memory and sets it up as a prototype TPA. Returns a pointer to the memory block.

```
BASEPAGE *Pexec(5, 0L, char *commandLine, char *env);
```

GEMDOS allocates and creates an environment, then gets the largest block of free memory and creates a basepage in the first 0x100 bytes of that block. This is a "prototype" TPA, because the text, data, and bss start addresses and lengths are not set up in the basepage. The parent is responsible for loading the child into the memory block, relocating it, and filling in the missing basepage data. Probably the least used **Pexec** mode, because it involves so much work to load the child.

Note that mode 5 allocates a big TPA just like 0 and 3. If you don't need it (for example, because your program is in ROM) you can **Mshrink** the basepage to, say, $400 bytes, change **BP->hitpa** to the new ending address, fill in the (ROM) starting address of the child, then use mode 4 or 6. (This gives a user stack size of $300 bytes, which is minimal at best.) Another possibility is that the process is elsewhere in RAM (for instance, within the parent). Same trick applies. Finally, the process might be on CD ROM or something where GEMDOS doesn't know how to read it in. You can use mode 5 to build the basepage, then read in the text and data images at **BP+$100** (and fix them up if necessary), then fill in **BP+$100** as the text base address and use mode 4 or 6. Most TOS programs are going to want all of the fields of the basepage filled in, including text/data/bss base/len, but **Pexec** only requires the text base (and **hitpa** if you **Mshrink**).

Pexec mode 6 — Just go, then free

Execute a loaded process which owns its TPA. The child's memory is freed when it exits. Returns values in the same manner as mode 0.

```
LONG Pexec(6, 0L, BASEPAGE *basePage, 0L);
```

This function is available only in TOS versions 1.4 (GEMDOS version 0x1500) or higher. It functions like **Pexec** mode 4, "just go," with one important exception: memory ownership is changed to the child process. When the child terminates, GEMDOS frees any memory allocated to the child, including its TPA.

As with mode 4, the "basepage" argument must be a value returned from **Pexec** Mode 3 (load, don't go) or 5 (create basepage). It is the address of the basepage of the new process to execute. If the basepage was created with **Pexec** Mode 5, set the text-base field before using mode 6.

The difference between mode 4 and mode 6 is that mode 6 changes the owner of the new process's basepage and environment to be the new process itself. This way, when the process terminates, the basepage and environment are freed. This means that mode 3 (load, don't go) + mode 6 (just go) = mode 0 (load and go).

Pexec mode 7 — Create basepage (a.k.a. create psp) respecting PRGFLAGS

Allocates an environment, then gets the largest block of free memory and sets it up as a prototype TPA. Returns a pointer to the memory block. Unlike mode 5, the second argument is the **PRGFLAGS** value.

```
BASEPAGE *Pexec(7, prgflags, char *commandLine, char *env);
```

See **"Pexec and PRGFLAGS"** for more information.