GNO Kernel Reference Manual

By Jawaid Bazyar
Edited by Andrew Roughan and Devin Reade

19 November 1997

Contents

Chapter 1

Introduction

The GNO kernel is the heart of the GNO Multitasking Environment (GNO/ME). The GNO kernel provides a layer of communication between the shell (and shell-based programs) and the operating system, GS/OS. The kernel handles such things as multitasking, background processes, foreground processes and many other features that were not previously available on the Apple IIGS. It is these features which make GNO/ME very powerful.

This reference manual is highly technical in nature and is provided to help programmers develop utilities for the GNO Multitasking Environment. The beginner has no need to read this manual and is certainly not expected to understand its contents. However, Chapter 5 **Process Management** and Chapter 6 **Interprocess Communication** provide a good background discussion for anyone who is interested in the internal workings of the kernel.

Chapter 2

GNO Compliance

For a program to work effectively under GNO/ME, certain rules must be followed. Most of these rules boil down to one underlying concept — **never directly access features of the machine**. Always use GS/OS, the ToolBox, or GNO/ME to accomplish what you need. We have taken great care to provide the sorts of services you might need, such as checking for input without having to wait for it. GNO/ME compliance isn't just a matter of trying to make applications work well under the environment; it ensures that those applications stay compatible, no matter what changes the system goes through. Below are summarized the points you must consider when you're writing a GNO/ME compliant application.

2.1 Detecting the GNO Environment

If your application requires the GNO Kernel to be active (if it makes any kernel calls), you can make sure of this by making a **kernStatus** call at the beginning of your program. The call will return no error if the kernel is active, or it will return an error code of $0001 (Tool locator — tool not found), in which case the value returned will be invalid. The call actually returns a 1 if no error occurs, but the value returned will be indeterminate if the kernel is not active, so you should only check for an error (the function **toolerror**(3) or the variable **_toolErr** in C, the value in the A register in assembly).

You can also determine the current version of the GNO Kernel by making the **kernVersion** call. The format of the version number returned is the same as the standard ToolBox calls. For example a return value of $0201 indicates a version of 2.1.

kernStatus and **kernVersion** are defined in the `<gno/gno.h>` header file.

2.2 Terminal I/O

The Apple II has always been lacking in standardized methods for reading keyboard input and controlling the text screen. This problem was compounded when Apple stopped supporting the TextTools in favor of the GS/OS console driver. The console driver has a number of problems that prevent it from being a good solution under GNO/ME. There is high overhead involved in using it. It is generally accessed like a regular file, which means any I/O on it must filter through several layers before being handled. Even though in System 6.0.1 there is a provision for patching the low-level routines the special high-level user input features of the driver cannot be used over a modem or in a desktop program. And GS/OS must be called to access it, which means that while a console driver access is occurring, no other processes can execute. See Chapter 3 **Mutual Exclusion in GS/OS and ToolBox calls**.

GNO/ME ignores the GS/OS `.CONSOLE` driver and replaces the TextTools with a high performance, very flexible generic terminal control system. GNO/ME directly supports the console (keyboard and screen), as well as the serial ports, as terminals. In order for a user program to take advantage of these features and to be GNO/ME compliant, you must do terminal I/O only through the TextTools, or through stdin, stdout, and stderr (refNums 1,2, and 3 initially) via GS/OS. By its very nature TextTools is slow, so we recommend using them only for small and simple tasks. Calls to the GS/OS console driver will not crash the system, but they will make other processes stop until the call is completed.

You must not get input directly from the keyboard latch (memory location $E0/C000, nor may you write directly to the screen memory. GNO/ME's terminal I/O system has been designed so you don't have to do either of these things. If you need to check for keyboard input without stopping your application, you can make the appropriate **ioctl**(2) call to do what you need.

In the future, GNO/ME may provide a GNO/ME-friendly version of the GS/OS `.CONSOLE` driver.

2.3 Stack Usage

Stack space is at a premium on the Apple IIgs. Process stacks can only be located in Bank 0 — a total of 64k. This theoretical limit doesn't apply, however, as GS/OS and other bits of system software reserve a large chunk of this without any way to reclaim it. There is approximately 48K of usable stack space. This space also has to be shared with direct page space for Tools and certain types of device drivers, however. For a program to be GNO compliant, stack usage analysis must be done and acted upon. Use of the stack should be minimized so that many processes can coexist peacefully. From experience we've found that 1K usually suffices for well-written C applications, and at a maximum 4K can be allocated.

Assembly language programs tend to be very efficient when it comes to use of the stack. The 4K provided by default to applications is usually more than enough for assembly language programs. C programs can use up tremendous amounts of stack space, especially if recursion is employed or string manipulation is done without concern for stack usage; however, even assembly programs can be written poorly and use a lot of stack space. Below are some hints to keep stack usage at a minimum.

1. Avoid use of large local arrays and character strings. Instead, dynamically allocate large structures such as GS/OS strings with **malloc**(3) or the Memory Manager. Alternatively, you can designate such items as `''static''`, which causes the C compiler to allocate the space for the variable from main memory.

2. Try not to use recursion unless absolutely necessary. All recursive functions can be rewritten using standard loops and creative programming. This is a good general programming rule because your program will run faster because setting up stack frames is expensive in terms of time and memory.

3. ORCA/C 1.3 (and older) generates 8K of stack by default, in case the desktop is started up. Since GNO/ME compliant programs generally will not be desktop-based, make sure you judge how much stack your program will require and use the `#pragma stacksize` directive (or the **occ**(1) **-S** flag) to limit how much stack space ORCA/C tries to allocate for your program. Also, since ORCA/C 1.3 programs don't use the stack given them by GNO/ME and GS/OS, when you link your program include a small (256 bytes) stack segment. See the utilities sources for examples of this. ORCA/C 2.0.x (and later) allocates stack via the GS/OS supported method, so ORCA/C 2.0 programs use exactly the amount of stack specified by `#pragma stacksize`.

2.4 Disk I/O

Since the Apple IIgs doesn't have coprocessors to manage disk access and the serial ports, either of these requires the complete attention of the main 65816 processor. This can wreak havoc in an environment with slow disks or high-speed serial links, as accessing disks usually results in turning off interrupts for the duration of the access. This situation is lessened considerably with a DMA disk controller, such as the Apple High Speed SCSI or CV Technologies RamFAST. But this isn't as bad as it sounds; the IBM PC and Apple Macintosh also suffer from this problem, and the solution is robust programming. Make sure your communications protocol can handle errors where expected data doesn't arrive quite on time, or in full. The best solution would be an add-on card with serial ports and an on-board processor to make sure all serial data was received whether or not the main processor was busy (this is a hint to some enterprising hardware hacker, by the way).

Yet another concern for GNO/ME applications is file sharing. GS/OS provides support for file sharing, but it is up to the application author to use it via the requestAccess field in the `OpenGS` call. GS/OS only allows file sharing if all current references to a file (other instances of the file being opened) are read-only. GNO/ME authors should use read-only access as much as possible. For example, an editor doesn't need write permission when it's initially reading in a file. Note that the **fopen**(3) library routine in ORCA/C 1.2 does NOT support read-only mode (even if you open the file with a 'r' specificier), but it does in ORCA/C 1.3 and later.

2.5 Non-Compliant Applications

GNO/ME wasn't really designed with the intention of making EVERY program you currently run work under GNO/ME; that task would have been impossible. Our main goal was to provide a UNIX-based multitasking environment; that we have done. We made sure as many existing applications as we had time to track and debug worked with GNO/ME. The current list of compatible and non-compatible applications can be found in the file "RELEASE.NOTES" on the GNO/ME disk.

However, due to the sheer number of applications and authors, there are some programs that just plain don't work; and some that mostly work, except for annoyances such as two cursors appearing, or keyboard characters getting 'lost'. The problem here is that some programs use their own text drivers (since TextTools output was very slow at one time); since GNO/ME doesn't know about these custom drivers, it goes on buffering keyboard characters and displaying the cursor. There is a way, however, to tell GNO/ME about these programs that break GNO/ME's rules.

We've defined an auxType for S16 and EXE files, to allow distinction between programs that are GNO/ME compliant and those that are not. Setting the auxType of an application to $DC00 disables the interrupt driven keyboard buffering and turns off the GNO/ME cursor. Desktop programs use the GNO/ME keyboard I/O via the Event Manager, and thus should *not* have their auxType changed.

You can change a program's auxType with the following shell command:

```
chtyp -a \$DC00 filename
```

where filename is the name of the application. As more programmers become aware of GNO/ME and work to make their software compatible with it, this will become less of a problem, but for older applications that are unlikely to ever change (like the America OnLine software), $DC00 is a reasonable approach.

Chapter 3

Modifications to GS/OS

The GNO system modifies the behavior of a number of GS/OS calls in order to allow many programs to execute concurrently, and to effect new features. The changes are done in such a way that old software can take advantage of these new features without modification. Following is a complete description of all the changes made. Each section has details in text, followed by a list of the specific GS/OS or ToolBox calls affected.

3.1 Mutual Exclusion in GS/OS and ToolBox Calls

The Apple IIGS was not designed as a multitasking machine, and GS/OS and the Toolbox reflect this in their design. The most notable problem with making multitasking work on the Apple IIgs is the use of global (common to all processes) information, such as prefixes and direct page space for tool sets which includes information like SANE results, QuickDraw drawing information, etc. In most cases we've corrected these deficiencies by keeping track of such information on a per-process basis, that is, each process has its own copy of the information and changes to it do not affect any other process' information.

However, there were many other situations where this could not be done. Therefore, there is a limit of one process at a time inside either GS/OS or the ToolBox. GNO/ME automatically enforces this restriction whenever a tool or GS/OS call is made.

The method and details of making GS/OS calls does not change! The calls listed below have been expanded transparently. There are no new parameters and no new parameter values. In all cases, the corresponding ProDOS-16 interface calls are also supported, except ExpandPath and other calls which do not exist in ProDOS-16.

3.2 Pathnames and Prefixes

Normally under GS/OS there are 32 prefixes, and these are all under control of the current application. GNO/ME extends this concept to provide each process with it's own copies of all prefixes. When a process modifies one of these prefixes via the GS/OS SetPrefix call, it modifies only it's own copy of that prefix — the same numbered prefixes of any other processes are not modified.

Pathname processing has been expanded in GNO/ME. There are now two new special pathname operators that are accepted by any GS/OS call that takes a pathname parameter:

```
.    current working directory
..   parent directory
```

For example, presume that the current working directory (prefix 0) is `/foo/bar/moe`. "`./ls`" refers to the file "`/foo/bar/moe/ls`", and since a pathname was specified, this overrides the shell's hash table. "`../ls`" refers to "`/foo/bar/ls`". The operators can be combined, also, as in "`../../ls`" ("`/foo/ls`"), and "`./.././ls`" ("`/foo/bar/ls`"). As you can see, the '.' operator is simply removed and has no effect other than to force a full expansion of the pathname.

Shorthand device names (.d2, .d5, etc) as are used in the ORCA/Shell are available only under System Software 6.0 and later. The common pathname operator '~' (meaning the home directory) is handled by the shell; if the character appears in a GS/OS call it is not treated specially.

$2004	ChangePath
$200B	ClearBackupBit
$2001	Create
$2002	Destroy
$200E	ExpandPath
$2006	GetFileInfo
$200A	GetPrefix
$2010	Open
$2005	SetFileInfo
$2009	SetPrefix

3.3 Named Prefixes

In order to allow easy installation and configuration of third-party software into all systems, GNO/ME provides a feature called named prefixes. These prefixes are defined in the /etc/namespace file. Basically, since all UNIX systems have /bin, /usr, /etc, and other similar standard partitions, but Apple IIgs systems generally do not have these partitions, named prefixes provide a way to simulate the UNIX directories without forcing GNO/ME users to rename their partitions (an arduous and problem-filled task).

Named prefixes are handled by the GNO kernel in the same GS/OS calls described in Chapter 3 **Pathnames and Prefixes**. The format of the /etc/namespace file can be found in the **namespace**(5) manual page.

Note that if you have a physical partition that matches the name of a logical partition defined in the /etc/namespace file, then the physical parition will not be visible while running GNO.

3.4 Open File Tracking

Previously, a major problem with the way GS/OS handled open files was that unrelated programs could affect each other's open files. For example, a Desk Accessory (or a background program of any sort) could open a file and have it closed without it's knowledge by the main application program. This presented all kinds of problems for desk accessory authors. Apple presented a partial solution with System Software 5.0.4, but it wasn't enough for a true multitasking environment. GNO/ME keeps track of exactly which process opened which file. It also discontinues the concept of a global File Level, opting instead for a per-process File Level. Any operations a process performs on a file (opening, closing, etc.) do not affect any other process' files.

In addition to this behavior, when a process terminates in any manner all files that it currently has opened will be closed automatically. This prevents problems of the sort where a program under development terminates abnormally, often leaving files open and formerly necessitating a reboot.

The Flush GS/OS call is not modified in this manner as its effects are basically harmless.

The Close call accepts a refNum parameter of 0 (zero), to close all open files. This works the same way under GNO/ME, except of course that only the files of the process calling Close are in fact closed.

$2010	Open
$2014	Close
$201B	GetLevel
$201A	SetLevel

3.5 Quitting Applications

The QUIT and QuitGS calls have been modified to support the GNO/ME process scheme. Quitting to another application, whether by specifying a pathname or by popping the return stack, is accomplished with **execve**(2). When there are no entries on the return stack, the process is simply killed. See the *GS/OS Reference Manual* for more details on how the Quit stack works.

3.6 Refnums and File Descriptors

GS/OS tells you about open files in the form of refNums (reference numbers). UNIX's term for the same concept is "file descriptor". From a user's or programmer's view of GNO/ME, these terms are identical and will be used as such; which one depends on what seems most appropriate in context.

For each process, GNO/ME keeps track of which files that particular process has opened. No other process can directly access a file that another process opened (unless programmed explicitly), because it doesn't have access to any file descriptors other than its own. This is different from GS/OS in that GS/OS allows access to a file even if a program guessed the refNum, either deliberately or accidentally. This is one of the aspects of process protection in GNO/ME.

All of the various I/O mechanisms that GNO/ME supports (files, pipes, and TTYs) are handled with the same GS/OS calls you are familiar with. When you create a pipe, for example, you are returned file descriptors which, because of synonymity with refNums, you can use in GS/OS calls. Not all GS/OS calls that deal with files are applicable to a particular file descriptor; these are detailed in the sections on pipes and TTYs.

GNO/ME sets no limit on the number of files a process may have open at one time. (Most UNIX's have a set limit at 32).

3.7 GNO/ME Character Devices

GNO/ME supports a new range of character device drivers. These drivers are not installed like normal GS/OS drivers, but they are accessed the same way. There are the following built-in drivers:

.TTYCO	This is the GNO/ME console driver. The driver supports the TextTools Pascal control codes, plus a few GNO/ME specific ones. These are documented in Chapter 4 **TextTools Replacement**. This driver is highly optimized both through the GS/OS and TextTools interfaces.
.TTYA[0-9,A-F]	
.PTYQ[0-9,A-F]	Pseudo-terminal devices; PTYs are used for inter-process communication and in network activities.
.NULL	This driver is a bit bucket. Any data written to it is ignored, and any attempt to read from it results in an end-of-file error ($4C).

Just as with GS/OS devices, these GNO/ME drivers are accessed with the same Open, Read, Write, and Close calls that are used on files. Unlike GS/OS character devices, the characteristics of GNO/ME drivers are controlled through the **ioctl**(2) system call. The GS/OS Device calls (like DInfo, DStatus) are not applicable to GNO/ME drivers. See the **ioctl**(2) and **tty**(4) man pages for details.

Some GS/OS calls will return an error when given a refNum referring to a GNO/ME character driver or pipe because the concepts simply do not apply. The error returned will be $58 (Not a Block Device), and the calls are as follows:

$2016	SetMark
$2017	GetMark
$2018	SetEOF
$2019	GetEOF
$2015	Flush
$201C	GetDirEntry

GNO/ME loaded drivers (generally for serial communications, but other uses are possible) are configured in the **/etc/tty.config** file. Each line in **/etc/tty.config** describes one driver. The format of each line is:

```
filename slot devname
```

devname is the name of the device as it will be accessed (for example, **.ttya**).

slot is the slot in the device table from where the device will be accessed; it may refer to one of the physical expansion slots, as TextTools will use the specified driver when redirecting output to a slot. The **modem** and **printer** port drivers are configured for slots 2 and 1, respectively.

Pseudo-terminals are pre-configured into the kernel. PTYs are discussed further in Chapter 6 *Psuedo-Terminals PTYs*.

Since .ttyco and the pseudo-terminals are preconfigured in the GNO kernel, entries for these devices do not appear in **/etc/tty.config**.

3.8 Restartability

GS/OS supports the concept of program "restartability". This allows programs which are written in a certain way to remain in memory in a purgeable state so that if they are invoked again, and their memory has not been purged, they can be restarted without any disk access. This greatly increases the speed with which restartable programs can be executed.

The ORCA environment specifies whether or not a program is restartable via a flag character in the SYSCMND file. The GS/OS standard method, however, is to set the appropriate flags bit in the GS/OS Quit call. This is the method that GNO/ME supports. Provided with the GNO/ME standard library is a routine **rexit**(3). **rexit**(3) only works with ORCA/C 2.0. **rexit**(3) works just like the normal C **exit**(3) call but it sets the restart flag when calling QuitGS.

The standard ORCA/C 1.3 libraries are not restartable, but the ORCA/C 2.0 libraries are.

3.9 Miscellaneous

The following miscellaneous GS/OS calls have also been modified for GNO/ME:

$2027 GetName	Returns the name on disk of the process. This only returns valid information after an **execve**(2).
$2003 OSShutdown	This call has been modified to kill all processes before performing the actual shutdown operation.

Chapter 4

Modifications to the ToolBox

Several changes have been made to the ToolBox, the most major of which is the replacement of the entire TextTools tool set. The TextTools were replaced for a number of reasons — better control over text I/O, increased speed, and emulation of ORCA's redirection system with as little overhead as possible. Other changes were made to modify the behavior of some tool calls to be more consistent with the idea of a multitasking environment.

4.1 TextTools Replacement

The changes to the TextTools have turned it into a much more powerful general I/O manager. The TextTools now intrinsically handle pipes and redirection, and you can install custom drivers for TextTools to use. Also, the TextTools have had their old slot-dependence removed; the parameter that used to refer to 'slot' in the original texttools calls now refers to a driver number. A summary of driver numbers (including those that come pre-installed into GNO) are as follows:

0	null device driver
1	serial driver (for printer port compatibility)
2	serial driver (for modem port compatibility)
3	console driver (Pascal-compatible 80-column text screen)
4–5	user installed

See Chapter 3 **GNO/ME Character Devices**, for information on configuring these drivers.

There are also new device types in the TextTools; the complete list of supported device types and what their slotNum's (from SetInputDevice, SetOutputDevice, etc) mean is as follows:

Type	**Use**	**slotNum**
0	Used to be BASIC text drivers. These are no longer supported under GNO/ME, and setting I/O to a BASIC driver actually selects a Pascal driver.	Not applicable.
1	Pascal text driver. This is one of the drivers specified in /etc/ttys or built-in to GNO/ME.	Driver number as listed above.
2	RAM-based Driver (documented in *ToolBox Reference Volume 2*)	Pointer to the RAM-based driver's jump table.
3	File redirection	refNum (file descriptor) of the file to access through TextTools.

The new console driver supports all the features of the old 80-column Pascal firmware, and adds a few extensions, with one exception — the codes that switched between 40 and 80 columns modes are not supported. It is not compatible with the GS/OS ".console" driver. The control codes supported are as follows:

Hex	ASCII	Action
01	CTRL-A	Set cursor to flashing block
02	CTRL-B	Set cursor to flashing underscore
03	CTRL-C	Begin “Set Text Window” sequence
05	CTRL-E	Cursor on
06	CTRL-F	Cursor off
07	CTRL-G	Perform FlexBeep
08	CTRL-H	Move left one character
09	CTRL-I	Tab
0A	CTRL-J	Move down a line
0B	CTRL-K	Clear to EOP (end of screen)
0C	CTRL-L	Clear screen, home cursor
0D	CTRL-M	Move cursor to left edge of line
0E	CTRL-N	Normal text
0F	CTRL-O	Inverse text
11	CTRL-Q	Insert a blank line at the current cursor position
12	CTRL-R	Delete the line at the current cursor position.
15	CTRL-U	Move cursor right one character
16	CTRL-V	Scroll display down one line
17	CTRL-W	Scroll display up one line
18	CTRL-X	Normal text, mousetext off
19	CTRL-Y	Home cursor
1A	CTRL-Z	Clear entire line
1B	CTRL-[	MouseText on
1C	CTRL-“	Move cursor one character to the right
1D	CTRL-]	Clear to end of line
1E	CTRL-ˆ	Goto XY
1F	CTRL-_	Move up one line

(**Note**: the *Apple IIgs Firmware Reference* incorrectly has codes 05 and 06 reversed. The codes listed here are correct for both GNO/ME and the Apple IIgs 80-column firmware.)

FlexBeep is a custom beep routine that doesn't turn off interrupts for the duration of the noise as does the default Apple IIgs beep. This means that the beep could sound funny from time to time, but it allows other processes to keep running. We also added two control codes to control what kind of cursor is used. There are two types available as in most text-based software; they are underscore for 'insert' mode, and block for 'overstrike'. You may, of course, use whichever cursor you like. For example, a communications program won't have need of insert mode, so it can leave the choice up to the user.

The Set Text Window sequence (begun by a $03 code) works as follows:

```
CTRL-C '[' LEFT RIGHT TOP BOTTOM
```

CTRL-C is of course hex $03, and '[' is the open bracket character ($5B). TOP, BOTTOM, LEFT, and RIGHT are single-byte ASCII values that represent the margin settings. Values for TOP and BOTTOM range from 0 to 23; LEFT and RIGHT range from 0 to 79. TOP must be numerically less than BOTTOM; LEFT must be less than RIGHT. Any impossible settings are ignored, and defaults are used instead. The extra '[' in the sequence helps prevent the screen from becoming confused in the event that random data is printed to the screen.

After a successful Set Text Window sequence, only the portion of the screen inside the 'window' will be accessible, and only the window will scroll; any text outside the window is not affected.

The cursor blinks at a rate defined by the **Control Panel/Options/Cursor Flash** setting. Far left is no blinking (solid), and far right is extremely fast blinking.

`ReadLine` ($240C) now sports a complete line editor unlike the old TextTools version. Following is a list of the editor commands.

EOL	Terminates input (EOL is a parameter to the _ReadLine call).
LEFT-ARROW	Move cursor to the left.
RIGHT-ARROW	Move cursor to right. It won't go past rightmost character.
DELETE	Delete the character to the left of the cursor.
CTRL-D	Delete character under the cursor.
OA-D	Delete character under the cursor.
OA-E	Toggles between overwrite and insert mode.

`ReadChar` ($220C) has also been changed. The character returned may now contain the key modification flags ($C025) in the upper byte and the character typed in the lower byte. This is still compatible with the old TextTools ReadChar. To get the keyMod flags, call `SetInGlobals` ($090C) and set the upper byte of the AND mask to $FF. Typical parameters for `SetInGlobals` to get this information are: ANDmask = $FF7F, ORmask = $0000.

The default I/O masks have also been changed. They are now ANDmask = $00FF, ORmask = $0000. They are set this way to extend the range of data that can be sent through TextTools. GNO/ME Character drivers do not, like the previous TextTools driver, require the hi-bit to be set.

The new TextTools are completely reentrant. This means that any number of processes may be executing TextTools calls at the same time, increasing system performance somewhat. The TextTools are also the only toolset which is not mutexed.

The GNO/ME console driver also supports flow-control in the form of Control-S and Control-Q. Control-S is used to stop screen output, and Control-Q is used to resume screen output.

4.2 SysFailMgr

The MiscTool call SysFailMgr ($1503) has been modified so that a process calling it is simply killed, instead of causing system operation to stop. This was done because many programs use SysFailMgr when a simple error message would have sufficed. There are, however, some tool and GS/OS errors which are truly system failure messages, and these do cause system operation to stop. These errors are as follows:

$0305	Damaged heartbeat queue detected.
$0308	Damaged heartbeat queue detected.
$0681	Event queue damaged.
$0682	Queue handle damaged.
$08FF	Unclaimed sound interrupt.

What the system does after displaying the message is the same as for a system panic.

4.3 The Resource Manager

The Resource Manager has been modified in some subtle ways. First, GNO/ME makes sure that the CurResourceApp value is always correct before a process makes a Resource Manager call. Second, all open resource files are the property of the Kernel. When a GetOpenFileRefnum call is made, a new refnum is **dup**(2)'d to allow the process to access the file. Having the Kernel control resource files also allows all processes to share SYS.RESOURCES without requiring each process to explicitly open it.

4.4 The Event Manager

GNO/ME starts up the Event Manager so it is always available to the kernel and shell utilities. Changes were made so that the Event Manager obtains keystrokes from the GNO/ME console driver (.ttyco). This allows UNIX-style utilities and desktop applications to share the keyboard in a cooperative manner. This also makes it possible to suspend desktop applications; see Chapter 7, **Suspend NDA**.

EMStartUp sets the GNO console driver to RAW mode via an **ioctl**(2) call, to allow the Event Manager to get single keystrokes at a time, and to prevent users from being able to kill the desktop application with Ĉ or other interrupt characters. The four "GetEvent" routines, GetNextEvent, GetOSEvent, EventAvail, and OSEventAvail now poll the console for input characters instead of using an interrupt handler.

4.5 The Control Panel

In most cases, the CDA menu is executed as an interrupt handler. Since the Apple IIgs interrupt handler firmware isn't reentrant, task switching is not allowed to occur while the control panel is active. This basically means that all processes grind to a halt. In many ways, however, this is not undesirable. It definitely eases debugging, since a static system is much easier to deal with than a dynamic system. Also, CDAs assume they have full control of the text screen; multitasking CDAs would confuse and be confused in terms of output.

During the execution of the Control Panel, the original non-GNO/ME TextTools tool is reinstalled to prevent compatibility problems. Another step, taken to maintain user sanity, makes CDAs run under the kernel's process ID.

All the changes were made to two tool calls: `SaveAll` ($0B05) and `RestAll` ($0C05).

4.6 QDStartup

The `QDStartup` ($0204) call has been modified to signal an error and terminate any process that tries to make the call when it's controlling terminal is not the Apple IIgs console. This prevents a user on a remote terminal from bringing up a desktop application on the console, an operation he could not escape from and one that would greatly annoy the user at the console.

Another change ensures that an attempt to execute two graphics-based applications concurrently will fail; the second process that tries to call `QDStartUp` is killed and a diagnostic message is displayed on the screen.

Chapter 5

Process Management

Before discussing process management using Kernel calls, it would be wise to define just exactly what we refer to when we say *process*. A process is generally considered to be a program in execution. "A program is a passive entity, while a process is an active entity." (Operating Systems Concepts p.73, Silberschatz and Peterson, Addison-Wesley, 1989). The concept of process includes the information a computer needs to execute a program (such as the program counter, register values, etc).

In order to execute multiple processes, the operating system (GNO/ME and GS/OS in this case) has to make decisions about which process to run and when. GNO/ME supports what is termed *preemptive multitasking*, which means that processes are interrupted after a certain amount of time (their time slice), at which point another process is allowed to run. The changing of machine registers to make the processor execute a different process is called a *context switch*, and the information the operating system needs to do this is called its *context*. The GNO kernel maintains a list of all active processes, and assigns time slices to each process according to their order in the list. When the kernel has run through all the processes, it starts again at the beginning of the list. This is called *round-robin scheduling*. Under certain circumstances, a process can actually execute longer than its allotted time slice because task switches are not allowed during a GS/OS or ToolBox call. In these cases, as soon as the system call is finished the process is interrupted.

Processes can give up the rest of their time slice voluntarily (but not necessarily explicitly) in a number of ways, terminal input being the most common. In this case, the rest of the time slice is allocated to the next process in line (to help smooth out scheduling). A process waiting on some event to happen is termed *blocked*. There are many ways this can happen, and each will be mentioned in its place.

An important item to remember is the *process ID*. This is a number which uniquely identifies a process. The ID is assigned when the process is created, and is made available for reassignment when the process terminates. A great many system calls require process IDs as input. Do not confuse this with a userID, which is a system for keeping track of memory allocation by various parts of the system, and is handled (pardon the pun) by the Memory Manager tool set. Also, do not confuse Memory Manager userID's with Unix user ID's — numbers which are assigned to the various human users of a multiuser machine.

There are two methods for creating new processes: the system call **fork**(2) (or **fork2**(2)) and the library routine **exec**(3) (specifics for calling these functions and others is in Appendix A *Making System Calls*). **fork** starts up a process which begins execution at an address you specify. **exec** starts up a process by loading an executable file (S16 or EXE). **fork** is used mainly for use inside a specific application, such as running shell built-ins in the background, or setting up independent entities inside a program. Forked processes have some limitations, due to the hardware design of the Apple IIgs. The parent process (the process which called fork) must still exist when the children die, either via **kill** or by simply exiting. This is because the forked children share the same memory space as the parent; the memory the children execute from is tagged with the parent's userID. If the parent terminated before the children, the children's code would be deallocated and likely overwritten. A second caveat with **fork** is the difference between it's UNIX counterpart. UNIX **fork** begins executing the child at a point directly after the call to **fork**. This cannot be accomplished on the Apple IIgs because virtual memory is required for such an operation; thus the need to specify a **fork** child as a C function. Note that an appropriately written assembly language program need not necessarily have these restrictions. When a process is forked, the child process is given it's own direct page and stack space under a newly allocated userID, so that when the child terminates this memory is automatically freed.

exec(3) is used when the process you wish to start is a GS/OS load file (file type S16 and EXE). **exec** follows the procedure outlined in the *GS/OS Reference Manual* for executing a program, and sets up the new program's environment as it expects. After **exec** has loaded the program and set up it's environment, the new process is started and **exec** returns immediately.

Both **fork**(2) and **exec**(3) return the process ID of the child. The parent may use this process ID to send *signals* to the child, or simply wait for the child to exit with the **wait**(2) system call; indeed, this is the most common use. Whenever a child process terminates or is stopped (See Chapter 6 *Interprocess Communication*), the kernel creates a packet of information which is then made available to the process' parent. If the parent is currently inside a wait call, the call returns with the information. If the parent is off doing something else, the kernel sends the parent process a `SIGCHLD` signal. The default is to ignore `SIGCHLD`, but a common technique is to install a handler for `SIGCHLD`, and to make a **wait** call inside the handler to retrieve the relevant information.

exec(3) is actually implemented as two other system calls: **fork**(2), and one called **execve**(2). **execve** loads a program from an executable file, and begins executing it. The current process' memory is deallocated. The shell uses a **fork**/**execve** pair explicitly, so it can set up redirection and handle job control.

5.1 Process Table

Information about processes is maintained in the process table, which contains one entry for each possible process (**NPROC**, defined in the C header file `<gno/conf.h>`. There is other per-process information spread about the kernel, but those are usually used for maintaining compatibility with older software, and thus are not described here. Please note that the data in this section is informational only (e.g. for programs like **ps**(1)). Do not attempt to modify kernel data structures or the GNO Kernel will likely respond with a resounding crash. Only 'interesting' fields are documented.

Copies of process entries should be obtained by using the Kernel Virtual Memory (KVM) routines (**kvm_open**(2), and so forth). These are documented in the electronic manual pages.

processState Processes have a state associate with them. The state of the process is a description of what the process is doing. The possible process states (as listed in `<gno/proc.h>` and described here) are:

RUNNING	The process is currently in execution.
READY	The process is not currently executing, but is ready to be executed as soon as it is assigned a time slice.
BLOCKED	The process is waiting for a slow I/O operation to complete (for instance, a read from a TTY).
NEW	The process has been created, but has not executed yet.
SUSPENDED	The process was stopped with SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU.
WAITING	The process is waiting on a semaphore "signal" operation. Programs waiting for data from a pipe have this state.
WAITSIGCH	The process is waiting to receive a SIGCHLD signal.
PAUSED	The process is waiting for any signal.

ttyID The device number of the controlling TTY for this process. This is not a GS/OS refnum; rather, it is an index into the kernel's internal character device table. The value of this field can be interpreted as follows:

- **0** .null
- **1** .ttya
- **2** .ttyb
- **3** .ttyco
- **6** .ptyq0 pty0 master side
- **7** .ttyq0 pty0 slave side

Other values may be appropriate depending on the `/etc/tty.config` file. Namely, **1** and **2** (by default the modem and printer port drivers), and **4** and **5** (unassigned by default) may be assigned to different devices.

ticks The number of full ticks this process has executed. If a process gives up it's time slice due to an I/O operation, this value is not incremented. A tick is 1/60 second.

alarmCount If an **alarm**(2) request was made, this is the number of seconds remaining until the process is sent SIGALRM.

openFiles This is a structure which stores information about the files a process has open. See `struct ftable` and `struct fdentry` in `<gno/proc.h>` .

irq_A, irq_X, irq_Y, irq_S, irq_D, irq_B, irq_P, irq_state, irq_PC, irq_K Context information for the process. These fields are the values of the 65816 registers at the last context switch. They only truly represent the machine state of the process if the process is not RUNNING.

args This is a NULL-terminated (C-style) string that contains the command line with which the process was invoked. This string begins with "BYTEWRKS", the shell identifier.

For more details and an example of how to investigate process information, look at the source code for the "GNO Snooper CDA".

5.2 Task Switching

As mentioned earlier, user code can often unwittingly initiate a context switch by reading from the console (and other miscellaneous things). There are a few situations where this can cause a problem, namely inside interrupt handlers. While the kernel makes an attempt to prevent this, it cannot predict every conceivable problem. The kernel attempts to detect and prevent context switches inside interrupt handlers by checking for the following situations.

- Is the system busy flag non-zero? (The busy flag is located at address `$E100FF`.)

- Is the "No-Compact" flag set? (Located at `$E100CB`.)
- Does the stack pointer point to anything in thr range `$0100-$01FF`?
- Is the interrupt bit in the processor status register set?

If any of these conditions are met, a context switch will not take place. This can cause problems in certain circumstances. The basic rule is to avoid making Kernel calls that might cause a context switch or change in process state from inside an interrupt handler. This includes the following:

- reading from the console
- accessing a pipe
- any of the following kernel traps: **_execve**(2) (or other calls in the **exec** family), **fork**(2), **fork2**(2), **kill**(2), **pause**(2), **procreceive**(2), **sigpause**(2), or **wait**(2).

Calls such as **procsend**(2), however, may be used from inside an interrupt handler, and in fact are very useful in such situations.

5.3 Job Control

Job control is a feature of the kernel that helps processes orderly share a terminal. It prevents such quandaries as "What happens when two processes try to read from the terminal at the same time?".

Job control works by assigning related processes to a *process group*. For example, all of the processes in a pipeline belong to one process group. Terminal device drivers also belong to process groups, and when the process group of a job does not match that of its *controlling terminal* the job is said to be in the background. Background jobs have access to their controlling terminal restricted in certain ways.

- If a background job attempts to read from the terminal, the kernel suspends the process by sending the `SIGTTIN` signal.
- The interrupt signals `SIGTSTP` and `SIGINT`, generated by $\hat{\mathrm{Z}}$ and $\hat{\mathrm{C}}$ respectively, are sent only to the foregound job. This allows backgrounded jobs to proceed without interruption.

- Certain **ioctl**(2) calls cannot be made by a background job; the result is a `SIGTTIN` signal.

Job control is accessed by software through the **tcnewpgrp**, **tctpgrp, and settpgrp**(2) system calls. See the **jobcontrol**(2) and **ioctl**(2) man pages.

Chapter 6

Interprocess Communication

Oh, give me a home
Where the semaphores roam,
and the pipes are not deadlocked all day ...
— unknown western hero

The term Interprocess Communication (*IPC*) covers a large range of operating system features. Any time a process needs to send information to another process some form of IPC is used. The GNO Kernel provides several basic types: semaphores, signals, pipes, messages, ports, and pseudo-terminals. These IPC mechanisms cover almost every conceivable communication task a program could possibly need to do.

6.1 Semaphores

In the days before radio, when two ships wished to communicate with each other to decide who was going first to traverse a channel wide enough only for one, they used multicolored flags called semaphores. Computer scientists, being great lovers of anachronistic terms, adopted the term and meaning of the word semaphore to create a way for processes to communicate when accessing shared information.

GNO/ME, like other multitasking systems, provides applications with semaphore routines. Semaphores sequentialize access to data by concurrently executing processes. You should use semaphores whenever two or more processes want to access shared information. For example, suppose there were three processes, each of which accepted input from user terminals and stored this input into a buffer in memory. Suppose also that there is another process which reads the information out of the buffer and stores it on disk. If one of the processes putting information in the buffer (writer process) was in the middle of storing information in the buffer when a context switch occurred, and one of the other processes then accessed the buffer, things would get really confused. Code that accesses the buffer should not be interrupted by another process that manipulates the buffer; this code is called a *critical section*; in order to operate properly, this code must not be interrupted by any other attempts to access the buffer.

To prevent the buffer from becoming corrupted, a semaphore would be employed. As part of it's startup, the application that started up the other processes would also create a semaphore using the **screate**(2) system call with a parameter of 1. This number means (among other things) that only one process at a time can enter the critical section, and is called the *count*.

When a process wishes to access the buffer, it makes a **swait**(2), giving as argument the semaphore number returned by **screate**(2). When it's done with the buffer, it makes an **ssignal**(2) call to indicate this fact.

This is what happens when **swait** is called: the kernel first decrements the count. If the count is then less than zero, the kernel suspends the process, because a count of less than zero indicates that another process is already inside a critical section. This suspended state is called 'waiting' (hence the name of **swait**). Every process that tries to call **swait** with count ¡ 0 will be suspended; a queue of all the processes currently waiting on the semaphore is associated with the semaphore.

Now, when the process inside the critical section leaves and executes **ssignal**, the kernel increments the count. If there are processes waiting for the semaphore, the kernel chooses one arbitrarily and restarts it. When the process resumes execution at its next time slice, its **swait** call will finish executing and it will have exclusive control of the critical section. This cycle continues until there are no processes waiting on the semaphore, at which point its count will have returned to 1.

When the semaphore is no longer needed, you should dispose of it with the **sdelete**(2) call. This call frees any processes that might be waiting on the semaphore and returns the semaphore to the semaphore pool.

One must be careful in use of semaphores or *deadlock* can occur.

There are (believe it or not) many situations in everyday programming when you may need semaphores, moreso than real UNIX systems due to the Apple IIgs's lack of virtual memory. The most common of these is your C or Pascal compiler's stdio library; these are routines like **printf**(3) and **writeln**(3). In many cases, these libraries use global variables and buffers. If you write a program which forks a child process that shares program code with the parent process (i.e. doesn't **execve**(2) to another executable), and that child and the parent both use *non-reentrant* library calls, the library will become confused. In the case of text output routines, this usually results in garbaged output.

Other library routines can have more disastrous results. For example, if a parent's **free**(3) or **dispose**(3) memory management call is interrupted, and the child makes a similar call during this time, the linked lists that the library maintains to keep track of allocated memory could become corrupted, resulting most likely in a program crash.

GNO/ME provides *mutual exclusion* (i.e., lets a maximum of one process at a time execute the code) automatically around all Toolbox and GS/OS calls as described in Chapter 3, and also uses semaphores internally in many other places. Any budding GNO/ME programmer is well advised to experiment with semaphores to get a feel for when and where they should be used. Examples of semaphore use can be found in the sample source code, notably `dp.c` (Dining Philosophers demo) and `pipe*.c` (a sample implementation of pipes written entirely in C).

6.2 Signals

Another method of IPC is software signals. Signals are similar to hardware interrupts in that they are asynchronous; that is, a process receiving a signal does not have to be in a special mode, does not have to wait for it. Also like hardware interrupts, a process can install signal handlers to take special action when a signal arrives. Unlike hardware interrupts, signals are defined and handled entirely through software.

Signals are generally used to tell a process of some event that has occurred. Between the system-defined and user-defined signals, there is a lot of things you can do. GNO/ME currently defines 32 different signals. A list of signals and their codes can be found in **signal**(2) and the header file `<gno/signal.h>`.

There are three types of default actions that occur upon receipt of a signal. The process receiving the signal might be terminated, or stopped; or, the signal might be ignored. The default action of any signal can be changed by a process, with some exceptions. Not all of the defined signals are currently used by GNO/ME, as some are not applicable to the Apple IIgs, or represent UNIX features not yet implemented in GNO/ME . Here is a list of the signals that are used by GNO/ME.

SIGINT This signal is sent to the foreground job when a user types Ĉ at the terminal keyboard.

SIGKILL The default action of this signal (termination) cannot be changed. This provides a sure-fire means of stopping an otherwise unstoppable process.

SIGPIPE Whenever a process tries to write on a pipe with no readers, it is sent this signal. SIGALRM SIGALRM is sent when an alarm timer expires (counts down to zero). An application can start an alarm timer with the **alarm**(2) system call.

SIGTERM This is the default signal sent by **kill**(1). Use of this signal allows applications to clean up (delete temporary files, free system resources like semaphores, etc) before terminating at the user's bequest.

SIGSTOP This signal is used to stop a process' execution temporarily. Like SIGKILL, processes are not allowed to install a handler for this signal.

SIGCONT To restart a stopped process, send this signal.

SIGTSTP This is similar to SIGSTOP, but is sent when the user types Ẑ at the keyboard. Unlike SIGSTOP, this signal can be ignored, caught, or blocked.

SIGCHLD A process receives this signal whenever a child process is stopped or terminates. **gsh** uses this to keep track of jobs, and the wait system call waits for this signal to arrive before exiting.

SIGTTIN This signal also stops a process. It is sent to background jobs that try to get input from the terminal.

SIGTTOU Similar to SIGTTIN, but is sent when a background process tries to write to the terminal. This behavior is optional and is by default turned off.

SIGUSR1, SIGUSR2 These two signals are reserved for application authors. Their meaning will change from application to application.

As you can see, signals are used by many aspects of the system. For detailed information on what various signals mean, consult the appropriate electronic manual page — see **tty**(4), **wait**(2), and **signal**(2).

For an example of signal usage, consider a print spooler. A print spooler takes files that are put in the spool directory on a disk and sends the data in the files to a printer. There are generally two parts to a print spooler: The *daemon*, a process that resides in memory and performs the transfer of data to the printer in the background; and the spooler. There can be many different types of spoolers, say one for desktop printing, one for printing source code, etc. To communicate to the daemon that they have just placed a new file in the spool directory, the spoolers could send the daemon SIGUSR. The daemon will have a handler for SIGUSR, and that handler will locate the file and set things up so the print will begin. Note that the actual implementation of the print spooling system in GNO/ME, **lpr**(1) and **lpd**(8), is somewhat more complex and uses messages and ports instead of signals. However, an earlier version of the spooler software *did* use signals for communication.

Signals should not be sent from inside an interrupt handler, nor from inside a GS/OS or Toolbox call. Window Manager update routines are a prime example of code that should not send signals; they are executed as part of a tool call. The GS/OS aspect of this limitation is a little harder to come up against. GS/OS does maintain a software signal facility of it's own, used to notify programs when certain low-level events have occurred. Do not confuse these GS/OS signals with GNO/ME signals, and above all, don't send a GNO/ME signal from a GS/OS signal handler.

When a process receives a signal for which it has installed a handler, what occurs is similar to a context switch. The process' context is saved on the stack, and the context is set so that the signal handler routine will be executed. Since the old context is stored on the stack, the signal handler may if it wishes return to some other part of the program. It accomplishes this by setting the stack pointer to a value saved earlier in the program and jumping to the appropriate place. Jumps like this can be made with C's **setjmp**(3) and **longjmp** (3) functions. The following bit of code demonstrates this ability.

```
void sighandler (int sig, int code)
{
    printf("Got a signal!");
    longjmp(jmp_buf);
}

void routine(void)
{
    signal(SIGUSR, sighandler);
    if (setjmp(jmp_buf)) {
        printf("Finally done! Sorry for all that...\n");
    } else {
        while(1) {
            printf("While I wait I will annoy you!\n");
        }
    }
}
```

This program basically prints an annoying message over and over until SIGUSR is received. At that point, the handler prints “Got a Signal!” and jumps back to the part of the if statement that prints an apology. If the signal handler hadn’t made the **longjmp**, when the handler exited control would have returned to the exact place in the **while** loop that was interrupted.

Similar techniques can be applied in assembly language.

6.3 Pipes

This third form of IPC implemented in GNO/ME is one of the most powerful features ever put into an operating system. A pipe is a conduit for information from one process to another. Pipes are accessed just like regular files; the same GS/OS and ToolBox calls currently used to manipulate files are also used to manipulate pipes. When combined with GNO/ME standard I/O features, pipes become very powerful indeed. For examples on how to use **gsh** to connect applications with pipes, see the *GNO Shell Reference Manual*.

Pipes are unidirectional channels between processes. Pipes are created with the **pipe**(2) system call, which returns two GS/OS refNums; one for the write end, and one for the read end. An attempt to read from the write end or vice-versa results in an error.

Pipes under GNO/ME are implemented as a circular buffer of 4096 bytes. Semaphores are employed to prevent the buffer from overflowing, and to maintain synchronization between the processes accessing the pipe. This is done by creating two semaphores; their counts indicate how many bytes are available to be read and how many bytes may be written to the buffer (0 and 4096 initially). If an I/O operation on the pipe would result in the buffer being emptied or filled, the calling process is blocked until the data (or space) becomes available.

The usual method of setting up a pipeline between processes, used by **gsh** and utilities such as script, is to make the **pipe** call and then **fork**(2) off the processes to be connected by the pipe.

```
/* No error checking is done in this fragment.  This is
 * left as an exercise for the reader.
 */

int fd[2];
int
testPipe(void)
```

```
{
    pipe(fd);      /* create the pipe */
    fork(writer);  /* create the writer process */
    fork(reader);  /* create the reader process */
    close(fd[0]);  /* we don't need the pipe anymore, because */
    close(fd[1]);  /*    the children inherited them          */

    { wait for children to terminate ... }
}

void
writer(void) {
    /* reset the standard output to the write pipe */
    dup2(STDOUT_FILENO, fd[1]);

    /* we don't need the read end */
    close(fd[0]);
    { exec writer process ...}
}

void
reader(void) {
    /* reset the standard input to the write pipe */
    dup2(STDIN_FILENO, fd[0]);

    /* we don't need the write end */
    close(fd[1]);
    { exec reader process ...}
}
```

Recall that when a new process is forked, it inherits all of the open files of it's parent; thus, the two children here inherit not only standard I/O but also the pipe. After the forks, the parent process closes the pipe and each of the child processes closes the end of the pipe it doesn't use. This is actually a necessary step because the kernel must know when the reader has terminated in order to also stop the writer (by sending `SIGPIPE`. Since each open refNum to the read end of the pipe is counted as a reader, any unnecessary copies must be closed.

For further examples of implementing and programming pipes, see the sample source code for `pipe.c`.

6.4 Messages

GNO's Message IPC is borrowed from the XINU Operating System, designed by Douglas Comer. It is a simple way to send a datum (a message) to another process. Messages are 32-bit (4-byte) longwords.

The Message IPC is centered around two calls, **procsend**(2) and **procreceive**(2). The **procsend** call sends a message to a specified process ID. To access that message, a process must use **procreceive**. If no message is waiting for a process when it calls **procreceive**, the process will block until a message becomes available.

Since a process can only have one pending message, the Message IPC is useful mostly in applications where two or more cooperating processes only occasionally need to signal each other; for example, the **init**(8) program communicates with the **initd** daemon by sending messages. Various attributes are encoded in the 32-bit value sent to **initd**(8) to instruct it on how to change its state.

If a process doesn't want to indefinitely block waiting for a message, it can call **procrecvtim**(2). The **procrecvtim** call accepts a timeout parameter which indicates the maximum amount of time to wait for a message.

6.5 Ports

GNO/ME Ports IPC can be thought of as an extended version of Messages. Whereas only one message can be pending at once, a port can contain any number of pending messages (up to a limit defined when an application creates a port).

Like Messages, Ports transmit 32-bit values between processes. The calls **psend**(2) and **preceive**(2) work similarly to their Message counterparts.

A Port is created with the **pcreate**(2) call. The application specifies the size of the port in this call. When the application is done with the port, it should call **pdelete**(2) to free up the resources used by the port.

One of the most important aspects of ports is the ability to bind a *name* to a port. Whereas many of GNO/ME IPC mechanisms require the communicating processes to be related in some way (common children of the same parent, for instance) being able to give a port a name means that totally unrelated processes can communicate. For example, the GNO/ME print spooling system uses a named port for communicating information about the addition of new jobs to the print queue. The printer daemon, **lpd**(8), creates a port with a specific name; the name is defined by the author of the print daemon; any application that wishes to have the daemon print a spool file also knows this name. (The standard print daemon uses the name "LPDPrinter"). The name allows an application to find lpd's port regardless of the actual numeric port ID (which might be different from system to system, or even from session to session on the same machine).

Names are bound to ports with the **pbind**(2) call. The numeric port ID can be obtained by passing a name to **pgetport**(2).

6.6 Pseudo-Terminals (PTYs)

Pseudo-terminals are a bi-directional communication channel that can be used to connect two processes (or more correctly, a process group to another process). You may (correctly) ask why two pipes would not do the same thing; the answer is that a lot of modern UNIX software relies on the way the terminal interface works, and thus would malfunction when presented with a pipe as standard input. What PTYs provide is a lot like two pipes, but with a TTY interface.

PTYs can be used in a number of important and exciting applications, such as windowing systems and 'script-driven' interfaces.

Windowing systems like the UNIX X windowing system (known as just "**X**") use PTYs to give a process group an interface that looks exactly like a real terminal; however, the 'terminal' in this case is actually a window in a graphics-based system. The program that manages the window ('xterm' in **X**) is called the *master*. It is responsible for setting up the PTY, and starting up the process with redirection (usually a shell) that is to run in the window. The process running in the window is called the *slave*.

To allocate a PTY, the master opens in turn each PTY device starting with .ptyq0. If a PTY is already in use, the open call will return an error (the kernel uses the EXCL flag internally). When an open succeeds, the master then has exclusive access to that PTY. At this point, the master opens the corresponding TTY file (.ttyq0 — .ttyqf), or the slave device. It then forks off a process, which sets redirection up in the normal fashion and then exec's the program to run on the PTY.

The following code fragment is taken from the source code for the Graphical Shell Interface (GSI) NDA. `initPipe` scans the PTY devices, looking for a free one as discussed above. Note that the master side of a PTY does *not* have (by default) a terminal interface; it is a very raw device, with only a few **ioctl**'s to be able to send signals and handle other such low-level tasks.

```
char buffer[1024];
int ptyno, master;

int
initPipe(void)
{
    int cl[2];
    struct sgttyb sb;
    char *ptyname = ".ptyq0";

    unsigned i;

    /* We have to open the master first */
    for (i = 0; i<2; i++) {
        /* generate a PTY name from the index */
        ptyname[5] = intToHex(i);
        master = open(ptyname,O_RDWR);
        if (master > 0) {
            break; /* successful open */
        }
    }

    if (master < 1) {
        return -1;
    }

    ptyno = i;
    pid1 = fork(producer);
    return 0;
}
```

`producer()` sets up redirection for the shell, and also opens the slave side of the PTY. The slave processes must not have any access whatsoever to the master side of the PTY, so **close(0)** is used to close all open files (which includes, at this point, the master PTY file descriptor from initPipe). Note that as in many pipe applications, the file descriptor that will be assigned to a newly opened file is assumed, and that can be safely done in this case because it is clear that with no files open the next file descriptor will be 1.

```
/* the shell is executed here */
```

```
#pragma databank 1
void
producer(void)
{
    char *ptyname = ".ttyq0";

    /* we must not have access to ANY other ttys */

    close(0); /* close ALL open files */
    ptyname[5] = intToHex(ptyno);

    /* modify the tty slave name to correspond
     * to the master */

    slave = open(ptyname,O_RDWR); /* file descriptor 1 */
    dup(slave);                   /* fd 2 */
    dup(slave);                   /* fd 3 */

    /* Set up the TextTools redirection */
    SetOutputDevice(3,2l);
    SetErrorDevice(3,3l);
    SetInputDevice(3,1l);

    WriteCString("Welcome to GNO GSI\r\n");
    _execve(":bin:gsh","gsh -f");

    /* If we get here, we were unable to run
     * the shell.
     *
     * GDR note: printf should not be used here,
     * since we're in the child process */
    printf("Could not locate :bin:gsh : %d", errno);
}
#pragma databank 0
```

`consume()` is called as part of GSI's event loop. It simply checks to see if there is any data for the master by using the FIONREAD ioctl, one of the few ioctl's supported by the master side. See PTY(4) for details. Any data that is available is sent to the window via a routine toOut, which inserts the new data into a TextEdit record.

```
void
consume(CtlRecHndl teH)
{
```

```
    char ch;
    int fio, fio1, i;

    ioctl(master,FIONREAD,&fio);
    if (fio) {
        if (fio > 256) {
            fio = 256;
        }
        fio1 = read(master,buffer,fio);
        buffer[fio] = 0;
        toOut(buffer,fio,teH);
        updateWind1(fio,fio1);
    }
}
```

When the user types a key, the keypress is sent to the slave by simply writing the data with a write call.

```
void
writedata(char k)
{
    write(master, &k, 1);
}
```

When the user is done with the window and closes it, GSI closes the master end of the PTY.

```
void
closePipe(void)
{
    int cl[2];
    close(master);
}
```

When this is done, the slave process receives a SIGHUP signal, to indicate that the connection was lost. Since the standard behavior of SIGHUP is to terminate the process, the slave dies and either the slave or the kernel closes the slave end. At this point, the PTY is available for re-use by another application.

As you can see, PTYs are very simple to program and use. The simplicity can be misleading, for PTYs are a very powerful method of IPC. As another example of the use of PTYs, we point out that PTYs can be used to drive programs with 'scripts'. These scripts are a series of 'wait-for' and 'print' operations, much like auto-logon macros in communications programs such as ProTERM. Script-driving a program can be used to automate testing or use of an application.

PTYs can be used to test software that would normally work over a regular terminal (such as a modem). Since PTYs are identical (to the slave) to terminals, the application being tested doesn't know the difference. What this means to the programmer is incredible power and flexibility in testing the application. For example, a communications program could be nearly completely tested without ever dialing to another computer with a modem!

There are so many applications of PTYs that to attempt to discuss them all here would be impossible; as PTYs are discovered by more GNO/ME programmers we expect that more useful PTY applications will become available.

6.7 Deadlock

With interprocess communication comes the problem of *deadlock*. If a situation arises where two or more processes are all waiting for an signal from one of the other waiting processes, the processes are said to be deadlocked.

The best way to explain deadlock is to give an example. Suppose that two processes are connected with two pipes so that they can communicate bidirectionally. Also suppose that each of the pipes are full, and that when each process writes into one of the pipes they are blocked. Both processes are blocked waiting for the other to unblock them.

There is no way for the operating system to detect every conceivable deadlock condition without expending large amounts of CPU time. Thus, the only way to recover from a deadlock is to kill the processes in question. Responsibility for preventing deadlock situations is placed on the programmer. Fortunately, situations where deadlock can occur are infrequent; however, you should keep an eye out for them and try to work around them when they do occur.

Appendix A

Making System Calls

The GNO Kernel is accessed through system calls. The actual procedure is very simple from C: simply #include the appropriate header file as noted in the synopsis of the call's manual page, and call it as you would any other C function. From assembly language the procedure is no more difficult, using the advanced macros provided for the APW and ORCA assemblers. Make sure, however, that you have defined a word variable **errno**. Lowercase is important, use the 'case on' and 'case off' directives to ensure that the definition of **errno** is case-sensitive. The system call interface libraries store any error codes returned by the kernel in this variable.

If you are going to be accessing the kernel from a language other than those for which interfaces are provided, then the following information is for you.

A.1 System Call Interface

The system calls are implemented as a user toolset, tool number 3. These tools are called the same way regular system tools (such as QuickDraw) are called, except that you must `JSL` to **$E10008** instead of to **$E10000** (or to **$E1000C** instead of to **$E10004** for the alternate entry point). The function numbers for the currently defined tools are as follows:

getpid *	$0903	kill	$0A03
fork	$0B03	swait	$0D03
ssignal	$0E03	screate	$0F03
sdelete	$1003	kvm_open	$1103
kvm_close	$1203	kvm_getproc	$1303
kvm_nextproc	$1403	kvm_setproc	$1503
signal	$1603	wait	$1703
tcnewpgrp	$1803	settpgrp	$1903
tctpgrp	$1A03	sigsetmask	$1B03
sigblock	$1C03	execve	$1D03
alarm	$1E03	setdebug *	$1F03
setsystemvector *	$2003	sigpause	$2103
dup	$2203	dup2	$2303
pipe	$2403	getpgrp	$2503
ioctl	$2603	stat	$2703
fstat	$2803	lstat	$2903
getuid	$2A03	getgid	$2B03
geteuid	$2C03	getegid	$2D03
setuid	$2E03	setgid	$2F03
procsend	$3003	procreceive	$3103
procrecvclr	$3203	procrecvtim	$3303
setpgrp	$3403	times	$3503
pcreate	$3603	psend	$3703
preceive	$3803	pdelete	$3903
preset	$3A03	pbind	$3B03
pgetport	$3C03	pgetcount	$3D03
scount	$3E03	fork2	$3F03
getppid	$4003	SetGNOQuitRec	$4103
alarm10	$4203		

The following system calls are new to GNO v2.0.6:

select	$4303	InstallNetDriver	$4403
socket	$4503	bind	$4603
connect	$4703	listen	$4803
accept	$4903	recvfrom	$4A03
sendto	$4B03	recv	$4C03
send	$4D03	getpeername	$4E03
getsockname	$4F03	getsockopt	$5003
setsockopt	$5103	shutdown	$5203
setreuid	$5303	setregid	$5403

Parameters should be pushed onto the stack in the same order as defined by the C prototypes outlines in the synopsis section of the manual pages; that is, left-to-right. In addition to those parameters, all of the functions (except those denoted by a *****) take an integer pointer parameter **errno**. This is a pointer to a word value which will contain the **errno** code returned by the function if an error occurs, and should be pushed onto the stack after all the other parameters. The calls do not clear this code to 0 if no error occurs; thus, you must check the return value of the function to see if an error occurred, and then check errno to get the actual error code.

Do not forget to also push space on the stack (before the parameters) for the call to store its return value.

These low-level system call interfaces are not to be used in general programming. It is assumed the programmer will use the libraries provided, or use this information to create a new library. The system call interface is subject to change without notice; any changes will, of course, be documented in future versions of GNO/ME.

A.2 System Call Error Codes

The following codes are taken from `<sys/errno.h>`. The codes up to EPERM are the same values as those defined by ORCA/C for compatibility reasons. Error conditions are usually reported by system calls by returning a -1 (word) or NULL (long) value. Which error codes can be expected from a particular call are detailed in the errors section in the appropriate manual page.

EDOM Domain error. Basically an undefined error code.

ERANGE Range error. A value passed to a system call was too large, too small, or illegal.

ENOMEM Not enough memory. The kernel could not allocate enough memory to complete the requested operation.

ENOENT No such file or directory. The file specified could not be found.

EIO I/O error. An error occurred trying to perform an I/O operation, such as that caused by bad media. It also refers to a disk error not covered by the other errno codes.

EINVAL Invalid argument. An argument to a system call was invalid in some way.

EBADF Bad file descriptor. The file descriptor passed to the kernel does not represent an open file.

EMFILE Too many files are open. The kernel cannot open any more files for this process; it's open file table is full. Close some other open files and retry the operation.

EACCESS Access bits prevent the operation. One of the access bit settings (delete, rename, read, write) associated with the file does not allow the requested operation.

EEXIST The file exists. An attempt to create a new file with the same name as an existing file results in this error.

ENOSPC No space on device. There is not enough room on the requested device to complete the operation. This is usually indicative of a full disk.

EPERM Not owner. Not yet used in GNO.

ESRCH No such process. The process ID specified does not refer to an active process. Possibly the process terminated earlier.

EINTR Interrupted system call. Certain system calls can be interrupted by signals. In cases where the user has specified that those calls not be automatically restarted, the call will return this error.

E2BIG Arg list too long. Too many arguments were specified in an **_execve**(2) call.

ENOEXEC Exec format error. The file specified is not in an executable format (OMF load file).

ECHILD No children. This error is returned by **wait**(2) when there are no child processes left running.

EAGAIN No more processes. The process table is full, the **fork**(2) cannot complete.

ENOTDIR Not a directory. One of the elements in a pathname refers to a file which is not a directory.

ENOTTY Not a terminal. The file descriptor passed to an **ioctl**(2) or job control call does not refer to a terminal file.

EPIPE Broken pipe. If a process attempts to write on a pipe with no readers, and has blocked or ignored SIGPIPE, this error is returned by the write operation.

ESPIPE Illegal seek. Similar to ENOTBLK, but specific for pipes.

ENOTBLK Not a block device. An attempt to perform an operation on a character device that only makes sense on a block device.

A.3 System Panics

In most cases, if the kernel detects an error in operation an appropriate error code is returned by the function in question (GS/OS calls, ToolBox calls, or system calls as described above). However, there are rare circumstances where the kernel detects what should be an impossible condition. This can happen due to bugs in the kernel, because the kernel was overwritten by a buggy program, or for any number of other reasons.

When the kernel does come across such an error, system operation cannot continue and what ensues is called a *system panic*. Panics are very easily noticed-the kernel will print an error message on the screen and ensure that the text screen is visible, turning off any graphics mode if necessary. The kernel then sets the text and background colors to red on white - a very noticeable condition. At that point, the kernel turns off context switching to prevent any background process or other interrupt driven code from further confusing the system. This is done mainly to prevent damage to disk directory structures by a bad system.

When a system panic does occur, the only thing you can do is reboot your system. If you can reliably reproduce a system panic, please record the panic message and the sequence of events necessary to evoke the panic and report the information to Procyon, Inc.

Appendix B

Miscellaneous Programming Issues

B.1 Option Arguments

The Free Software Foundation (also known as the FSF), invented user friendly long format option arguments, and defined the "+¿" character for interpretation that a long format follows. This interpretation is generally followed in the UNIX community. There are two files which will assist you in programming GNO/ME utilities with both short and long format options, `<getopt.h>` for short options, and `<getopt1.h>` for long options.

B.2 Pathname Expansion

Those of you familiar with programming in the ORCA environment should be familiar with the shell calls `InitWildcard` and `NextWildcard`. These shell calls, while supported by **gsh**, are no longer necessary. All shell utilities that work with multiple filenames do not need to provide support for file globbing, as this is taken care of transparently to the command.

Appendix C

Glossary

Asynchronous An event that may take place at any time. See synchronous.

BASIC Beginners All-purpose Symbolic Instruction Code. A simple computer language.

Blocked Refers to a process waiting for some event to occur. Processes can block on terminal I/O, signals, and other IPC and I/O functions.

Console The terminal which represents the IIGS's keyboard and monitor.

Context The attributes which define the state of a process. This includes the program counter, stack pointer, and other machine registers (both CPU and other computer hardware).

Controlling terminal The terminal which "controls" a process or process group; processes can receive keyboard signals (such as SIGTSTP, or ˆZ) only from their controlling terminal.

Critical Section A piece of code inside which only one process at a time may be allowed to execute. Critical sections are usually protected by semaphores.

Daemon A process that runs in the background and waits to act on an asynchronous event. These can be anything: waiting for a caller on a modem, waiting for spooled files to print, etc. Daemons are usually started at boot time by the **initd**(8) process.

Deadlock A situation where two or more communicating processes are blocked, waiting on each other. See Chapter 5, "Deadlock".

Errno A variable which holds a descriptive numeric error code, returned from C libraries and system calls.

Foobar, foo, bar Foobar derives from an old military acronym FUBAR. In it's politest interpretation it stands for Fouled Up Beyond All Recognition. Computer scientists borrowed the term and created foobar. When a name for an object in a code fragment is needed but the name itself is not important, foo and bar are first choice among computing science types. They should not be used in production code.

Executable A program as it resides on disk. Executables can be compiled or assembled programs, or shell scripts. Executables are run by typing their name on the shell's command line and frequently take paramters to determine what data they operate on and particulars of how they do it.

GNO/ME GNO Multitasking Environment. The complete package including the GNO kernel and the GNO Shell.

GNO Kernel Heart of GNO/ME. Executes processes when asked by the GNO Shell.

GNO Shell Provides an interface between the user and the GNO kernel.

gsh GNO Implementation of a UNIX-like shell.

GS/OS A 16 bit Operating System for the Apple IIgs.

IPC "Inter-Process Communication". Any method by which processes can pass information to other processes.

Job A set of related processes. Jobs are generally composed of processes with a common parent and the same controlling terminal.

Manpage Refers to the system call and utility documentation provided with GNO. Manpages exist on disk as either **nroff**(1) or **aroff**(1) source. They can also be preformatted by **catman**(1). They can be viewed by various utilites on a variety of output devices.

Master Refers to the .PTYxx side of a pseudo-terminal, and also the process controlling that device. The master is usually responsible for setting up the PTY and running a process on it.

Message A 32-bit value that is passed via the Messages IPC mechanism to another process.

Mutex Short for mutual exclusion, a term that refers to protecting a critical section.

Panic An unrecoverable kernel error, usually indicating that an internal data structure has become corrupted.

Parent When talking about a process, the parent of a process is the one that spawned it; i.e., made the **fork**(2) system call.

Pipe A unidirectional IPC mechanism. Pipes transmit binary 8-bit data.

Pipeline Two or more processes connected by pipes.

Port A flow-controlled IPC mechanism that can pass longwords of data.

Process A program in execution.

Process Group An identifying code for a job. Process groups are also assigned to TTYs, which allows the TTY to differentiate background jobs from foreground jobs when sending interrupt signals.

Pseudo-terminal A bidirectional communications channel, normally used in windowing systems or for advanced control and testing applications.

PTY See 'pseudo-terminal'.

Semaphore A data object used to synchronize concurrent processes.

Sequentialization The task of ensuring that critical sections are only executed by one concurrent process at a time.

Signal A software interrupt and IPC mechanism.

Slave 1. A good term to describe the relationship of Joe Citizen to the IRS. 2. The .TTYxx side of a pseudo-terminal; the slave is usually an application program of some kind, like a shell.

Suspended Refers to a process whose execution has been stopped.

Synchronous An event that takes place at a predetermined time or sequence of times. Also used to indicate the act of waiting for an event to happen. See asynchronous.

Terminal Any device that looks like a terminal; this includes pseudo-ttys. By definition, a terminal supports all of the **tty**(4) ioctl calls.

Tty Short for Teletype. TTY is an anachronistic term; in modern usage it is taken to mean "terminal".

UNIX Popular operating system which has growing use in education and business. One of the first operating systems to support multitasking.

Index