– CHAPTER 2 –

GEMDOS

Overview

GEMDOS contains functions which comprise the highest level of **TOS**. In many cases, **GEMDOS** devolves into **BIOS** calls which handle lower level device access. **GEMDOS** is responsible for file, device, process, and high-level input/output management. The current revision number of **GEMDOS** is obtained by calling **Sversion()**. You should note that the **GEMDOS** version number is independent of the **TOS** version number and you should not count on any particular version of **GEMDOS** being present based on the **TOS** version present.

Much of **GEMDOS** closely resembles its CPM 68k and MS-DOS heritage. In fact, the file system and function calls are mostly compatible with MS-DOS. MS-DOS format floppy disks are readable by an Atari computer and vice-versa.

For the creation of **MultiTOS**, **GEMDOS** was merged with the **MiNT** operating environment which derives many of its calls from the UNIX operating system.

The TOS File System

GEMDOS is responsible for interaction between applications and file-based devices. Floppy and hard disk drives as well as CD-ROM, WORM, and Magneto-Optical drives are all accessed using **GEMDOS** calls.

Prior to the advent of **MultiTOS**, Atari programmers were limited to the **TOS** file system for file storage and manipulation. With the introduction of **MultiTOS**, it is now possible for developers to create custom file systems so that almost any conceivable disk format becomes accessible.

As a default, **MultiTOS** will manage files between the **TOS** file system and alternative file systems to maintain backward compatibility. Applications which wish to support extra file system features may do so. The **Pdomain()** call may be used to instruct **MultiTOS** to stop performing translations on filenames, etc. Other calls such as **Dpathconf()** can be used to determine the requirements of a particular file system.

The explanation of the file system contained herein will limit itself to the **TOS** file system.

Drive Identifiers

Each drive connected to an Atari system is given a unique alphabetic identifier which is used to identify it. Drive ‘A’ is reserved for the first available floppy disk drive (usually internal) and drive ‘B’ for the second floppy disk drive. If only one floppy drive exists, two letters will still be reserved and **GEMDOS** will treat drive ‘B’ as a pseudo-drive and request disk swaps as necessary. This feature is automatically handled by **GEMDOS** and is transparent to the application.

Drives ‘C’ through ‘P’ are available for use by hard disk drives. One letter is assigned per hard drive partition so a multiple-partition drive will be assigned multiple letters. **MultiTOS** extends drive letter assignments to ‘Z’ drive. Drive ‘U’ is a special drive reserved for **MultiTOS** and is unavailable for assignment.

The amount of free storage space remaining on a drive along with a drive’s basic configuration can be determined using the **Dfree()** call.

GEMDOS Filenames

Under **GEMDOS**, each file located on a device is given a filename upon its creation which serves to provide identification for the file. The filename has two parts consisting of a name from one to eight characters long and an optional file extension of up to three characters long. If a file extension exists, the two components are separated by a period. The extension should serve to identify the format of the data whereas the name itself should identify the data itself.

Filenames may be changed after creation with the function **Frename()**; however, under no circumstances may two files with the same filename reside in the same directory.

All **GEMDOS** functions ignore the alphabetic case of file and pathnames. The following characters are legal filename characters:

Legal **GEMDOS** Filename Characters
A-Z, a-z, 0-9
! @ # $ % ^ & ( )
+ - = ~ ` ; ‘ “ ,
< > | [ ] ( ) _

GEMDOS Directories

To further organize data, **GEMDOS** provides file directories (or folders). Each drive may contain any number of directories which, in turn, may contain files and additional directories. This organization creates a tree-like structure of files and folders. A file’s location in this tree is called the path.

Directory names follow the same format as **GEMDOS** filenames with a maximum filename length of 8 characters and an optional 3 character extension. The first directory of a disk which contains all subdirectories and files is called the root directory.

The **Dcreate()** and **Ddelete()** system calls are used to create and delete subdirectories.

Two special, system-created subdirectories are present in some directories. A subdirectory with the name ‘..’ (two periods) refers to the parent of the current directory. The ‘..’ subdirectory is present in every subdirectory.

A subdirectory with the name ‘.’ refers to the current directory. There is a ‘.’ subdirectory in every directory.

GEMDOS Path Specifications

To access a file, a complete path specification must be composed of the drive letter, directory name(s), and filename. A file named 'TEST.PRG' located in the 'SYSTEM' directory on drive 'C' would have a path specification like the following:

```
C:\SYSTEM\TEST.PRG
```

The drive letter is the first character followed by a colon. Each directory and subdirectory is surrounded by backslashes. If 'TEST.PRG' were located in the root directory of 'C' the path specification would be:

```
C:\TEST.PRG
```

The drive letter and colon may be omitted causing **GEMDOS** to reference the default drive as follows:

```
\TEST.PRG
```

A filename by itself will be treated as the file in the default directory and drive. The current **GEMDOS** directory and drive may be found with the functions **Dgetpath()** and **Dgetdrv()** respectively. They may be changed with the functions **Dsetpath()** and **Dsetdrv()**.

Wildcards

The **GEMDOS** functions **Fsfirst()** and **Fsnext()** are used together to enumerate files of a given path specification. These two functions allow the use of wildcard characters to expand their search parameters.

The '?' character is used to represent exactly one unknown character. The '*' character is used to represent any number of unknown characters. The following table gives some examples of the uses of these characters.

Filename	Found	Not Found
.	All files	None
*.GEM	TEST.GEM ATARI.GEM	TEST.G ATARI.IMG
A?ARI.?	ATARI.O ADARI.C	ADARI.IMG ATARI.GEM
ATARI.???	ATARI.GEM ATARI.IMG	ATARI.O ATARI.C

Disk Transfer Address (DTA)

When using **Fsfirst()** and **Fsnext()** to build a list of files, **TOS** uses the Disk Transfer Address (DTA) to store information about each file found. The format for the DTA structure is as follows:

```
typedef struct
{
    BYTE     d_reserved[21];  /* Reserved - Do Not Change */
    BYTE     d_attrib;        /* GEMDOS File Attributes */
    UWORD    d_time;          /* GEMDOS Time */
    UWORD    d_date;          /* GEMDOS Date */
    LONG     d_length;        /* File Length */
    char     d_fname[14];     /* Filename */
} DTA;
```

When a process is started, its DTA is located at a point where it could overlay potentially important system structures. To avoid overwriting memory a process wishing to use **Fsfirst()** and **Fsnext()** should allocate space for a new DTA and use **Fsetdta()** to instruct the OS to use it. The original location of the DTA should be saved first, however. Its location can be found with the call **Fgetdta()**. At the completion of the operation the old address should be replaced with **Fsetdta()**.

File Attributes

Every **TOS** file contains several attributes which define it more specifically. File attributes are specified when a file is created with **Fcreate()** and can be altered later with **Fattrib()**.

The ‘read-only’ attribute bit is set to prevent modification of a file. This bit should be set at the user’s discretion and not cleared unless the user explicitly requests it.

If the ‘hidden’ attribute is set, the file will not be listed by the desktop or file selector. These files may still be accessed in a normal manner but will not be present in an **Fsfirst()** or **Fsnext()** search unless the correct **Fsfirst()** bits are present.

The ‘system’ attribute is unused by **TOS** but remains for MS-DOS compatibility.

The ‘volume label’ attribute should be present on a maximum of one file per drive. The file which has it set should be in the root directory and have a length of 0. The filename indicates the volume name of the drive.

The ‘archive’ attribute is a special bit managed by **TOS** which indicates whether a file has been written to since it was last backed up. Any time a **Fcreate()** call creates a file or **Fwrite()** is used on a file, the Archive bit is set. This enables file backup applications to know which files have been modified since the last backup. They are responsible for clearing this bit when backing up the file.

File Time/Date Stamp

When a file is first created a special field in its directory entry is updated to contain the date and time of creation. **Fdatime()** can be used to access or modify this information as necessary.

File Maintenance

New files should be created with **Fcreate()**. When a file is successfully created a positive file handle is returned by the call. That handle is what is used to identify the file for all future operations until the file is closed. After a file is closed its handle is invalidated.

Files which are already in existence should be opened with **Fopen()**. As with **Fcreate()**, this call returns a positive file handle upon success which is used in all subsequent **GEMDOS** calls to reference the file.

Each process is allocated an OS dependent number of file handles. If an application attempts to open more files than this limit allows, the open or create call will fail with an appropriate error code. File handles may be returned to the system by closing the open file with **Fclose()**.

Fopen() may be used in read, write, or read/write mode. In read mode, **Fread()** may be used to access existing file contents. In write mode, any original information in the file is not cleared but the data may be overwritten with **Fwrite()**. In read/write mode, either call may be used interchangeably.

Every file has an associated file position pointer. This pointer is used to determine the location for the next read or write operation. This pointer is expressed as a positive offset from the beginning of the file (position 0) which is set upon first creating or opening a file. The pointer may be read or modified with the function **Fseek()**.

Existing files may be deleted with the **GEMDOS** call **Fdelete()**.

File/Record Locking

File and record locking allow portions or all of a file to be locked against access from another computer over a network or another process in the same system.

All versions of **TOS** have the ability to support file and record locking but not all have the feature installed. If the '_FLK' cookie is present in the system cookie jar then the **Flock()** call is present. This call is used to create locks on individual sections (usually records) in a file.

Locking a file in use, when possible, is recommended to prevent other processes from modifying the file at the same time.

Special File Handles

Several special file handles are available for access through the standard **Fopen()/Fread()/Fwrite()** calls. They are as follows:

Name	Handle	Filename	Device
GSH_BIOSCON	0xFFFF	CON:	Console (screen). Special characters such as the carriage return, etc. are interpreted.
GSH_BIOSAUX	0xFFFE	AUX:	Modem (serial port). This is the ST-compatible port for machines with more than one.
GSH_BIOSPRN	0xFFFD	PRN:	Printer (attached to the Centronics Parallel port).
GSH_BIOSMIDIIN	0xFFFC		Midi In
GSH_BIOSMIDIOUT	0xFFFB		Midi Out

GSH_CONIN	0x00	—	Standard Input (usually directed to **GSH_BIOSCON**)
GSH_CONOUT	0x01	—	Standard Output (usually directed to **GSH_BIOSCON**)
GSH_AUX	0x02	—	Auxillary (usually directed to **GSH_BIOSAUX**)
GSH_PRN	0x03	—	Printer (usually directed to **GSH_BIOSPRN**)
None	0x04	—	Unused
None	0x05	—	Unused
None	0x06 and up	User-Specified	User Process File Handles

These files may be treated like any other **GEMDOS** files for input/output and locking. Access to these devices is also provided with **GEMDOS** character calls (see later in this chapter).

File Redirection

Input and output to a file may be redirected to an alternate file handle. For instance you may redirect the console output of a **TOS** process to the printer.

File redirection is handled by the use of the **Fforce()** call. Generally you will want to make a copy of the file handle with **Fdup()** prior to redirecting the file so that it may be restored to normal operation when complete.

Memory Management

Atari systems support two kinds of memory. Standard RAM (sometimes referred to as 'ST RAM') is general purpose RAM that can be used for any purpose including video and DMA. Current Atari architecture limits the amount of standard RAM a system may have to 14MB.

Alternative RAM (sometimes referred to as 'TT RAM') can be accessed faster than standard RAM but is not suitable for video memory or DMA transfers.

The **Malloc()** and **Mxalloc()** calls allocate memory blocks from the system heap. **Malloc()** chooses the type of memory it allocates based on fields in the program header (see later in this chapter). **Mxalloc()** allows the application to choose the memory type at run-time.

MultiTOS uses memory protection to prevent an errant process from damaging another. It is possible with **Mxalloc()** to dynamically set the protection level of an allocated block.

Memory allocated with either **Malloc()** or **Mxalloc()** may be returned to the system with **Mfree()**. Memory allocated by a process is automatically freed when the process calls **Pterm()**.

GEMDOS Processes

The **GEMDOS** call **Pexec()** is responsible for launching executable files. The process which calls **Pexec()** is called the parent and the file launched becomes the child. Each process may have more than one child process. Depending on the mode used with **Pexec()**, the child may share data and address space and/or run concurrently (under **MultiTOS**) with the parent. **GEMDOS** executable files (**GEM** and **TOS** applications or desk accessories) contain the following file header:

Name	Offset	Contents
PRG_magic	0x00	This **WORD** contains the magic value (0x601A).
PRG_tsize	0x02	This **LONG** contains the size of the TEXT segment in bytes.
PRG_dsize	0x06	This **LONG** contains the size of the DATA segment in bytes.
PRG_bsize	0x0A	This **LONG** contains the size of the BSS segment in bytes.
PRG_ssize	0x0E	This **LONG** contains the size of the symbol table in bytes.
PRG_res1	0x12	This **LONG** is unused and is currently reserved.
PRGFLAGS	0x16	This **LONG** contains flags which define certain process characteristics (as defined below).
ABSFLAG	0x1A	This **WORD** flag should be non-zero to indicate that the program has no fixups or 0 to indicate it does. Since some versions of **TOS** handle files with this value being non-zero incorrectly, it is better to represent a program having no fixups with 0 here and placing a 0 longword as the fixup offset.
Text Segment	0x1C	This area contains the program's TEXT segment. A process is started by JMP'ing to **BYTE** 0 of this segment with the address of your processes basepage at 4(sp).
Data Segment	***PRG_tsize*** + 0x1C	This area contains the program's DATA segment (if one exists).
Symbol Segment	***PRG_tsize*** + ***PRG_dsize*** + 0x1C	This area contains the program's symbol table (if there is one). The symbol table area is used differently by different compiler vendors. Consult them for the format.
Fixup Offset	***PRG_tsize*** + ***PRG_dsize*** + ***PRG_ssize*** + 0x1C	This **LONG** indicates the first location in the executable (as an offset from the beginning) containing a longword needing a fixup. A 0 means there are no fixups.

Fixup Information	***PRG_tsize*** + ***PRG_dsize*** + ***PRG_ssize*** + 0x20	This area contains a stream of **BYTE**s containing fixup information. Each byte has a significance as follows: **Value** — **Meaning** 0 — End of list. 1 — Advance 254 bytes. 2-254 (even) — Advance this many bytes and fixup the longword there.

PRGFLAGS is a bit field defined as follows:

Definition	Bit(s)	Meaning
PF_FASTLOAD	0	If set, clear only the BSS area on program load, otherwise clear the entire heap.
PF_TTRAMLOAD	1	If set, the program may be loaded into alternative RAM, otherwise it must be loaded into standard RAM.
PF_TTRAMMEM	2	If set, the program's **Malloc()** requests may be satisfied from alternative RAM, otherwise they must be satisfied from standard RAM.
—	3	Currently unused.
See left.	4 & 5	If these bits are set to 0 (**PF_PRIVATE**), the processes' entire memory space will be considered private (when memory protection is enabled). If these bits are set to 1 (**PF_GLOBAL**), the processes' entire memory space will be readable and writable by any process (i.e. global). If these bits are set to 2 (**PF_SUPERVISOR**), the processes' entire memory space will only be readable and writable by itself and any other process in supervisor mode. If these bits are set to 3 (**PF_READABLE**), the processes' entire memory space will be readable by any application but only writable by itself.
—	6-15	Currently unused.

When a process is started by **GEMDOS**, it allocates all remaining memory, loads the process into that memory, and JMP's to the first byte of the application's TEXT segment with the address of the program's basepage at 4(sp). An application should use the basepage information to decide upon the amount of memory it actually needs and **Mshrink()** to return the rest to the system. The exception to this is that desk accessories are only given as much space as they need (as indicated by their program header) and their stack space is pre-assigned.

The following code illustrates the proper way to release system memory and allocate your stack (most ‘C’ startup routines do this for you):

```
stacksize    =   $2000            ; 8K

                 .text

_start:
        move.l   4(sp),a0         ; Obtain pointer to basepage
        move.l   a0,basepage      ; Save a copy
        move.l   $18(a0),a1       ; BSS Base address
        adda.l   $1C(a0),a1       ; Add BSS size
        adda.l   #stacksize,a1        ; Add stack size

        move.l   a1,sp            ; Move your stack pointer to
                                  ; your new stack.

        suba.l   basepage,a1      ; TPA size
        move.l   a1,-(sp)
        move.l   basepage,-(sp)
        clr.w    -(sp)
        move.w   #$4a,-(sp)       ; Mshrink()
        trap     #1
        lea      12(sp),sp        ; Fix up stack
                                  ; and fall through to main
_main:
       ...

                 .bss

basepage:        ds.l    1

                 .end
```

The **GEMDOS BASEPAGE** structure has the following members:

Name	Offset	Meaning
p_lowtpa	0x00	This **LONG** contains a pointer to the Transient Program Area (TPA).
p_hitpa	0x04	This **LONG** contains a pointer to the top of the TPA + 1.
p_tbase	0x08	This **LONG** contains a pointer to the base of the text segment
p_tlen	0x0C	This **LONG** contains the length of the text segment.
p_dbase	0x10	This **LONG** contains a pointer to the base of the data segment.
p_dlen	0x14	This **LONG** contains the length of the data segment.
p_bbase	0x18	This **LONG** contains a pointer to the base of the BSS segment.
p_blen	0x1C	This **LONG** contains the length of the BSS segment.
p_dta	0x20	This **LONG** contains a pointer to the processes’ DTA.

p_parent	0x24	This **LONG** contains a pointer to the processes' parent's basepage.
p_reserved	0x28	This **LONG** is currently unused and is reserved.
p_env	0x2C	This **LONG** contains a pointer to the processes' environment string.
p_undef	0x30	This area contains 80 unused, reserved bytes.
p_cmdlin	0x80	This area contains a copy of the 128 byte command line image.

Processes terminate themselves with either **Pterm0()**, **Pterm()**, or **Ptermres()**. **Ptermres()** allows a segment of a file to remain behind in memory after the file itself terminates (this is mainly useful for TSR utilities).

The Atari Extended Argument Specification

When a process calls **Pexec()** to launch a child, the child may receive a command line up to 125 characters in length. The command line does not normally contain information about the process itself (what goes in *argv*[0] in 'C'). The Atari Extended Argument Specification (ARGV) allows command lines of any length and correctly passes the child the command that started it. The ARGV specification works by passing the command tail in the child's environment rather than in the command line buffer.

Both the parent and child have responsibilities when wanting to correctly handle the ARGV specification. If a process wishes to launch a child with a command line of greater than 125 characters it should follow these steps:

1. Allocate a block of memory large enough to hold the existing environment, the string 'ARGV=' and its terminating **NULL**, a string containing the complete path and filename of the child process and its terminating **NULL**, and a string containing the child's command line arguments and its terminating **NULL**.
2. Next, copy these elements into the reserved block in the order given above.
3. Finally, call **Pexec()** with this environment string and a command line containing a length byte of 127 and the first 125 characters of the command line with a terminating **NULL**.

For a child to correctly establish that a parent process is using ARGV it should check for the length byte of 127 and the ARGV variable. Some parents may assign a value to ARGV (found between the 'ARGV=' and the terminating **NULL** byte). It should be skipped over and ignored. If a child detects that its parent is using ARGV, it then has the responsibility of breaking down the environment into its components to properly obtain its command line elements.

It should be noted that many compilers include ARGV parsing in their basic startup stubs. In addition, applications running under **MultiTOS** should use the **AES** call **shel_write()** as it automatically creates an ARGV environment string.

GEMDOS Vectors

GEMDOS reserves eight system interrupt vectors (of which only three are used) for various system housekeeping. The **BIOS** function **Setexc()** should be used to redirect these vectors when necessary. The **GEMDOS** vectors are as follows:

Name	Setexc() Vector Number	Usage
VEC_TIMER	0x0100	Timer Tick Vector: This vector is jumped through 50 times per second to maintain the time-of-day clock and accomplish other system housekeeping. A process intercepting this vector does not have to preserve any registers but should jump through the old vector when completed. Heavy use of this vector can severly affect system performance. Return from this handler with RTS.
VEC_CRITICALERR	0x0101	Critical Error Handler: This vector is used by the **BIOS** to service critical alerts (an **Rwabs()** disk error or media change request). When called, the **WORD** at 4(sp) is a **GEMDOS** error number. On return, D0.L should contain 0x0001000 to retry the operation, 0 to ignore the error, or 0xFFFFFFxx to return an error code (xx). D3-D7 and A3-A6 must be preserved by the handler. Return from this handler with RTS.
VEC_PROCTERM	0x0102	Process Terminate Vector: This vector is called just prior to the termination of a process ended with CTRL-C. Return from this handler with RTS.
—	0x103-0x0107	Currently unused.

MiNT

MiNT is Now TOS (**MiNT**) is the extension to **GEMDOS** that allows **GEMDOS** to multitask under **MultiTOS**. **MiNT** also provides memory protection (on a 68030 or higher) to protect an errant process from disturbing another.

Processes

MiNT assigns each process a process identifier and a process priority value. The identifier is used to distinguish the process from others in the multitasking environment. **Pgetpid()** is used to obtain the **MiNT** ID of the process and **Pgetppid()** can be used to obtain the ID of the processes' parent.

MiNT also supports networking file systems that support the concept of user and process group control. The **Pgetpgrp()**, **Psetpgrp()**, **Pgetuid()**, **Psetuid()**, **Pgeteuid()**, and **Pseteuid()** get and set the process, user, and effective user ID for a process.

MiNT has complete control over the amount of time allocated to individual processes. It is possible, however, to set a process 'delta' value with **Pnice()** or **Prenice()** which will be used by **MiNT** to decide the amount of processor time a process will get per timeslice. **Syield()** can be used to surrender the remaining portion of a timeslice.

Information about a processes' resource usage can be obtained by calling **Prusage()**. These values can be modified with **Psetlimit()**. System configuration capabilities may be obtained with **Sysconf()**.

Each process can have a user-defined longword value assigned to itself with **Pusrval()**.

The functions **Pwait()**, **Pwait3()**, and **Pwaitpid()** attempt to determine the exit codes of stopped child processes.

Threads

It is possible under **MiNT** to split a single process into 'threads'. These threads continue execution independently as unique processes. The **Pfork()** and **Pvfork()** calls are used to split a process into threads.

The original process that calls **Pfork()** or **Pvfork()** is considered the parent and the newly created process is considered the child.

Child processes created with **Pfork()** share the TEXT segment of the parent, however they are given a copy of the DATA and BSS segments. Both the parent and child execute concurrently.

Child processes created with **Pvfork()** share the entire program code and data space including the processor stack. The parent process is suspended until the child exits or calls **Pexec()**'s mode 200.

Child processes started with either call may make **GEM** calls but a child process started with **Pfork()** must call **appl_init()** to force **GEM** to uniquely recognize it as an independent process. This is not necessary with **Pvfork()** because all program variables are shared.

The following is a simple example of using a thread in a **GEM** application:

```
VOID
UserSelectedPrint( VOID )
{
        /* Prevent the user from editing buffer being printed. */
        LockBufferFromEdits();

        if( Pfork() == 0)
        {
              /* Child enters here */

              appl_init();                  /* Required for GEM threads. */

              DisplayPrintingWindow();      /* Do our task. */
              PrintBuffer();

              /* Send an AES message to the parent telling it to unlock buffer. */
              SendCompletedMessageToParent();

              /* Cleanup and exit thread. */
              appl_exit();
```

```
             Pterm( 0 );
       }

       /* Parent returns and continues normal execution. */
}
```

File System Extensions

MiNT provides several new file and directory manipulation functions that work with **TOS** and other loadable file systems. The **Fcntl()** function performs a large number of file-based tasks many of which apply to special files like terminal emulators and ‘U:\’ files. **Fxattr()** is used to obtain a file’s extended attributes. Some extended attributes are not relevant to the **TOS** file system and will not return meaningful values (see the ***Function Reference*** for details).

Fgetchar() and **Fputchar()** can be used to get and put single characters to a file. **Finstat()** and **Foutstat()** are used to determine the input or output status of a file. **Fselect()** is used to select from a group of file handles those ready to be read from or written to (often used for pipes).

Flink(), **Fsymlink()**, and **Freadlink()** are used to create hard and symbolic links to another file. Links are not supported by all file systems (see the entries for these functions for more details).

Some file systems may support the concept of file ownership and access permissions (**TOS** does not). The **Fchown()** and **Fchmod()** calls are used to adjust the ownership flags and access permissions of a file. **Pumask()** can be used to set the minimum access permissions assigned to each subsequently created file.

Fmidipipe() is used to redirect the file handles used for MIDI input and output.

MiNT provides four new functions for directory enumeration (they provide similar functionality to **Fsfirst()** and **Fsnext()** with a slightly easier interface). **Dopendir()** is used to open a directory for enumeration. **Dreaddir()** steps through each entry in a directory. **Drewinddir()** resets the file pointer to the beginning of the directory. **Dclosedir()** closes a directory.

Dlock() allows disk-formatters and other utilities which require exclusive access to a drive the ability to lock a physical device from other processes.

Dgetcwd() allows a process to obtain the current **GEMDOS** working directory for any process in the system (including itself).

Dcntl() performs device and file-system specific operations (consult the ***Function Reference*** for more details).

Pseudo Drives

MiNT creates a pseudo drive ‘U:’ which provides access to device drivers, processes, and other system resources. In addition to creating a directory on drive U: for each system drive, **MiNT** may create any of the following directories at the ROOT of the drive:

Folder Name	Contents

\DEV	Loaded devices
\PIPE	System pipes
\PROC	System processes
\SHM	Shared memory blocks

Drive directories on ‘U:’ act as if they were accessed by their own drive letter. Folder ‘U:\C\’ contains the same files and folders as ‘C:\’.

The ‘U:\PROC’ Directory

Each system process has a file entry in the ‘U:\PROC’ directory. The filename given a process in this directory is the basename for the file (without extension) with an extension consisting of the **MiNT** process identifier. The MINIWIN.PRG application might have an entry named ‘MINIWIN.003’.

The file size listed corresponds to the amount of memory the process is using. The time and date stamp contains the length of time the process has been executing as if it were started on Jan. 1st, 1980 at midnight. The file attribute bits tell special information about a process as follows:

Name	Attribute Byte	Meaning
PROC_RUN	0x00	The process is currently running.
PROC_READY	0x01	The process is ready to run.
PROC_TSR	0x02	The process is a TSR.
PROC_WAITEVENT	0x20	The process is waiting for an event.
PROC_WAITIO	0x21	The process is waiting for I/O.
PROC_EXITED	0x22	The process has been exited but not yet released.
PROC_STOPPED	0x24	The process was stopped by a signal.

Loadable Devices

MiNT contains a number of built-in devices and also supports loadable device drivers. Current versions of **MiNT** may contain any of the following devices:

Device Filename	Device
CENTR	Centronics Parallel Port
MODEM1	Modem Port 1
MODEM2	Modem Port 2
SERIAL1	Serial Port 1
SERIAL2	Serial Port 2
MIDI	MIDI ports
PRN	PRN: device (usually the Centronics Parallel Port)
AUX	AUX: device (usually the RS232 Port)
CON	Current Terminal
TTY	Current Terminal (same as CON)
STDIN	Current File Handle 0 (standard input)
STDOUT	Current File Handle 1 (standard output)
STDERR	Current File Handle 2 (standard error)
CONSOLE	Physical Console (keyboard/screen)

MOUSE	Mouse (system use only)
NULL	NULL device
AES_BIOS	**AES BIOS** Device (system use only)
AES_MT	**AES** Multitasking Device (system use only)

Each of these devices is represented by a filename (as shown in the table above) in the 'U:\DEV\' directory. Using standard **GEMDOS** calls (ex: **Fread()** and **Fwrite()**) on these files yields the same results as accessing the device directly. New devices, including those directly accessible by the **BIOS**, may be added to the system with the **Dcntl()** call using a parameter of **DEV_INSTALL**, **DEV_NEWBIOS**, or **DEV_NEWTTY**. See the **Dcntl()** call for details.

MiNT versions 1.08 and above will automatically load device drivers with an extension of '.XDD' found in the root or '\MULTITOS' directory. '.XDD' files are special device driver executables which are responsible for installing one (or more) new devices. **MiNT** will load the file and JSR to the first instruction in the TEXT segment (no parameters are passed). The device driver executable should not attempt to **Mshrink()** or create a stack (one has already been created).

The '.XDD' may then either install its device itself with **Dcntl()** and return **DEV_SELFINST** (1L) in register D0 or return a pointer to a **DEVDRV** structure to have the **MiNT** kernel install it (the 'U:\DEV\' filename will be the same as the first eight characters of the '.XDD' file). If for some reason, the device can not be initialized, 0L should be returned in D0.

When creating a new **MiNT** device with **Dcntl(DEV_INSTALL**, *devname*, &*dev_descr* **)** the structure **dev_descr** contains a pointer to your **DEVDRV** structure defined as follows:

```
typedef struct devdrv
{
       LONG (*open)( FILEPTR *f );
       LONG (*write)( FILEPTR *f, char *buf, LONG bytes );
       LONG (*read)( FILEPTR *f, char *buf, LONG bytes );
       LONG (*lseek)( FILEPTR *f, LONG where, LONG whence );
       LONG (*ioctl)( FILEPTR *f, WORD mode, VOIDP buf );
       LONG (*datime)( FILEPTR *f, WORD *timeptr, WORD rwflag );
       LONG (*close)( FILEPTR *f, WORD pid );
       LONG (*select)( FILEPTR *f, LONG proc, WORD mode );
       LONG (*unselect)( FILEPTR *f, LONG proc, WORD mode );
       LONG reserved[3];
} DEVDRV;
```

Each of the assigned members of this structure should point to a valid routine that provides the named operation on the device. The routine must preserve registers D2-D7 and A2-A7 returning its completion code in D0. No operating system **TRAP**s should be called from within these routines, however, using the vector tables provided in the **kerinfo** structure returned from the **Dcntl()** call, **GEMDOS** and **BIOS** calls may be used. The specific function that each routine is responsible for is as follows:

Member	Meaning
open	This routine is called by the **MiNT** kernel after a **FILEPTR** structure has been created for a file determined to be associated with the device. The routine should perform whatever initialization is necessary and exit with a standard **GEMDOS** completion code. This routine is responsible for validating the sharing mode and other file flags to verify that the file may be legally opened and should respond with an appropriate error code if necessary.
write	This routine should write *bytes* number of **BYTE**s from *buf* to the file specified in **FILEPTR**. If the file pointer has the **O_APPEND** bit set, the kernel will perform an *lseek()* call to the end of the file prior to calling this function. If the *lseek()/write()* series of calls does not guarantee that data will be written at the end of a file associated with your device, this function must ensure that the data specified is actually written at the end of the file. This function should return with a standard **GEMDOS** error code or the actual number of **BYTE**s written to the file when complete.
read	This routine should read *bytes* number of **BYTE**s from the file specified in **FILEPTR** and place them in the buffer *buf*. This function should return with a standard **GEMDOS** error code or the actual number of bytes read by the routine.
lseek	This routine should move the file position pointer to the appropriate location in the file as specified by the parameter *where* in relation to the seek mode *whence*. Seek modes are the same as with **Fseek()**. The routine should return a **GEMDOS** error code or the absolute new position from the start of the file if successful.
ioctl	This routine is called from the system's perspective as **Fcntl()** and is used to perform file system/device specific functions. At the very least, your device should support **FIONREAD**, **FIONWRITE**, and the file/record locking modes of **Fcntl()**. The *arg* parameter of **Fcntl()** is passed as *buf*.
datime	This routine is used to read or modify the date/time attributes of a file. *timeptr* is a pointer to two **LONG**s containing the time and date of the file respectively. These **LONG**s should be used to set the file date and time if *rwflag* is non-zero or filled in with the file's creation date and time if *rwflag* is 0. This function should return with a standard **GEMDOS** error code or **E_OK** (0) if successful.
close	This routine is used by the kernel to close an open file. Be aware that if *f->links* is non-zero, additional processes still have valid handles to the file. If *f->links* is 0 then the file is really being closed. *pid* specifies the process closing the file and may not necessarily be the same as the process that opened it. Device drivers should set the **O_LOCK** bit on f->flag when the **F_SETLK** or **F_SETLKW** *ioctl()* call is made. This bit can be tested for when a file is closed and all locks on all files associated with the same physical file owned by process *pid* should be removed. If the file did not have any locks created on it by process *pid,* then no locks should be removed. This routine should return with a standard **GEMDOS** error code or **E_OK** (0) if successful.
select	This routine is called when a call to **Fselect()** names a file handled by this device. If *mode* is **O_RDONLY** then the select is for reading, otherwise, if *mode* is **O_WRONLY** then it is for writing. If the user **Fselect()**'s for both reading and writing then two calls to this function will be made. The routine should return 1L if the device is ready for reading or writing (as appropriate) or it should return 0L and arrange to 'wake up' process *proc* when I/O becomes possible. This is usually accomplished by calling the *wakeselect()* member function of the kernel structure. Note that the value in *proc* is not the same as a **PID** and is actually a pointer to a **PROC** structure private to the **MiNT** kernel.
unselect	This routine is called when a device waiting for I/O should no longer be waited for. The *mode* and

	proc parameters are the same as with **select()**. As with **select()**, if neither reading nor writing is to be waited for, two calls to this function will be made. This routine should return a standard **GEMDOS** error code or **E_OK** (0) if successful.

The **FILEPTR** structure pointed to by a parameter of each of the above calls is defined as follows:

```
typedef struct fileptr
{
       WORD            links;
       UWORD           flags;
       LONG            pos;
       LONG            devinfo;
       fcookie             fc;
       struct devdrv   *dev;
       struct fileptr  *next;
} FILEPTR;
```

The members of **FILEPTR** have significance as follows:

Member	Meaning
links	This member contains a value indicating the number of copies of this file descriptor currently in existence.
flags	This member contains a bit mask which indicates several attributes (logically OR'ed together) of the file as follows: **Name** / **Mask** / **Meaning** **O_RDONLY** 0x0000 File is read-only. **O_WRONLY** 0x0001 File is write-only. **O_RDWR** 0x0002 File may be read or written. **O_EXEC** 0x0003 File was opened to be executed. **O_APPEND** 0x0008 Writes start at the end of the file. **O_COMPAT** 0x0000 File-sharing compatibility mode. **O_DENYRW** 0x0010 Deny read and write access. **O_DENYW** 0x0020 Deny write access. **O_DENYR** 0x0030 Deny read access. **O_DENYNONE** 0x0040 Allow reads and writes. **O_NOINHERIT** 0x0080 Children cannot use this file. **O_NDELAY** 0x0100 Device should not block for I/O on this file. **O_CREAT** 0x0200 File should be created if it doesn't exist. **O_TRUNC** 0x0400 File should be truncated to 0 **BYTE**s if it already exists. **O_EXCL** 0x0800 Open should fail if file already exists. **O_TTY** 0x2000 File is a terminal. **O_HEAD** 0x4000 File is a pseudo-terminal "master." **O_LOCK** 0x8000 File has been locked.
pos	This field is initialized to 0 when a file is created and should be used by the device driver to store the file position pointer.
devinfo	This field is reserved for use between the file system and the device driver and may be used as desired. The exception to this is if the file is a TTY, in which case *devinfo* must be a pointer to a *tty* structure.
fc	This is the file cookie for the file as follows: `typedef struct f_cookie`

	`{`
	`        FILESYS   *fs;`
	`        UWORD     dev;`
	`        UWORD     aux;`
	`        LONG      index;`
	`} fcookie;`
	
	fs is a pointer to the file system structure responsible for this device. *dev* is a **UWORD** giving a useful device ID (such as the **Rwabs()** device number). The meaning of *aux* is file system dependent. *index* should be used by file systems to provide a unique means of identifying a file.

dev	This is a pointer to the **DEVDRV** structure of the device driver responsible for this file.
next	This pointer may be used by device drivers to link copies of duplicate file descriptors to implement file locking or sharing code.

Upon successful return from the **Dcntl()** call, a pointer to a **kerinfo** structure will be returned. The **kerinfo** structure is defined below:

```
typedef LONG (*Func)();

struct kerinfo
{
        WORD       maj_version;
        WORD       min_version;
        UWORD      default_mode;
        WORD       reserved1;

        Func       *bios_tab;
        Func       *dos_tab;

        VOID       (*drvchng)( UWORD dev );

        VOID       (*trace)( char *, ... );
        VOID       (*debug)( char *, ... );
        VOID       (*alert)( char *, ... );
        VOID       (*fatal)( char *, ... );

        VOIDP      (*kmalloc)( LONG size );
        VOID       (*kfree)( VOIDP memptr );
        VOIDP      (*umalloc)( LONG size );
        VOID       (*ufree)( LONG memptr );

        WORD       (*strnicmp)( char *str1, char *str2, WORD maxsrch );
        WORD       (*stricmp)( char *str1, char *str2 );
        char *     (*strlwr)( char *str );
        char *     (*strupr)( char *str );
        WORD       (*sprintf)( char *strbuf, const char *fmtstr, ... );

        VOID       (*millis_time)( ULONG ms, WORD *td );
        LONG       (*unixtim)( UWORD time, UWORD date );
        LONG       (*dostim)( LONG unixtime );

        VOID       (*nap)( UWORD n );
        VOID       (*sleep)( WORD que, WORD cond );
        VOID       (*wake)( WORD que, WORD cond );
        VOID       (*wakeselect)( LONG proc );

        WORD       (*denyshare)( FILEPTR *list, FILEPTR *f );
        LOCK *     (*denylock)( LOCK *list, LOCK *new );
        LONG      res2[9];
};
```

The members of the **kerinfo** structure are defined as follows:

Member	Meaning
maj_version	This **WORD** contains the kernel version number.
min_version	This **WORD** contains the minor kernel version number.
default_mode	This **UWORD** contains the default access permissions for a file.
reserved1	Reserved.
bios_tab	This is a pointer to the **BIOS** function jump table. Calling *bios_tab*[0x00]() is equivalent to calling **Getmpb()** and is the only safe way from within a device driver or file system.
dos_tab	This is a pointer to the **GEMDOS** function jump table. Calling *dos_tab*[0x3D]() is equivalent to calling **Fopen()** and is the only safe way from within a device driver or file system.
drvchng	This function should be called by a device driver if a media change was detected on the device during an operation. The parameter *dev* is the **BIOS** device number of the device.
trace	This function is used to send information messages to the kernel for debugging purposes.
debug	This function is used to send error messages to the kernel for debugging purposes.
alert	This function is used to send serious error messages to the kernel for debugging purposes.
fatal	This function is used to send fatal error messages to the kernel for debugging purposes.
kmalloc	Use this internal heap memory management function to allocate memory.
kfree	Use this internal heap memory management function to free memory allocated with *kmalloc*().
umalloc	Use this internal heap memory management function to allocate memory and attach it to the current process. The memory will be released automatically when the current process exits.
ufree	Use this internal heap memory management function to allocate memory allocated with *ufree*().
strnicmp	This function compares *maxsrch* characters of *str1* to *str2* and returns a negative value if *str1* is lower than *str2*, a positive value if *str1* is higher than *str2*, or 0 if they are equal.
stricmp	This function compares two **NULL** terminated strings, *str1* to *str2,* and returns a negative value if *str1* is lower than *str2*, a positive value if *str1* is higher than *str2*, or 0 if they are equal.
strlwr	This function converts all alphabetic characters in *str* to lower case.
strupr	This function converts all alphabetic characters in *str* to upper case.
sprintf	This function is the same as the 'C' library sprintf() function except that it will only convert **SPRINTF_MAX** characters (defined in TOSDEFS.H).
millis_time	This function converts the millisecond time value in *ms* to a **GEMDOS** time in *td*[0] and date in *td*[1].
unixtim	This function converts a **GEMDOS** time and date in a UNIX format **LONG**.
dostim	This function converts a UNIX format **LONG** time/date value into a **GEMDOS** time/date value. The return value contains the time in the upper **WORD** and the date in the lower **WORD**.
nap	This function causes a delay of *n* milliseconds.
sleep	This function causes the current process to sleep, placing it on the system que *que* until condition *cond* is met.
wake	This function causes all processes in que *que*, waiting for condition *cond*, to be woken.
wakeselect	This function wakes a process named by the code *proc* currently doing a select operation.
denyshare	This function determines whether the sharing mode of *f* conflicts with any of the files given in the linked list *list.*
denylock	This function determines whether a new lock *new* conflicts with any existing lock in the linked list *list*. The **LOCK** structure is used internally by the kernel and is defined as follows:

	`typedef struct ilock`
	`{`
	`    FLOCK         l;`
	`    struct ilock  *next;`
	`    LONG          reserved[4];`
	`} LOCK;`
	
	l is the structure actually containing the lock data (as defined in **Fcntl()**). *next* is a pointer to the next **LOCK** structure in the linked list or **NULL** if this is the last lock. *reserved* is a pointer to four **LONG**s currently reserved.

res2	These longwords are reserved for future expansion.

Loadable File Systems

MiNT supports loadable file systems to provide support for those other than **TOS** (such as POSIX, HPFS, ISO 9660 CD-ROM, etc.) The **MiNT** kernel will automatically load file system '.XFS' executables found in the \MULTITOS or root directory. As of **MiNT** version 1.08, it is also possible to have a TSR program install a file system with the **Dcntl()** call.

When the file system is executed by **MiNT** (i.e. not via **Dcntl()**), **MiNT** creates an 8K stack and shrinks the TPA so a call to **Mshrink()** is not necessary. The first instruction of the code segment of the file is JSR'ed to with a pointer to a **kerinfo** (as defined above) structure at 4(sp). The file system should use this entry point to ensure that it is running on the minimum version of **MiNT** needed and that any other aspects of the system are what is required for the file system to operate.

It is not necessary to scan existing drives to determine if they are compatible with the file system as that is accomplished with the file system *root()* function (defined below). If the file system needs to make **MiNT** aware of drives that would not be automatically recognized by the system, it should update the longword variable *_drvbits* at location 0x04F2 appropriately.

If the file system was unable to initialize itself or the host system is incapable of supporting it, the entry stub should return with a value of 0L in d0. If the file system installs successfully, it should return a pointer to a **FILESYS** (defined below) structure in d0. A file system should never call **Pterm()** or **Ptermres()**.

All file system functions, including the entry stub, must preserve registers d2-d7 and a2-a7. Any return values should be returned in d0. Function arguments are passed on the stack. The following listing defines the **FILESYS** structure:

```
typedef struct filesys
{
       struct filesys     *next;
       LONG               fsflags;
       LONG               (*root)( WORD drv, fcookie *fc );
       LONG               (*lookup)( fcookie *dir, char *name, fcookie *fc );
       LONG               (*creat)( fcookie *dir, char *name, UWORD mode, WORD
       attrib,
                                      fcookie *fc );
       DEVDRV             *(*getdev)( fcookie *fc, LONG *devspecial );
```

```
        LONG                (*getxattr)( fcookie *file, XATTR *xattr );
        LONG                (*chattr)( fcookie *file, WORD attr );
        LONG                (*chown)( fcookie *file, WORD uid, WORD gid );
        LONG                (*chmode)( fcookie *file, WORD mode );
        LONG                (*mkdir)( fcookie *dir, char *name, UWORD mode );
        LONG                (*rmdir)( fcookie *dir, char *name );
        LONG                (*remove)( fcookie *dir, char *name );
        LONG                (*getname)( fcookie *relto, fcookie *dir, char *pathname
        );
        LONG                (*rename)( fcookie *olddir, fcookie *oldname,
                                         fcookie *newdir, fcookie *newname );
        LONG                (*opendir)( DIR *dirh, WORD tosflag );
        LONG                (*readdir)( DIR *dirh, char *name, WORD namelen,
                                         fcookie *fc );
        LONG                (*rewinddir)( DIR *dirh );
        LONG                (*closedir)( DIR *dirh );
        LONG                (*pathconf)( fcookie *dir, WORD which );
        LONG                (*dfree)( fcookie *dir, long *buf );
        LONG                (*writelabel)( fcookie *dir, char *name );
        LONG                (*readlabel)( fcookie *dir, char *name );
        LONG                (*symlink)( fcookie *dir, char *name, char *to );
        LONG                (*readlink)( fcookie *file, char *buf, short buflen );
        LONG                (*hardlink)( fcookie *fromdir, char *fromname,
                                         fcookie *todir, char *toname );
        LONG                (*fscntl)( fcookie *dir, char *name, WORD cmd, LONG arg
        );
        LONG                (*dskchng)( WORD dev );
        LONG                zero;
} FILESYS;
```

The members of the **FILESYS** structure are interpreted by **MiNT** as follows:

Member	Meaning
next	This member is a pointer to the next **FILESYS** structure in the kernel's linked list. It should be left as **NULL**.
fsflags	This is a bit mask of flags which define attributes of the file system as follows: __Name__ / __Mask__ / __Meaning__ **FS_KNOPARSE** 0x01 Kernel shouldn't do directory parsing (common for networked file systems). **FS_CASESENSITIVE** 0x02 File system names are case-sensitive (common for Unix compatible file systems). **FS_NOXBIT** 0x04 Files capable of being read are capable of being executed (present in most file systems).
root	This function is called by the kernel to retrieve a file cookie for the root directory of the drive associated with **BIOS** device *dev*. When initializing, the kernel will query each file system, in turn, to determine which file system should handle a particular drive. If your file system recognizes the drive specified by *dev* it should fill in the **fcookie** structure as appropriate and return **E_OK**. If the drive is not compatible with your file system, return an appropriate negative **GEMDOS** error code (usually **EDRIVE**).

lookup	This function should translate a file name into a cookie. If the **FS_KNOPARSE** bit of *fsflags* is not set, *name* will be the name of a file in the directory specified by the **fcookie** *dir*. If the **FS_KNOPARSE** bit was set, *name* will be a path name relative to the specified directory *dir*. If the file is found, the **fcookie** structure *fc* should be filled in with appropriate details and either **E_OK** or **EMOUNT** (if *name* is '..' and *dir* specifies the root directory) should be returned, otherwise an appropriate error code (like **EFILNF**) should be returned. A *lookup()* call with a **NULL** *name* or with a *name* of '.' should always succeed and return a cookie representing the current directory. When creating a file cookie, symbolic links should never be followed.
creat	This function is used by the kernel to instruct the file system to create a file named *name* in the directory specified by *dir* with *attrib* attributes (as defined by **Fattrib()**) and *mode* permissions as follows:

Name	Mask	Permission
S_IXOTH	0x0001	Execute permission for all others.
S_IWOTH	0x0002	Write permission for all others.
S_IROTH	0x0004	Read permission for all others.
S_IXGRP	0x0008	Execute permission for processes with same group ID.
S_IWGRP	0x0010	Write permission for processes with same group ID.
S_IRGRP	0x0020	Read permission for processes with same group ID.
S_IXUSR	0x0040	Execute permission for processes with same user ID.
S_IWUSR	0x0080	Write permission for processes with same user ID.
S_IRUSR	0x0100	Read permission for processes with same user ID.
S_ISVTX	0x0200	Unused
S_ISGID	0x0400	Alter effective group ID when executing this file.
S_ISUID	0x0800	Alter effective user ID when executing this file.
S_IFCHR	0x2000	File is a **BIOS** special file.
S_IFDIR	0x4000	File is a directory.
S_IFREG	0x8000	File is a regular file.
S_IFIFO	0xA000	File is a FIFO.
S_IMEM	0xC000	File is a memory region.
S_IFLNK	0xE000	File is a symbolic link.

If the file is created successfully, the fcookie structure fc should be filled in to represent the newly created file and **E_OK** should be returned. On an error, an appropriate **GEMDOS** error code should be returned.

getdev	This function is used by the kernel to identify the device driver that should be used to do file I/O on the file named by *fc*. The function should return a pointer to the device driver and place a user-defined value in the longword pointed to by *devspecial*. If the function fails, the function should return and place a negative **GEMDOS** error code in the longword pointed to by *devspecial*.
getxattr	This function should fill in the **XATTR** structure pointed to by *xattr* with the extended attributes of file *fc*. If the function succeeds, the routine should return **E_OK**, otherwise a negative **GEMDOS** error code should be returned.
chattr	This function is called by the kernel to instruct the file system to change the attributes of file *fc* to those in *attr* (with only the low eight bits being signifigant). The function should return a standard **GEMDOS** error code on exit.
chown	This function is called by the kernel to instruct the file system to change the file *fc*'s group and user ownership to *gid* and *uid* respectively. The kernel checks access permissions prior to calling this function so the file system does not have to.

chmode	This function is called by the kernel to instruct the file system to change the access permissions of file *fc* to those in *mode*. The *mode* parameter passed to this function will never contain anything but access permission information (i.e. no file type information will be contained in *mode*). The call should return a standard **GEMDOS** error code on exit.
mkdir	This function should create a new subdirectory called *name* in directory *dir* with access permissions of *mode*. The file system should ensure that directories such as '.' and '..' are created and that a standard **GEMDOS** error code is returned.
rmdir	This function should remove the directory whose name is *name* and whose cookie is pointed to by *dir*. This call should allow the removal of symbolic links to directories and return a standard **GEMDOS** error code.
remove	This function should delete the file named *name* that resides in directory *dir*. If more than one 'hard' link to this file exists, then only this link should be destroyed and the file contents should be left untouched. Symbolic links to file *fc*, however, should be removed. This function should not allow the deletion of directories and should return with a standard **GEMDOS** error code.
getname	This function should fill in the buffer pointed to by *pathname* with as many as **PATH_MAX** (128) characters of the path name of directory *dir* expressed relatively to directory *relto*. If relto and dir point to the same directory, a **NULL** string should be returned. For example, if *relto* points to directory "\FOO" and *dir* points to directory "\FOO\BAR\SUB" then *pathname* should be filled in with "\BAR\SUB".
rename	This function should rename the file *oldname* which resides in directory *olddir* to the new name *newname* which resides in *newdir*. The file system may choose to support or not support cross-directory renames. The function should return a standard **GEMDOS** error code. If no renames at all are supported then **EINVFN** should be returned.

opendir	This function opens directory *dirh* for reading. The parameter *tosflag* is a copy of the *flags* member of the **DIR** structure as defined below:
	
	`typedef struct dirstruct`
	`{`
	`        fcookie   fc;           /* Directory cookie */`
	`        UWORD     index;        /* Index of current entry */`
	`        UWORD     flags;        /* TOS_SEARCH (1) or 0 */`
	`        char      fsstuff[60]; /* File system dependent */`
	`} DIR;`
	
	If *tosflags* (*dirstruct.flags*) is contains the mask **TOS_SEARCH** the file system is responsible for parsing the names into something readable by **TOS** domain applications. The file system should initialize the *index* and *fsstuff* members of *dirh* and return an appropriate **GEMDOS** error code.

readdir	This function should read the next filename from directory *dirh*. The **fcookie** structure *fc* should be filled in with the details of this file. If *dirh->flags* does not contain the mask **TOS_SEARCH** then the filename should be copied into the buffer pointed to by *name*. If *dirh->flags* does contain the mask **TOS_SEARCH** then the first four bytes of *name* should be treated as a longword and filled in with an index value uniquely identifying the file and the filename should be copied starting at *&name[4]*. In either case, if the filename is longer than *namelen*, rather than filling in the buffer *name*, the function should return with **ENAMETOOLONG**. If this is the last file in the directory, **ENMFIL** should be returned, otherwise return **E_OK**.
rewinddir	This function should reset the members of *dirh* so that any internal pointers point at the first file of directory *dirh*. This function should return a standard **GEMDOS** error code.
closedir	This function should clear any allocated memory and clean up any structures used by the search on *dirh*. This function should return a standard **GEMDOS** error code.

pathconf	This function should return information about the directory *dir* based on mode *mode*. For *mode* values and return values, see **Dpathconf()**.
dfree	This function should return free space information about the drive directory *dir* is located on. The format of the buffer pointed to by *buf* is the same as is used by **Dfree()**.This function should return a standard **GEMDOS** error code.
writelabel	This function is used to change the volume name of a drive which contains the directory *dir*. The new name *name* should be used to write (or rename the volume label). If the write is actually an attempt to rename the label and the file system does not support this function then **EACCDN** should be returned. If the file system does not support the concept of volume labels then **EINVFN** should be returned. Otherwise, a return value of **E_OK** is appropriate.
readlabel	This function should copy the volume label name of the drive on which directory *dir* is contained in the buffer *name*. If *namelen* is less than the size of the volume name, **ENAMETOOLONG** should be returned. If the concept of volume names is not supported by the file system, **EINVFN** should be returned. If no volume name was ever created, **EFILNF** should be returned. Upon successful error of the call, **E_OK** should be returned.
symlink	This function should create a symbolic link in directory *dir* named *name*. The symbolic link should contain the **NULL** terminated string in *to*. If the file system does not support symbolic links it should return **EINVFN**, otherwise a standard **GEMDOS** error code should be returned.
readlink	This function should copy the contents of symbolic link *file* into buffer *buf*. If the length of the contents of the symbolic link is greater than *buflen*, **ENAMETOOLONG** should be returned. If the file system does not support symbolic links, **EINVFN** should be returned. In all other cases, a standard **GEMDOS** error code should be returned.
hardlink	This function should create a 'hard' link called *toname* residing in *todir* from the file named *fromname* residing in *fromdir*. If the file system does not support hard links, **EINVFN** should be returned. Otherwise, a standard **GEMDOS** error code should be returned.
fscntl	This function performs a file system specific function on a file whose name is *name* that resides in directory *dir*. The *cmd* and *arg* functions parallel those of **Dcntl()**. In most cases, this function should simply return **EINVFN**. If your file system wishes to expose special features to the user through **Dcntrl()** then your file system should handle them here as it sees fit.
dskchng	This function is used by the kernel to confirm a 'media change' state reported by **Mediach()**. If the file system agrees that a media change has taken place, it should invalidate any appropriate buffers, free any allocated memory associated with the device, and return 1. The kernel will then invalidate any open files and relog the drive with the *root()* functions of each installed file system. If a media change has not taken place, simply return a value of 0.
zero	This member is reserved for future expansion and must be set to 0L.

MiNT Interprocess Communication

Pipelines

A pipeline is a special file used for data communication in which the data being read or written is kept in memory. Pipes are created by **Fcreate()**'ing a file in the special directory 'U:\PIPE'. A process which initially opens a pipe is considered the 'server.' Processes writing to or reading from the open pipe are called 'clients.' Both servers and clients may read to and write from the pipe.

Fcreate()'s *attr* byte takes on a special meaning with pipes as follows:

Name	Bit	Meaning
FA_UNIDIR	0x01	If this bit is set, the pipe will be unidirectional (the server can only write, the client can only read).
FA_SOFTPIPE	0x02	Setting this bit causes reads when no one is writing to return **EOF** and writes when no one is reading to raise the signal **SIGPIPE**.
FA_TTY	0x04	Setting this bit will make the pipe a pseudo-TTY, i.e. any characters written by the server will be interpreted (CTRL-C will cause a **SIGINT** signal to be generated to all clients).

Fpipe() can also be used to create pipes quickly with the **MiNT** kernel resolving any name conflicts. A pipe is deleted when all processes that had obtained a handle to it **Fclose()** it.

A single process may serve as both the client and the server if it maintains two handles (one obtained from **Fopen()** and one from **Fcreate()**). In addition, child processes of the server may inherit the file handle, and thus the server end of the pipe.

A special system call, **Salert()**, sends a string to a pipe called 'U:\PIPE\ALERT'. If a handler is present that reads from this pipe, an alert with the text string will be displayed.

Signals

Signals are messages sent to a process that interrupt normal program flow in a way that may be defined by the receiving application. Signals are sent to a process with the function **Pkill()**. The call is named **Pkill()** because the default action for most signals is the termination of the process. If a process expects to receive signals it should use **Psignal()**, **Psigsetmask()**, **Psigblock()**, or **Psigaction()** to modify that behavior by installing a handler routine, ignoring the signal, or blocking the signal completely.

Signal handlers should return by executing a 680x0 RTS instruction or by calling **Psigreturn()**. Current signals sent and recognized by **MiNT** processes are as follows:

Signal	Number	Meaning
SIGNULL	0	This signal is actually a dead signal since it has no effect and is never delivered. Its only purpose is to determine if a child process has exited. A **Pkill()** call with this signal number will return successfully if the process is still running or fail if not.
SIGHUP	1	This signal indicates that the terminal connected to the process is no longer valid. This signal is sent by window managers to processes when the user has closed your window. The default action for this signal is to kill the process.
SIGINT	2	This signal indicates that the user has interrupted the process with CTRL-C. The default action for this signal is to kill the process.
SIGQUIT	3	This signal is sent when the user presses CTRL-\. The default action for this signal is to kill the process.

SIGILL	4	This signal is sent after a 680x0 Illegal Instruction Exception has occurred. The default action for this signal is to kill the process. Catching this signal is unrecommended.
SIGTRAP	5	This signal is sent after each instruction is executed when the system is in single-step trace mode. Debuggers should catch this signal, other processes should not.
SIGABRT	6	This signal is sent when something has gone wrong internally and the program should be aborted immediately. The default action for this signal is to kill the process. It is unrecommended that you catch this signal.
SIGPRIV	7	This signal is sent to a process that attempts to execute an instruction that may only be executed in supervisor mode while in user mode. The default action for this signal is to kill the process.
SIGFPE	8	This signal is sent when a division by 0 or floating-point exception occurs. The default action for this signal is to kill the process.
SIGKILL	9	This signal forcibly kills the process. There is no way to catch or ignore this signal.
SIGBUS	10	This signal is sent when a 680x0 Bus Error Exception occurs. The default action for this signal is to kill the process.
SIGSEGV	11	This signal is sent when a 680x0 Address Error Exception occurs. The default action for this signal is to kill the process.
SIGSYS	12	This signal is sent when an argument to a system call is bad or out of range and the call doesn't have a way to report errors. For instance, **Super(**0L**)** will send this signal when already in supervisor mode. The default action for this signal is to kill the process.
SIGPIPE	13	This signal is sent when a pipe you were writing to has no readers. The default action for this signal is to kill the process.
SIGALRM	14	This signal is sent when an alarm sent by **Talarm()** is triggered. The default action for this signal is to kill the process.
SIGTERM	15	This signal indicates a 'polite' request for the process to cleanup & exit. This signal is sent when a process is dragged to the trashcan on the desktop. The default action for this signal is to kill the process.
SIGSTOP	17	This signal is sent to a process to suspend it. It cannot be caught, blocked, or ignored. This signal is usually used by debuggers.
SIGTSTP	18	This signal is sent when the user presses CTRL-Z requesting that the process suspend itself. The default action for this signal is to suspend the process until a **SIGCONT** signal is caught.
SIGCONT	19	This signal is sent to restart a process stopped with **SIGSTOP** or **SIGTSTP**. The default action for this signal is to resume the process.

SIGCHLD	20	This signal is sent when a child process has exited or has been suspended. As a default, this signal causes no action.
SIGTTIN	21	This signal is sent when a process attempts to read from a terminal in a process group other than its own. The default action is to suspend the process.
SIGTTOU	22	This signal is sent when a process attempts to write to a terminal in a process group other than its own. The default action is to suspend the process.
SIGIO	23	This signal is sent to indicate that I/O is possible on a file descriptor. The default action for this signal is to kill the process.
SIGXCPU	24	This signal is sent when the maximum CPU time allocated to a process has been used. This signal will continue to be sent to a process until it exits. The default action for this signal is to kill the process.
SIGXFSZ	25	This signal is sent to a process when it attempts to modify a file in a way that causes it to exceed the processes' maximum file size limit. The default action for this signal is to kill the process.
SIGVTALRM	26	This signal is sent to a process which has exceed its maximum time limit. The default action for this signal is to kill the process.
SIGPROF	27	This signal is sent to a process to indicate that its profiling time has expired. The default action for this signal is to kill the process.
SIGWINCH	28	This signal indicates that the size of the window in which your process was running has changed. If the process cares about window size it can use **Fcntl()** to obtain the new size. The default action for this signal is to do nothing.
SIGUSR1	29	This signal is one of two user-defined signals. The default action for this signal is to kill the process.
SIGUSR2	30	This signal is one of two user-defined signals. The default action for this signal is to kill the process.

Memory Sharing

With the enforcement of memory protection under **MultiTOS**, the availability of shared memory blocks is important for applications wishing to share blocks of memory. A shared memory block is opened by **Fcreate()**'ing a file in the directory 'U:\SHM'. After that, a memory block allocated with **Malloc()** or **Mxalloc()** may be attached to the file with **Fcntl(** *handle*, *memptr*, **SHMSETBLK)**.

Any process which uses **Fopen()** and **Fcntl()** with a parameter of **SHMGETBLK** can now read that memory as if it were a disk file. After a process obtains the address of a shared memory block with **SHMGETBLK** the memory is guaranteed to be valid until it calls **Mfree()** on that block even if it **Fclose()**'s the original file handle.

Note that the address returned by **Fcntl()** may be different in different processes. Because of this, data in shared memory blocks should not contain absolute pointers.

When a process is finished with a shared memory block, it should **Mfree()** the address returned by the **Fcntl()** call. A shared memory block is also deleted by the **Fdelete()** call if the file is currently unopened by any other processes.

Other Methods of Communication

Psemaphore() can be used to create named flags which can synchronize the behavior of multiple applications (if adhered to). **Pmsg()** is used to send simple messages between two processes.

MiNT Debugging

MiNT allows a processes' TEXT, DATA, and BSS space to be read and written to with standard **GEMDOS** file commands by opening the process on 'U:\PROC\' A file named "TEST" with a **MiNT** identification of 10 could be opened by specifying the name as 'U:\PROC\TEST.10' or 'U:\PROC\.10'. Opening a file to 'U:\PROC\.-1' will open your own process whereas opening a file to 'U:\PROC\.-2' will open your parent process.

Tracing

A process may be setup for tracing in a number of ways. A child process may be started in trace mode by OR'ing 0x8000 with the **Pexec()** mode number in a **Pexec()** call. A process may also trace another process by opening it as described above and using the **Fcntl()** call with a parameter of **PTRACESFLAGS**. Processes may start tracing on themselves if their parent is prepared for it.

When in trace mode, the process being traced halts and generates a **SIGCHLD** signal to its tracer after every instruction (unless this action is modified). The example below shows how to obtain the process ID of the stopped child and the signal that caused the child to stop.

```
#define WIFSTOPPED(x)      (((int)((x) & 0xFF)==0x7F) && ((int)(((x)>>8)&0xFF)!=0))
#define WSTOPSIG(x)        ((int)(((x)>>8) & 0xFF))

void
HandleSignal( LONG signo )
{
        WORD pid;
        WORD childsignal;
        ULONG r;

        if( signo == SIGCHLD )
        {
              r = Pwait3( 0x2, 0L );
              if( WIFSTOPPED( r ) )
              {
                      pid = r >> 16;
                      childsignal = WSTOPSIG( r );
              }
        }
}
```

After reception of this signal, the child process may be restarted with **Fcntl()** using either the **PTRACEGO**, **PTRACEFLOW**, or **PTRACESTEP** commands. Setting **PTRACEFLOW** or

PTRACESTEP causes a **SIGTRAP** signal to be raised on the next program flow change (ex: BRA or JMP) or the instruction respectively.

Modifying the Process Context

A processes' registers may be modified during tracing using the method as illustrated in the following example:

```
struct context
{
       LONG      regs[15];        // Registers d0-d7, a0-a6
       LONG      usp;             // User stack pointer
       WORD      sr;              // Status register
       LONG      pc;              // Program counter
       LONG      ssp;             // Supervisor stack pointer
       LONG      tvec;            // GEMDOS terminate vector
       char      fstate[216];     // Internal FPU state
       LONG      fregs[3*8];      // Registers FP0-FP7
       LONG      fctrl[3]         // Registers FPCR/FPSR/FPIAR

       // More undocumented fields exist here
} c;

void
ModifyContext( LONG handle )
{
       LONG curprocaddr, ctxtsize;

       Fcntl( handle, &curprocaddr, PPROCADDR );
       Fcntl( handle, &ctxtsize, PCTXTSIZE );

       curprocaddr -= 2 * ctxtsize;

       Fseek( curprocaddr, handle, SEEK_SET );
       Fread( handle, (LONG)sizeof(struct context), &c );

       /* Modify context c here */

       Fseek( curprocaddr, handle, SEEK_SET );
       Fwrite( handle, (LONG)sizeof(struct context), &c );
}
```

MiNT Debugging Keys

MiNT may be programmed to output special debugging messages to the debugging device through the use of special system keys. The supported system keys are shown in the table below:

Key Combination	Meaning
CTRL-ALT-F1	Increase the system debugging level by one.
CTRL-ALT-F2	Decrease the system debugging level by one.
CTRL-ALT-F3	Cycle the **BIOS** output device number used for system debugging messages. This key cycles **BIOS** devices in the order 1-6-7-8-9-2.
CTRL-ALT-F4	Restore debugging output to the console device.
CTRL-ALT-F5	Output a memory usage map to the debugging device.
CTRL-ALT-F6	Output a list of all system processes to the debugging device.

CTRL-ALT-F7	Toggles debug 'logging' off and on. When debug logging is on, a 50-line buffer is maintained which contains recent debugging messages. Each time a new debugging message is output, the entire 50 line buffer is output as well.
CTRL-ALT-F8	Outputs the 50-line debug log to the debugging device.
CTRL-ALT-F9	Outputs the system memory map to the debugging device. The memory protection flags of each page are shown.
CTRL-ALT-F10	Outputs an extended system memory map to the debugging device. The memory protection status, owner's PID, and format of each memory block are output to the debugging device.

CTRL-ALT-F1 and CTRL-ALT-F2 alter the current system debugging level. **MiNT** supports four debugging levels as follows:

Level	Meaning
0	Only fatal OS errors are reported to the debugging device (this is the default mode).
1	Processor exceptions are output to the debugging device.
2	Processor exceptions and failed system calls are output to the debugging device.
3	Constant **MiNT** status reports, processor exceptions, and failed system calls are output to the debugging device.

The MINT.CNF File

MultiTOS looks for an ASCII text file upon bootup called 'MINT.CNF' which may be used to execute commands or set **MiNT** variables. The following table illustrates what commands are recognized in the 'MINT.CNF' file:

Command	Example	Meaning
cd	`cd c:\multitos`	Change the **GEMDOS** working directory.
echo	`echo “Atari Computer Booting...”`	Echo a string to the screen.
ren	`ren c:\test.prg c:\test.app`	Rename a file.
sln	`sln c:\level1\level2\level3 u:\deep`	Create a symbolic link on drive 'U:'.
alias	`alias x: u:\proc`	Create an alias drive.
exec	`exec c:\sam.prg`	Execute a program.

The following **MiNT** variables may be set in the 'MINT.CNF' file:

Variable	Meaning
INIT	Execute the named **TOS** program. For example: `INIT=c:\multitos\sam.prg`
GEM	Execute the named **GEM** program. For example: `GEM=c:\multitos\miniwin.app`
CON	Redirect console input and output to the named file. For example: `CON=u:\dev\modem1`
PRN	Redirect printer output to the named file. For example: `PRN=c:\spool.txt`
DEBUG_LEVEL	Set the **MiNT** debugging level (default is 0). For example: `DEBUG_LEVEL=1`
DEBUG_DEVNO	Set the **BIOS** device number that **MiNT** will send debugging messages to. For example: `DEBUG_DEVNO=1`
SLICES	Set the number of 20ms time slices given to an application at a time (the default is 2). For example: `SLICES=3`
MAXMEM	Set the maximum amount of memory (in kilobytes) any application can be allocated (the default is unlimited). For example: `MAXMEM=8192`
BIOSBUF	Enable/Disable **Bconout()** optimizations. The parameter should be 'Y' to enable or 'N' to disable these optimizations. For example: `BIOSBUF=Y`

GEMDOS Character Functions

GEMDOS provides a number of functions to communicate on a character basis with the default system devices. Because of irregularities with these calls in some **TOS** versions, usage of the **BIOS** functions is usually recommended instead (the **BIOS** does not support redirection, however).

The **GEMDOS** character functions are illustrated in the table below:

Device:	Input	Output	Status
con:	**Cconin()** - Character **Cnecin()** - No Echo **Cconrs()** - String	**Cconout()** - Character **Cconws()** - String	**Cconis()** - Input **Cconos()** - Output
prn:	None	**Cprnout()**	**Cprnos()**

aux:	**Cauxin()**	**Cauxout()**	**Cauxis()** - Input **Cauxos()** - Output
N/A	**Crawio()** and **Crawcin()**	**Crawio()**	**Cconis()** - Input **Cconos()** - Output

GEMDOS Time & Date Functions

GEMDOS provides four functions for the manipulation of time. **Tsetdate()** and **Tsettime()** set the date and time respectively. **Tgetdate()** and **Tgettime()** get the date and time respectively.

As of **TOS** 1.02, the **GEMDOS** time functions also update the **BIOS** time.

GEMDOS Function Calling Procedure

GEMDOS system functions are called via the TRAP #1 exception. Function arguments are pushed onto the current stack in reverse order followed by the function opcode. The calling application is responsible for correctly resetting the stack pointer after the call.

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

The following example for **Super()** illustrates calling **GEMDOS** from assembly language:

```
clr.l       -(sp)
move.w      #$20,-(sp)
trap        #1
addq.l      #4,sp
```

‘C’ compilers often provide a reusable interface to **GEMDOS** that allows new **GEMDOS** calls to be added with a macro as in the following example:

```
#define Super( a )  gemdos( 0x20, a )
```

The gemdos() function used in the above macro can be written in assembly language as follows:

```
              .globl      _gemdos

              .text
_gemdos:
              move.l      (sp)+, t1sav     ; Save return address
              trap        #1               ; Call GEMDOS
              move.l      t1sav,-(sp)      ; Restore return address
              rts

              .bss

t1sav:        ds.l        1                ; Return address storage

              .end
```

GEMDOS is not guaranteed to be re-entrant and therefore should not be called from an interrupt handler.