USFiles®
Processor-Independent DOS/Win95 File System
User’s Manual

Revision 3.02

October 2001

U S SOFTWARE.
A Lantronix company

Copyright and Trademark Information

United States Software Corporation
7175 NW Evergreen Parkway, Suite 100
Hillsboro, OR 97124
(503) 844-6614
Fax (503) 844-6480
E-mail: support@ussw.com

Documentation Conventions

Computer output and code examples: Courier, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (*mt_busy*).

File names: Times bold (the file **usrclk.asm**), in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

NOTE: Indicates important information.

CAUTION: Indicates potential damage to hardware or data.

WARNING: Indicates potential injury to users.

Revision History

Revision	Date
Previous issue	June 1997
Reorganized and reformatted	December 1997
3.00 Added new features	February 2000
3.01 Updated for new directory structure	September 2000
3.02 New configuration options	October 2001

NOTES

Contents

1. USFiles Internals

Chapter Contents

Introduction

This Chapter will explain a bit about how USFiles handles things internally. As we have mentioned before, three layers of functions implement the USFiles stream I/O features. The top layer is the *stream I/O* layer. The middle layer is the *file manager*, and the lower layer is the *device driver*. The standard C-level functions like ***mt_fgetc()*** call the file manager routines associated with the stream, which in turn call the driver routines. The driver also has *interrupt service routines* associated with it for interrupt-driven devices.

The system is configured for a fixed number of devices as specified in **userio.h**, and a maximum number of open streams (i.e. ports or files) as defined by the `NUMSTREAMS` parameter, which is described in Chapter 4, *Configuring USFiles*. Each stream has a structure of type `FILE` (an alias of `MTFILE`) associated with it, which contains all of the control information for the stream. These `FILE` structures are dynamically allocated by ***mt_fopen()*** via a call to ***alloc_mem()***. The `FILE` type structure is defined in **mtio.h**, and it contains all the information about the stream, including pointers to other structures needed for control of the stream. If you develop a driver for a new type of device, it might be necessary to add some new structures to some of the union types in this file.

The file manager, device driver, and ISR (Interrupt Service Routine) all access the `FILE` structure for the stream they are currently operating on. We will refer to the `FILE` structure for a stream as its *file descriptor*.

The *file descriptor* for each stream contains pointers to the *file manager*, *device driver*, and *device data* structure associated with that stream. The file manager is a structure of type `FILEMAN` (defined in **mtio.h**). This structure consists of function pointers to the routines that constitute the file manager. The driver is a similar structure of type `DRIVER`, which contains function pointers to the functions that constitute the device driver, and the data structure is of type `DEVICE`.

If your system has several ports with the same characteristics (same type UART chip, diskette, etc.) they would most likely be using the same driver and file manager. The BIOS-based driver supplied in **biosdrv.c** combines diskette and hard disk control into one driver. The high bit of the unit number selects hard disk versus diskette operation. If you develop new drivers for hard disk and diskette drives, it is more likely that these functions will be in separate drivers if you are controlling the hardware directly. Not all file managers require the presence of the device driver. The pipe file manager this is provided with MultiTask! is an example of this.

File Managers

`pcfm`	PC File System Manager (in this package).
`sfm`	Serial File Manager (in MultiTask! product).
`pipefm`	Pipe File Manager (in MultiTask! product).
cdfm	CD-ROM File System Manager (in USFiles for CD-ROM).

Drivers

`biosdrv`	USFiles diskette/hard disk driver for 80 x 86 PC-style system via BIOS calls (in this package).
`ramdrv`	USFiles RAM disk driver (in this package).
`flopdrv`	USFiles PC diskette driver accessing controller directly (in this package, developed for 80x86 real mode).
`lbahddrv`	USFiles ATA (IDE) LBA mode hard disk driver accessing ATA interface directly on a PC. Also works with non-LBA drives (in this package, developed for 80x86 real mode).
cdromdrv	ATAPI CD-ROM driver (in USFiles for CD-ROM).
pcmciadrv	Driver used to initialize PCMCIA controller for use with CompactFlash Cards (in USFiles for CompactFlash).
`driver0`	MultiTask! serial driver (in MultiTask! product).
other	User-supplied drivers for interfacing with either `pcfm`, or `sfm`, or other file managers.

Code Hierarchy

Figure 3-1 below illustrates the code hierarchy. Only the files with names in bold are part of USFiles. The PC file manager is divided among the four files **pcfmapi.c, pcfmbuf.c, pcfmclus.c**, and **pcfmdir.c**.

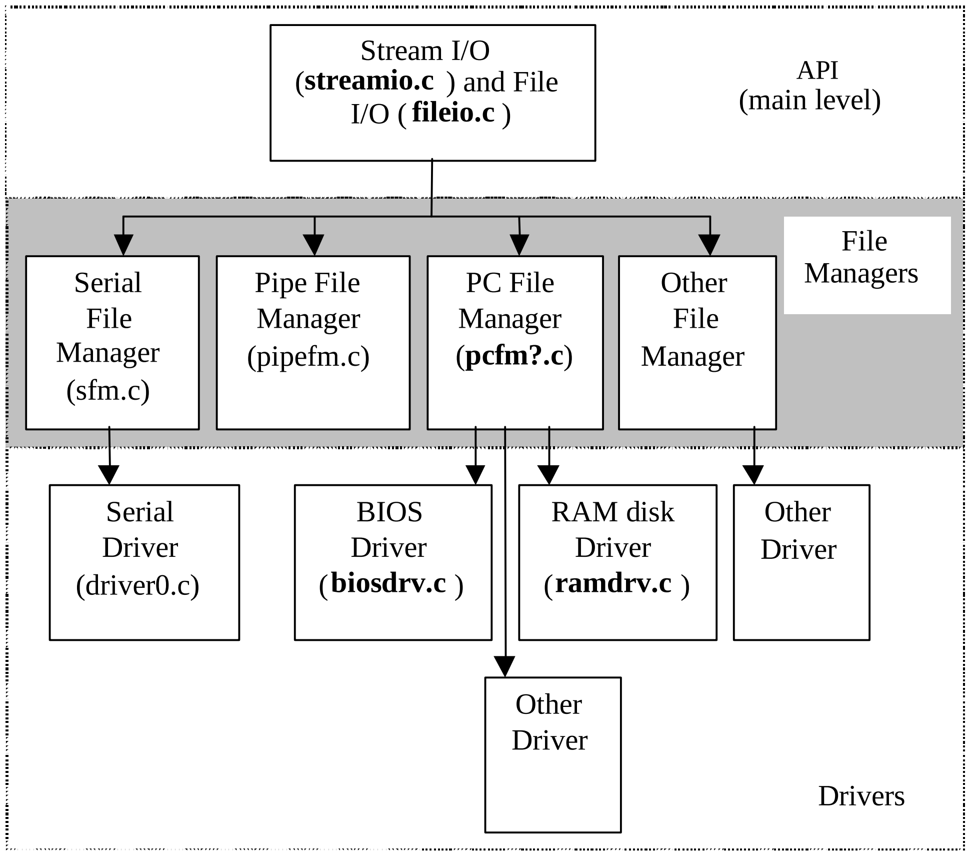
Stream I/O (streamio.c) and File I/O (fileio.c)
API (main level)
Serial File Manager (sfm.c)
Pipe File Manager (pipefm.c)
PC File Manager (pcfm?.c)
Other File Manager
File Managers
Serial Driver (driver0.c)
BIOS Driver (biosdrv.c)
RAM disk Driver (ramdrv.c)
Other Driver
Other Driver
Drivers

Figure 3-1: Code Hierarchy for USFiles

Stream I/O

The stream I/O routines are primarily found in the files **streamio.c** and **fileio.c**. Applications will typically directly interface with only the stream I/O layer. There are a few functions provided as utilities to the user at the file manager level that will bypass stream I/O. These will be discussed in the *File Manager* section of this chapter.

Stream I/O Function Summary

The stream I/O functions that USFiles provides are:

mt_fopen	***mt_fread***	***mt_fwrite***	***mt_fgetc***
mt_fgets	***mt_fputc***	***mt_fputs***	***mt_printf***
mt_fprintf	***mt_sprintf***	***mt_vsprintf***	***mt_sscanf***
mt_fgetpos	***mt_fsetpos***	***mt_fseek***	***mt_ftell***
mt_fflush	***mt_fclose***	***mt_mkdir***	***mt_remove***
mt_rewind	***mt_rmdir***	***mt_feof***	***mt_ferror***
mt_clearerr	***mt_rename***		

The full syntax of these functions can be found in the *Library Reference* chapter, but they can be divided into several groups.

Functions for File Control

mt_fopen	***Opens a file***
mt_fclose	***Closes a file***
mt_rename	***Renames a file or directory***
mt_remove	***Removes a file***
mt_mkdir	***Creates a directory***
mt_rmdir	***Removes a directory***
mt_rewind	***Sets file pointer to beginning***

mt_fseek	***Positions file pointer to desired location***
mt_fsetpos	***Positions file pointer to desired location***
mt_ftell	***Reports position of file pointer***
mt_fgetpos	***Reports position of file pointer***

Functions for Writing

mt_fwrite	Writes to a file
mt_fputc	Writes a single character to a file
mt_fputs	Writes a string to a file
mt_printf	Writes formatted output to `stdout`
mt_fprintf	Writes formatted output to a file
mt_sprintf	Writes formatted output to a string
mt_vsprintf	Writes formatted output to a string
mt_fflush	Flushes file's output buffer

Functions for Reading

mt_fread	***Reads from a file***
mt_fgetc	***Reads a single character from a file***
mt_fgets	***Reads a string from a file***
mt_sscanf	***Converts a string according to specified format***

Functions for Error Reporting

mt_feof	***Tests for end of file***
mt_ferror	***Returns file error condition***
mt_clearerr	***Clears file error condition***

Error Reporting

Error reporting deserves some special attention, since errors may arise in various places. For the ANSI stream I/O functions that we provide, we follow the ANSI specification. These functions often return an integer value. If that value is zero, it means that the function executed successfully. If it is non-zero (usually `EOF`), then an error has occurred. There are exceptions to this, so please check Chapter 5, *Library Reference*, for particular functions. To determine the details of an error, the variable *`errno`* is used. The possible values that *`errno`* can have can be found in Appendix G, *Error Codes*.

NOTE: Be aware that no function ever clears *`errno`*. Once it is set, you must be sure to clear it after you handle any error recovery.

The ***mt_fopen()*** function does not return an integer, but rather a file pointer. If the pointer returned is `NULL`, then this signals an error. The following code snippet gives an example of how an error encountered by ***mt_fopen()*** could be tested.

```
fp = mt_fopen(“C:\\myfile.txt”,”w”);
if( !fp ){
   if( errno == ENOPATH)
          /* Device probably not in device_tab[] */
   else
          /* Some default error handling */
}
errno = 0;    /* Clear errno */
```

This example is reentrant if the RTOS implementation of *`errno`* is multitasking safe, which is the case for MultiTask! and TronTask!. For other RTOSes, you will have to study the RTOS or tool chain implementation of *`errno`*.

Another (less reliable) method of error checking is provided by the ***mt_ferror()*** function, which checks the error code for a specific open file pointer. Although this is an ANSI C function, it is not specified under what conditions the file pointer error code should be set. USFiles sets this error code when a driver error is encountered. Often (but not necessarily always) when a driver error is reported, *errno* is set to the same value.

Please check Chapter 5, *Library Reference* to determine how each function reports an error individually. Not all functions return EOF for an error and zero for success. For example, if ***mt_fread()*** returns a value of zero, an error has occurred. We feel that a careful use of *errno* works best to determine error conditions.

File Allocation

USFiles maintains a static array of file pointers. The number of elements in this array is determined by the NUMSTREAMS parameter, which is discussed in Chapter 4, *Configuring USFiles*. When a file is opened, memory for the file structure is dynamically allocated at the stream I/O level. This file structure is represented in Figure 3-2, and the complete structure definition can be found in **mtio.h**.

Device Number (index into device table)
File Number
Error Code
Pointer to Driver Jump Table
Pointer to File Manager Jump Table
Pointer to Device Data Structure
File System Parameters (stream-specific data)
Other Items

Figure 3-2: Elements of the MTFILE Structure

The device number identifies which device table entry is associated with the file. The file number indicates which entry in the open streams table the file occupies. The error code is used to indicate driver errors that occurred while operating on the file, and the pointers provide access to the functions that are used to handle the file operations, which are coordinated using the file system parameters.

The Device Table

To enable the stream I/O functions to communicate with a particular device, we need to configure the device table *`device_tab[]`* in **devtab.c** (found in the **siosrc** directory). The device table is an array of device structures. The format for the device structure is defined in **mtio.h** and is outlined in Figure 3-3.

Device Name
Device Type (serial, PC file, etc.)
Capabilities (read, write, etc.)
Unit Number
Partition Number
Pointer to Device Driver Jump Table
Pointer to Device File Manager Jump Table
File Pointer*
Flags
Number of Open Paths

* Not used by USFiles

Figure 3-3: Elements in the `device_s` Structure

This is a sample entry in a device table for the first partition on a hard drive:

```
&pcparmC,                 /* device-dependent data */
“C”,                      /* name */
FM_PCFM,                  /* device type = PC device */
0xf,                      /* bits: text write read */
0x80,                     /* unit# */
0,                        /* partition */
(DRIVER *)&lbadrv_s,      /* pointer to driver */
&pcfm,                    /* pointer to file manager */
NULL,                     /* pointer to FILE */
0,                        /* flags */
0,                        /* # open paths (RAM) */
```

At this point, the important items to note are that the driver is `lbadrv_s`, the file manager is `pcfm`, and the device name is `“C”`. Many of the device table entry fields are not used by USFiles.

When a call to ***mt_fopen()*** is made by the application, the entire path name to the file must be specified. This includes the drive name. If we wanted to open a file on the hard drive described above, we would need to specify the name as `c:\file.txt`. The ***mt_fopen()*** function recognizes that the portion of the file name in front of the colon is the device name. It then searches the device table until it finds the device with that name. Once it is found, the device table entry indicates which file manager and driver will be used to access the file. In this example, the file manager is for a PC file system, and the driver is a logical block addressing hard drive driver. Stream I/O functions will not call driver functions directly. They only deal with the file manager.

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

WARNING: The file **devtab.c** uses a new device structure. If you are copying any older device structures into **devtab.c**, be careful to reorder the fields. See **mtio.h** for the specifics.

The default device configuration for USFiles is simply a RAM disk (R:). To use another type of device you will have to add it to the device table. The file **siosrc\devtab.c** has samples for various kinds of devices. You will have to uncomment structure and variable definitions to support new devices. Look for file managers, device drivers, and device parameter structures in **devtab.c**.

File Managers

Once stream I/O has found the device table entry that belongs to a device, it is able to call the file manager functions.

File Manager Function Summary

The `pcfm` file manager provided is capable of controlling all types of DOS-compatible disk drives, including diskette drives, hard drives, and RAM- or ROM-based drives. FAT32 partitions are supported through an add-on to USFiles. The `sfm` file manager included with MultiTask! can control all types of serial ports. Each of these requires the addition of the appropriate low-level driver routines to interface to the actual hardware.

These are the defined file manager functions for any file manager. They are most often used in this order:

open()	***Opens a file***
read()	***Reads bytes from a file***
readln()	***Reads a string from a file***
write()	***Writes bytes to a file***
writeln()	***Writes a string to a file***
seek()	***Positions the file pointer***
makdir()	***Creates a directory***
_delete()	***Removes a file***
fmioctl()	***Other I/O control functions***
close()	***Closes a file***

The specific routines that constitute the `pcfm` file manager are:

pcfm_open()
pcfm_read()
pcfm_readln()
pcfm_write()
pcfm_writeln()
pcfm_seek()
pcfm_makdir()
pcfm_delete()
pcfm_fmioctl()
pcfm_close()

File Manager Function Descriptions

File Manager close() function

```
int close(MTFILE *fp)
```

The file manager ***close()*** function ends access to the stream, making its position in the open streams table available. This function returns zero if successful, or `EOF` if an error is detected.

File Manager delete() function

```
int _delete(MTFILE *fp)
```

The file manager ***delete()*** function removes the file described by *`fp`* from the file system. The file's storage is freed and its directory entry deleted. A zero is returned if no errors are detected.

File Manager fmioctl() function

```
int fmioctl(MTFILE *fp, int function, void *arg,
            size_t size);
```

The file manager ***fmioctl()*** function performs any other miscellaneous operations on the device. The function ***IO_FLUSH*** is defined for all device types to flush all output buffers associated with the device. Other operations are implementation-dependent.

The parameter `arg` is used to pass the argument(s) for the request. This may be a pointer to a simple variable, or a pointer to a structure if several variables need to be passed in. Values can be passed in, out, or in both directions.

The parameter size is useful if `arg` points to a variable-size buffer. The length of the buffer could be indicated by building a structure that includes size information. However, including size as a separate argument allows an arbitrary starting point and length to be passed without requiring the buffer to be modified or copied. The `size` parameter may also be useful for passing small integers with minimal overhead.

File Manager makdir() function

```
int makdir(MTFILE *fp)
```

The file manager ***makdir()*** function turns the newly create path described by `fp` into a directory. A zero is returned if no errors are detected.

File Manager open() function

```
MTFILE * open(MTFILE *fp, char *filename)
```

The ***open()*** function of a file manager is passed the file descriptor pointer `fp` and the `filename`. The ***open()*** function fills in the file descriptor structure for the stream, where necessary, with initial values, and may call a driver ***init*** routine to initialize the device. The ***open()*** function returns a pointer to the file descriptor structure if it is successful; otherwise it returns a `NULL` pointer.

File Manager read() function

```
size_t read(MTFILE *fp, byte *buf, size_t bytes)
```

The file manager ***read()*** routine reads the number of bytes specified by *`bytes`*, from the stream specified by *`fp`* into the buffer pointed to by *`buf`*.

File Manager readln() function

```
size_t readln(MTFILE *fp, byte *buf, size_t bytes)
```

The file manager ***readln()*** (read line) routine reads at most the number of bytes specified by *`bytes`*, from the stream specified by *`fp`* into the buffer pointed to by *`buf`*. The read will terminate early if the `EOL_CHAR` is read. In all other respects this call is the same as the ***read()*** function.

File Manager seek() function

```
int seek(MTFILE *fp, uint32 position)
```

The file manager ***seek()*** function takes action to assure that the next read or write to the file will be at absolute *`position`* bytes from the beginning of the file. A non-zero error code is returned if an error is detected.

File Manager write() function

```
size_t write(MTFILE *fp, byte *buf, size_t bytes)
```

The file manager ***write()*** function writes the number of bytes specified by *`bytes`* taken from the memory buffer pointed to by *`buf`*, and writes these to the stream specified by *`fp`*. The actual number of bytes written is returned by this function. This will be zero if an error occurs.

File Manager writeln() function

```
size_t writeln(MTFILE *fp, byte *buf, size_t bytes)
```

The file manager ***writeln()*** function is identical to the ***write()*** function except that the write will terminate before *`bytes`* have been transmitted if an `EOL_CHAR` is encountered in the output stream. (The ***writeln()*** terminates after the transmission of the `EOL_CHAR`.)

Text and Binary Files

Whether or not “text” mode stream I/O differs from “binary” mode depends upon the specific file manager or driver being used by the stream. Text mode is implemented for PCFM devices (disks). If the file is opened in text mode (which is the default), carriage return characters are removed upon read, transforming carriage return-linefeed pairs into only linefeeds (“`\n`”). On writes, each “`\n`” is written as “`\r\n`”.

Additional File Manager Functions

In addition to the functions provided via the file manager structure, **pcfm.c** contains a few other functions that may be safely accessed from an application. These additional functions are:

free_byte_cnt()	***Returns number of unallocated bytes on drive***
free_clust_cnt()	***Returns number of unallocated clusters on drive***
pcfm_chmod()	***Changes attributes of file (specified by path)***
pcfm_chmodfp()	***Changes attributes of file (specified by pointer)***
pcfm_chtime()	***Changes time and date of file (specified by path)***
pcfm_chtimefp()	***Changes time and date of file (specified by pointer)***
pcfm_chvlabel()	***Changes an existing volume label***

See also: Chapter 5, *Library Reference*, describes how to use these functions.

Buffers

The PC file manager maintains an array of physical record (generally referred to as sector) buffers. The number of buffers used is determined by the value of `NUMBUFFERS`, which is user configurable. The file in which you will find `NUMBUFFERS` depends on the RTOS being used.

See also: Chapter 4, *Configuring USFiles*, for more information on buffers.

The buffer is defined in **mtio.h** as:

```
typedef struct pcfm_buffer_s{
    DEVICE    *devp;
    uint32    lsect;
    uint32    serial_no;
    byte            *userbuf;
    uint16     nsects;
    uint16     age;
    int        error_status;
    byte       flags;
    byte       filenum;
    byte       devnum;
    byte       padding;
    byte       buf[512];
} PCFM_BUFFER;
```

Buffers are used to hold physical record contents in an attempt to limit the number of times that the driver has to read or write to the device. Since the buffers are maintained in memory, reading from or writing to them is much faster than accessing a disk.

By using the *`age`* parameter USFiles makes an attempt to keep track of buffers that are accessed regularly. When a buffer is allocated to a particular sector, we increase that buffer's *`age`* by a certain value. When we search through the buffer array and opt not to use a given buffer, we decrease that buffer's *`age`*. In this manner, buffers that are accessed frequently have higher *`ages`* than buffers that are rarely used.

This becomes important when we reach a situation where all buffers are being used, and we need a buffer to perform some operation. If the sector we are looking for is not already buffered, then we have to take one of the other buffers. We look for the oldest (least accessed, lowest *age* value) buffer that can be used. If this buffer is "dirty" (its sector contents have been modified but not yet saved to disk), then we save the sector and use it for the new sector. If the buffer is not dirty, then we simply use the buffer.

If you are doing binary (not text) reads and writes of data segments that span at least a full sector, then buffers may be bypassed. This can result in faster data transfer times.

Adding New File Managers

USFiles is delivered with a DOS file system manager, and a CD-ROM ISO 9660 file system manager can be provided as well. A serial file manager comes with MultiTask!, so several file managers are available to you. If you need to develop your own file manager, it can be done. It will involve a significant time investment, though.

The file **mtio.h** has most of the definitions necessary for adding a new file manager. The basic file manager structure is:

```
struct fileman_s{

int (*open)(MTFILE *, char *); /* character open routine */

  size_t (*read)(MTFILE *, byte *, size_t); /* read from
  stream */

  size_t (*readln)(MTFILE *, byte *, size_t); /* read line
  routine */

  size_t (*write)(MTFILE *, byte *, size_t); /* write to
  stream */

  size_t (*writeln)(MTFILE *, byte *, size_t); /* write line
  to stream */

  int (*close)(MTFILE *); /* close stream */

  int (*seek)(MTFILE *, uint32); /* reposition file */

  int (*makdir)(MTFILE *); /* create a directory */

  int (*_delete)(MTFILE *); /* delete a file */
  int (*fmioctl)(MTFILE *,int,void*,size_t); /*
  miscellaneous control */
  };
```

This file manager structure should be suitable for any file system, since it only depends on the MTFILE structure, which is common to all of USFiles.

The source file for the particular file manager defines the specific file manager structure. For example, the PC file manager is defined at the end of **pcfmapi.c** as:

```
FILEMAN const pcfm = {
  pcfm_open,
  pcfm_read,
  pcfm_readln,
  pcfm_write,
  pcfm_writeln,
  pcfm_close,
  pcfm_seek,
  pcfm_makdir,
  pcfm_delete,
  pcfm_fmioctl
};
```

The device table needs to know about the file managers in use, so **devtab.c** includes the line:

```
extern FILEMAN const pcfm;
```

When developing a new file manager, the PC file manager serves as a good starting point (see **pcfmapi.c**).

File System Parameters

File system parameters are defined in **mtio.h** in the form `XXX_FSP`. For example, `PCFM_FSP` provides PC file system parameters, `PIPE_FSP` provides useful parameters for a pipe file system, and `CDFM_FSP` provides the parameters for a CD file system.

These are joined together in a union like:

```
typedef union   fm_parm_u{
    struct sfm_fsp s;
    struct pcfm_fsp p;
    struct pipe_fsp pi;
    struct cdfm_fsp cd;
}FM_FSP_U;
```

Any new file system will likely have its own set of useful parameters. This new structure should be defined as `NEWFM_FSP` (for example), and an entry should be added to the `FM_FSP_U` union.

The parameter items are accessed through calls like this, which is from ***count_seq_clusters()*** in **pcfm.c**:

```
last = fp->fsp.p.cur_clust;
```

Identifying a File System

To be able to easily identify the file system associated with a particular device, we use macros defined in **mtio.h**. The ones that are defined for distribution with USFiles are:

```
#define FM_SFM            0
#define FM_PCFM     1
#define FM_PIPE     2
#define FM_CDFM     3
```

If you define a new file manager, this list should be updated.

Device Drivers

Each DOS device driver is defined to USFiles by specifying eight routines:

1. Driver initialize
2. Cylinder, Head, Sector read
3. CHS write
4. Format
5. Logical block read
6. Logical block write
7. Time stamp
8. Disk change

The time stamp routine records the date/time code at the address specified. The value should be encoded in MS-DOS directory entry format. See the ***biosdrv_timestamp()*** function the ***mak_ftime()*** and ***mak_fdate()*** macros for details. This routine can be replaced with a dummy with no ill effect other than the directory entries on files will not show the actual date/time of access. The dummy routine may return nothing, or zeroes, or anything else you desire.

The CHS read is only called by PCFM to get the first logical sector from a diskette that contains the DOS BPB, which describes the disk format (number of tracks and sectors, etc.). The logical block read and write routines perform the bulk of the work, although on disks using CHS format, the logical block routines calculate the appropriate cylinder, head, and sector, and call the CHS routines.

You can get away with only writing one read and one write routine. If you are doing diskette or old hard drive access (identifying a sector by cylinder, head, and sector), then you only need to develop a CHS read and a CHS write routine, which accept a drive number, cylinder, head, sector number, number of sectors, and a buffer pointer for the data.

The logical block read and write routines compute the cylinder, head, and sector number from the logical block number and call the CHS read/write routines. The logical read and write routines in **biosdrv.c** can be used for this purpose if you develop substitutes for the ***biosdrv_raw_read()*** and ***biosdrv_raw_write()*** routines in that file.

The logical block read and write routines are appropriate for accessing hard drives that use logical block addressing (LBA). In this case, the CHS read and write routines are simply dummy routines. A driver (called **lbahddrv.c**) that supports logical block addressing is provided with the 80x86 real mode release of USFiles.

See also: Chapter 7, *Porting Guide,* for assistance if you need to integrate your driver with an RTOS.

Driver Function Summary

The file manager makes the calls to the device driver functions. The device driver functions for an `sfm` device are described in the MultiTask! manual. The functions comprising a `pcfm` (disk) driver are described below.

A `pcfm` device driver consists of these functions, which are typically used in this order:

init()	***Initialize device***
format()	***Physically formats sector***
raw_read()	***Read sector specified by cylinder, head, and sector***
raw_write()	***Write sector specified by cylinder, head, and sector***
read()	***Read sector specified as a logical sector number***
write()	***Write sector specified as a logical sector number***
timestamp()	***Reports time and date***
diskchange()	***Reports if a disk has been changed***

For a specific instance of a driver, these routines will be given the above-mentioned names with a unique prefix prepended to them to designate the driver (e.g., ***pcfdrv_raw_read()***).

The exact function performed by these routines depends upon what the file manager calling them expects. The division of responsibilities between the file manager and the device driver may be altered if a new file manager is developed. The expectations of the `pcfm` file manager are as follows.

Driver Function Descriptions

Driver diskchange() function

```
int diskchange(DEVICE *devp);
```

The ***diskchange()*** function returns a non-zero error code if a media change or other error has been detected since the last read or write operation to the drive. Sensing the disk change status line for diskette drives is useful for this operation. For non-removable media, this is a dummy routine that always returns 0.

The purpose of returning an error code is to be able to distinguish between a disk change (`EDSKCHG`) and having no disk present (`ENORESP`). Other errors may be returned if required by your driver.

See also: Appendix A, *Handling Disk Changes*

Driver format() function

```
int format(DEVICE *devp, int cylinder, int head,
                     int nsects, void *buffer);
```

The ***format()*** function should physically format the track specified by *`cylinder`* and *`head`*, on the drive specified by *`devp->unit_no`*. The *`buffer`* contains format information for *`nsects`* if applicable. The ***format()*** function returns zero if successful or non-zero on an error. If the format function will not be used, ***format()*** can be a dummy function that does nothing.

Driver init() function

```
int init(DEVICE *devp);
```

The initialize function is called once for each drive controlled by the driver. It should do any initialization required by the device, such as hardware reset, initialize interrupt vectors, etc. Zero is returned if successful, and a non-zero error code if not. If more than one drive is called, ***init()*** should keep a static flag to tell it that it has already been called so it can avoid repeating operations that should be done only once. The ***flopdrv_init()*** function hooks controller-interrupt vectors on the first call, and for each time it is called (it will be called only once for each drive) it starts a motor control task.

Driver raw_read() function

```
int raw_read(DEVICE *devp, int cylinder, int head,
             int sector, int, nsects, void *buffer);
```

The file descriptor and all associated structures will be initialized with all available information before the driver ***raw_read()*** function is called. This function should:

1. Seek the specified drive to the indicated cylinder. The drive is specified by *`devp->unit_no`*..
2. Attempt to read *`nsects`* consecutive sectors starting with *`sector`*, under the specified *`head`* into the *`buffer`* indicated.
3. Retry several times if an error is encountered, and then return a non-zero error code if the error persists. If the read is successful, return a zero value with the data in the buffer.

Driver raw_write() function

```
int raw_write(DEVICE *devp, int cylinder, int head,
              int sector, int, nsects, void *buffer);
```

The file descriptor and all associated structures will be initialized with all available information before the driver ***raw_write()*** function is called. This function should:

1. Seek the specified drive to the indicated cylinder. The drive number is contained in *`devp->unit_no`*.
2. Attempt to write *`nsects`* consecutive sectors starting with *`sector`*, under the specified *`head`* into the *`buffer`* indicated.
3. Retry several times if an error is encountered, and then return a non-zero error code if the error persists. If the read is successful, return a zero value with the data in the buffer.

Driver read() function

```
int read(uint32 logical_sect, PCFM_BUFFER *bufp);
```

The driver ***read()*** function reads the indicated logical sector into the buffer at *`bufp->buf`*, from the drive indicated by the *`bufp`* structure. Any other information required by the driver about the device can be found through the *`bufp`* structure. Parameters in *`bufp`* may indicate that a consecutive number of sectors are to be read, in which case this action should be taken. The supplied driver in **biosdrv.c** should be used as a guide for coding a new driver.

This read routine may accomplish its function by converting the logical sector number into a physical cylinder, head, and sector number, and then calling the ***raw_read()*** routine, or by directly accessing the disk drive. If the read is successful the driver ***read()*** function returns a value of zero; otherwise it returns a non-zero error code.

If *`bufp->usrbuf`* is not `NULL`, then the ***read()*** function will read *`bufp->nsects`* sectors to the user's buffer at *`bufp->userbuf`*, instead of transferring a single sector to *`bufp->buf`*.

Driver timestamp() function

```
void timestamp(uint16 *time);
```

The ***timestamp()*** routine gets the time and date if available from the system, and encodes the time as a 16-bit value which it writes to location `*time`.

The date is encoded as a 16-bit value that is written to `time[1]`. See **biosdrv.c** for an explanation of the time and date encoding format. This routine can be replaced by a dummy function with no ill effect other than the date and time recorded in the directory entries for the file system will not be correct.

Driver write() function

```
int write(uint32 logical_sect, PCFM_BUFFER *bufp);
```

The driver ***write()*** function writes the indicated logical sector into the buffer at *bufp->buf*, from the drive indicated by *fp->device->unit_no*. Any other information required by the driver about the device can be found through the *bufp* structure. Parameters in *bufp* may indicate that a consecutive number of sectors are to be written, in which case this action should be taken. The supplied driver in **biosdrv.c** should be used as a guide for coding a new driver.

This write routine may accomplish its function by converting the logical sector number into a physical cylinder, head, and sector number, and then calling a ***raw_write()*** routine, or by accessing the disk drive directly. If the write is successful it returns a value of zero; otherwise it returns a non-zero error code.

If *bufp->usrbuf* is not `NULL`, then the ***write()*** function will write *bufp->nsects* sectors from the user's buffer at *bufp->userbuf* to disk, instead of transferring a single sector from *bufp->buf*.

RAM Disk Driver

USFiles is supplied with a configurable RAM disk device driver, which should be used in initial tests to verify that USFiles is functioning in your target environment. The RAM disk driver is contained in the file **ramdrv.c**.

The RAM disk driver can support any number of logical RAM drives. The number of drives supported is defined by the parameter `NUMRAMDRIVES` in the driver source file. This driver source duplicates the same timestamp routines used in **biosdrv.c**. If you are using both drivers, you really only need one set of the timestamp routines. You can leave one copy out and change the `DRIVER` structure for one of the devices in **userio.h** to use the timestamp routine of the other driver.

Each RAM drive must be initialized by a call to ***ramdrive_init()*** before you can open any file on that drive.

DOS BIOS Driver

USFiles is supplied with a DOS device driver called **biosdrv.c**, which may be used in x86 target systems supplied with PC-compatible BIOS. This driver supports diskette and hard disk devices.

The BIOS calls in **biosdrv.c** present an oversimplification of what a diskette read or write sector will actually do. You will usually need to step the head to track zero on initialization and record the track that each drive is currently on. Then when presented with a request for a sector on a new track, issue a seek command to step to the new track, and then the read command.

The raw driver routines should retry several times on soft errors and return a non-zero error code on failure. These routines should return zero when successful. The actual error codes can be user-defined, but need to be coordinated between the raw driver routines and the ***biosdrv_error_handler()*** routine (in **biosdrv.c**) or its replacement.

The ***biosdrv_error_handler()*** routine can be modified to take whatever action you want for critical errors. One critical error it must respond to is the diskette being changed. The BIOS routines return an error code of `0x06` if a disk change is signaled by the diskette drive. This value is currently hard-coded into ***biosdrv_error_handler()*** and also ***pcfm_get_bpb()*** in **pcfm.c**. This error needs to be passed back by the raw read/write routines to assure proper operation of the file system.

Hard Disk Driver

The hard disk driver delivered with the 80x86 real mode and i386 protected mode versions of USFiles (**lbahddrv.c**) provides direct access to an IDE hard drive. The drive can either operate in logical block addressing (LBA) mode, or in cylinder, head, sector (CHS) mode. Each drive's unit number in the device table determines how it is accessed.

See also: *Configuring USFiles* chapter for more details.

This driver assumes that PC hardware is in use, which includes the Intel 8259 Programmable Interrupt Controller (PIC) and the Intel 82062 Disk Controller. The driver initialization installs an interrupt service routine (ISR) in the expected DOS vector for IRQ 14. When operating in stand-alone mode, an ISR is also installed into the DOS timer interrupt vector to allow drive commands to timeout.

The hard disk driver does not implement ***raw_read()*** or ***raw_write()*** routines. It strictly uses ***read()*** and ***write()***.

Diskette Driver

The diskette driver is also provided with 80x86 real mode and i386 protected mode, and it has similar assumptions to the hard disk driver. It assumes the presence of the Intel 8259 PIC, 8272 Floppy Disk Controller, and 8237 Programmable DMA Controller. The diskette drive ISR is installed in the DOS vector used by the diskette drive controller.

The notes concerning the BIOS driver error handler also apply to the diskette driver. There are two routines that are used to check for a disk change. One is used only within the driver itself (***internal_pcfdrv_diskchange()***), and does not call the error handler. The other is ***pcfdrv_diskchange()*** and can be called by file manager functions (like ***pcfm_open()***). This routine will call the error handler.

Adding New Device Drivers

The driver structures are also defined in **mtio.h**. The PC file system driver is:

```
struct driver_p {   /* for PC Disk File System */
    int (*init)(DEVICE *);
    int (*raw_read)(DEVICE *, int, int, int, int, void *);
    int (*raw_write)(DEVICE *, int, int, int, int, void *);
    int (*format)(DEVICE *, int, int, int, void *);
    int (*read)(uint32, struct pcfm_buffer_s *);
    int (*write)(uint32, struct pcfm_buffer_s *);
    void (*timestamp)(uint16 *);
    int (*diskchange)(DEVICE *);
};
```

For comparison, the serial driver structure is:

```
struct driver_s {   /* For Serial devices */
    int (*init)(MTFILE *);
    byte (*read)(MTFILE *);
    void (*write)(MTFILE *, byte);
    int (*ioctl)(MTFILE *, int, va_list);
    void (*term)(MTFILE *);
};
```

If you are adding a new driver to USFiles to work with the PC file manager, then we recommend that you define it as an instance of the `driver_p` driver. There is no need to create an entirely new driver structure.

If you need a driver to work with a file manager that you are adding to USFiles, then you may find that a new driver structure is necessary. When developing USFiles for CD-ROM, we found this to be the case.

See also: The file **mtio.h** or Appendix E, *USFiles for CD-ROM*, to see the CD-ROM driver structure.

Any new driver types added to the system need to be included in the driver union in **mtio.h**:

```
union driver_u{
    struct driver_s     s;    /* sfm serial driver */
    struct driver_p     p;    /* pcfm disk driver */
    struct driver_cd    cd;   /* cdfm CD-ROM driver */
};
```

The source file for the particular driver defines the specific driver structure. For example, the RAM disk driver is defined at the end of **ramdrv.c** as:

```
struct driver_p ramdrv_s = {
  ramdrv_init,
  ramdrv_raw_read,
  ramdrv_raw_write,
  ramdrv_format,
  ramdrv_read,
  ramdrv_write,
  ramdrv_timestamp,
  ramdrv_diskchange
};
```

The device table needs to know about the drivers in use, so **devtab.c** includes the line:

```
extern struct driver_p ramdrv_s;
```

Driver Errors

All the driver functions except the ***timestamp()*** routine return an integer value to report errors. When a driver error occurs, some file manager functions will set *errno* to that driver error and signal an error to the application level. If the list of error codes in **mtio.h** (duplicated in Appendix G, *Error Codes*) does not contain a code that adequately describes the situation, you may extend the list.

Device Parameters

Since you can have several types of a particular device (e.g. two diskette drives), we need a mechanism to keep track of the data associated with each one separately. To do this we use a device parameter union (`DEVPARM`). Each device in the device table specifies the variable used to keep track of device parameters. When USFiles is delivered, the device table has two diskette drives (**A:** and **B:**). Each has its own device parameter variable. These are global variables defined in userio.h as:

```
PCFM_PARM   pcparmA = {1}; /* set motor_event for drive A */
PCFM_PARM   pcparmB = {2}; /* set motor_event for drive B */
```

Other fields in the device parameter structure will be filled when a device is initialized.

Depending on the device type, the parameters used to characterize it will differ. Examples of these are `PCFM_PARM` and `CDFM_PARM` in **mtio.h**. Since the driver structure has to support all these different parameter combinations, we use a union of pointers to device parameter structures:

```
typedef union devparm_u{
    PCFM_PARM     *pcd;    /* PC disk I/O parameters */
    SFM_PARM      *pcs;    /* Serial port parameters */
    PIPE_PARM     *pip;    /* Pipe parameters */
    CDFM_PARM     *cdparm; /* CD-ROM drive parameters */
    EXAMPLE_PARM  *other;  /* Add others here */
}DEVPARM;
```

If you add a new device type, you will have to add a new device parameter type to the union.

How It Ties Together

Remember that all of this really is brought together in the device table. The device table entry specifies which file manager, driver, and device parameter variable to use. It is through the device table that we are able to navigate the USFiles three-layer structure.

An Example

As an example of how things are used, let us see what happens when we open a file using ***mt_fopen()***. The function ***mt_fopen()*** is in **streamio.c**. After entering that routine, we determine what capabilities the file is to have, decide which device to access (e.g. **A:**), and set up some parameters for the file pointer. A pointer (*`devp`*) is set up to the device table entry for the device in question, and using the pointer, USFiles calls the appropriate file manager open function via:

```
status = devp->fileman->open(fp, fname);
```

The file manager is specified in the device table entry. For a diskette device, the file manager is `pcfm`, so the above function call will take us to ***pcfm_open()***. In ***pcfm_open()*** we access the driver ***init()*** call. This is done with:

```
devp = fp->device;
if(devp->driver->p.init(devp)){
   /* do stuff */
}
```

The driver that is called in the above example is also specified in the device table. If we were using a CD-ROM device instead of a diskette drive, the file manager open would resolve to ***cdfm_open()***, and within that call, the ***init()*** command could be accessed by:

```
devp = fp->device;
if(devp->driver->cd.init(devp)){
   /* do stuff */
}
```

The difference is resolved by the `driver_u` union, which was described above. Of course, we would have to make sure that the driver shown in the device table is truly a CD-ROM driver. If a new file manager is written with a new driver type, then its function calls would be accessed through:

```
fp->device->driver->newdriver.function(params);
```

It is important to realize that a file manager is associated with only one type of device driver. The PC file manager maps to PC device drivers, and the CD file manager is related to the CD-ROM driver, so this connection is hard-coded into the file managers through the ***p.function()*** or ***cd.funtion()*** lines described above. It seems conceivable that two different file managers could use the same type of driver, but it is difficult to imagine how this would occur. It is not possible for a particular file manager to use different driver types, though.

Function Call Hierarchy

Figure 3-4 shows how the file structure ties the various USFiles internal components together. Table 3-1 shows how the stream I/O, PC file manager, and driver functions all relate. When stream I/O opens a file, it finds the appropriate device table entry, and the file manager and driver pointers are copied from the device table into the file structure. Stream I/O functions then call the file manager, which calls the driver via the pointer in the file structure.

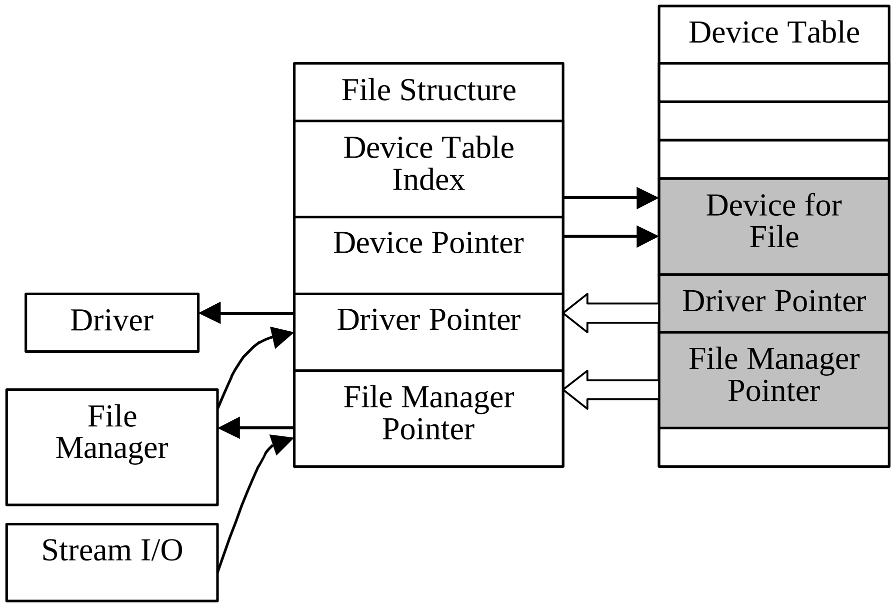

Figure 3-4: Schematic Linking the USFiles Internals Together

Table 3-1: Function Hierarchy

Stream I/O	**File Manager**	**Driver**
mt_clearerr()		
mt_fclose()	pcfm_close()	timestamp(), write()
mt_feof()		
mt_ferror()		
mt_fflush()	pcfm_fmioctl()	write()
mt_fgetc()	pcfm_read()	read()
mt_fgetpos()		
mt_fgets()	pcfm_readln()	read()
mt_fopen()	pcfm_open()	init(), diskchange(), read()
mt_fprintf()	pcfm_write()	write()
mt_fputc()	pcfm_write()	write()
mt_fputs()	pcfm_writeln()	write()
mt_fread()	pcfm_read()	read()
mt_fseek()	pcfm_seek()	read(), write() ***(indirectly called)***
mt_fsetpos()	pcfm_seek()	read(), write()***(indirectly called)***
mt_ftell()		
mt_fwrite()	pcfm_write()	write()
mt_mkdir()	pcfm_makdir()	timestamp(), write()

Table continued on next page.

Table 3-1 (continued): Function Hierarchy

Stream I/O	**File Manager**	**Driver**
mt_printf()	pcfm_write()	write()
mt_readdir()	pcfm_fmioctl(),	
mt_remove()	pcfm_open(), pcfm_delete()	init(), read(), write()
mt_rename()	pcfm_open(), pcfm_delete()	init(), read(), write()
mt_rewind()	pcfm_seek()	read(), write() ***(indirectly called)***
mt_rmdir()	pcfm_open(), pcfm_fmioctl()	init(), read(), write()
mt_sprintf()		
mt_sscanf()		
mt_vsprintf()		

Directory Access

The ***fopen()*** function can be used to access directories as ordinary files (read-only). This will allow you to use ***fseek(), rewind(),*** and ***fread()*** to access a directory. When using long file names, knowing how far to seek and read is not obvious.

USFiles comes with a function, ***mt_readdir()***, to read directory entries from a directory. Here is a sample use:

```
MTFILE *fp;
MT_DIRENT entry;
  fp = mt_fopen(pathname,"d");
  if(!fp)
        iprintf("Error opening directory");
  while ( !mt_feof(fp) ) {
        if(!mt_readdir(fp, &entry))
              iprintf("  %s\n",entry.name);
        else if(errno != 0)
              iprintf("mt_readdir error");
  }
  mt_clearerr(fp);
  mt_fclose(fp);
```

The "`pathname`" above will be the directory to open. For example, "`a:`" would open the root directory, and "`a:\SUBDIR`" would open a subdirectory named **SUBDIR** on drive **A:**.

See also: Please see the entry for ***mt_readdir()*** in Chapter 5, *Library Reference*, for a more detailed description of this function.

Global Variables

USFiles makes use of a few global variables that may be useful when debugging an application.

`DEVICE` *`device_tab[]`*
Device table

`MTFILE` *`*mtstreams[`*`NUMSTREAMS`*`]`*
Open streams table

`MEMHEAD_DEF` *`mem_rootptr[`*`NUMCOLORS`*`]`*
Heap array (only for stand-alone USFiles for 386 protected mode)

`DIR_SEARCH_BLK` *`workblk`*
Used to search for a PC-type file (protected by ***LOCK_FILESYSTEM()***)

`PCFM_BUFFER` *`pcfm_buf[`*`NUMBUFFERS`*`]`*
PC file manager sector buffers

`byte` *`pcfm_agescale`*
Signals when buffer age parameter wraps

2. Configuring USFiles

Chapter Contents

Configuration Overview

There are a variety of items that can be configured for USFiles, ranging from what devices are available to how many buffers are used. This chapter will describe the individual configuration parameters and details, and then provide a summary of these parameters arranged by file.

Configuring Devices

Possibly the most important configuration issue is setting up the device. The device table is an array defined in **devtab.c**. A device table entry consists of initialized data structures that define the device characteristics and map the appropriate file manager and driver routines to the device.

The default USFiles device that resides in the device table is a RAM disk. This device is functional on any board. To add device table entries to match your hardware, you must:

1. Declare the file manager. For USFiles devices this is already done with the line:

   ```
   extern FILEMAN const pcfm;
   ```

   NOTE: The `pcfm` structure is defined in **pcfmapi.c**.

1. Declare the driver. The USFiles RAM disk does this with:

   ```
   extern struct driver_p const ramdrv_s;
   ```

   NOTE: The `ramdrv_s` structure is defined in **ramdrv.c**.

1. Define the variable for storing device dependent data. The USFiles RAM disk uses:

   ```
   PCFM_PARM pcparmR;
   ```

1. Place an entry for the device in `device_tab[]`. The RAM disk entry is:

```
    &pcparmR,              /* device dependent data */
    “R”,                   /* name */
    FM_PCFM,               /* device type = PC device */
    0xF,                   /* bits: text write read */
    0,                     /* unit# */
    0,                     /* partition */
    (DRIVER *)&ramdrv_s,   /* pointer to driver */
    &pcfm,                 /* pointer to file manager */
    NULL,                  /* pointer to FILE */
    0,                     /* flags */
    0                      /* # open paths (RAM) */
```

For USFiles devices the file manager is always of type `FILEMAN`, the driver is always of type `driver_p`, and the device parameter variable is always of type `PCFM_PARM`. When using USFiles for CD-ROM, the driver and device parameter type change. Please see Appendix E, *USFiles for CD-ROM*, for more details.

You will very likely need to add or delete structure initializers from this table. Depending on the file manager and driver, not every element in the `device_s` structure will necessarily be used for a particular device. However, there must be initialized data present for each element as a place holder.

See also: For the definition of the device structure, see **mtio.h** (`struct device_s`) or Chapter 3, *USFiles Internals*.

In the above sample device table entry, the drive will be identified as “R:”. The device name does not need to be a single character; there is no size limit on the device name. The name given in this table is required to be all upper case characters. The pathname given to ***fopen()*** is case insensitive.

The capabilities field of the device table entry (fourth item in `device_s`) can be any combination of:

`CAP_READ`	Read is permitted
`CAP_WRITE`	Write is permitted
`CAP_UPDATE`	Read and Write (`== CAP_READ | CAP_WRITE`)
`CAP_TEXT`	Text mode is permitted

You will need to set up the device table to represent your hardware.

NOTE: Only the lower three bits of the `capabilities` field are significant to USFiles. Any values in the upper 5 bits are ignored. This means that `capabilities = 0x7` is the same as `capabilities = 0xF`.

Unit Numbers

When specifying a diskette or hard drive in the device table, the unit number assigned will affect how it is accessed. The particular bits in the unit number are presented in Table 4-1.

Table 4-1: Bits in the Unit Number

Bit	7	6	5	4	3	2	1	0
Use	drive type	unused				mode	device number	

The explanation of each field is:

drive type	0 = diskette drive; 1 = hard drive
mode	0 = CHS mode; 1 = LBA mode (hard drives only)
device number	Actual unit number

The PC file manager uses the drive type bit to determine how to read the device BPB. The mode bit is used in **lbahddrv.c** to decide whether a logical sector number should be converted to cylinder, head, and sector. For hard drives, LBA mode is preferred, since it allows access to larger devices.

Diskette drives can have numbers ranging from 0 to 3. Hard drives using CHS may be numbered from `80h` to `83h`, and those using LBA can range from `84h` to `87h`. A master drive is specified by bit 0 being clear, and the slave drive is specified by bit 0 being set. Therefore on the primary IDE cable, master drives are `80h` or `84h`, and slave drives are `81h` or `85h`, depending on the access mode.

Configuring Partition Numbers

Hard drive partitions are assigned consecutively. The first partition on a drive is partition 0, the second is partition 1, and so on.

NOTE: When DOS determines drive letters, all primary DOS partitions are assigned letters first, and then the logical drives in extended partitions are handled. For example, if you have two drives, each with a primary partition and a logical drive, then the first disk will have drives **C:** and **E:**. The second disk will have drives **D:** and **F:**. In the USFiles device table, you may associate drive letters with whatever partition you desire.

Configuring Drives and Drivers

The file **siosrc\sioconf.h** has definitions used to control the behavior of test programs in certain situations. When using a RAM disk, the application needs to ‘format’ the device before it can be accessed. By setting RIO to 1 in **sioconf.h**, the test programs know to execute the RAM disk format command.

Likewise, if you are using the i8086 BIOS driver (**biosdrv.c**) to access the hard disk or diskette drives, you should uncomment the line:

```
#define USEBIOS
```

When `USEBIOS` is not defined, the test programs will attempt to restore the interrupt service routines that DOS was originally using. The BIOS driver does not change these routines. So when the BIOS driver is used, we do not want to restore them.

`#define RIO 1`	enables test programs to format the RAM disk
`#define USEBIOS`	will not attempt to restore DOS interrupt service routines

Configuring Streams and Buffers

USFiles allows the user to specify the maximum number of open streams (files) and the number and types of sector buffers used internally. Each sector buffer requires 512 bytes for the sector contents, plus additional space for internal use. The file **siosrc\sioconf.h** contains the specification for the number of streams and buffers allowed.

`NUMSTREAMS`	specifies the maximum number of paths that can be open at the same time. `NUMSTREAMS` must be at least 1, and is limited to a maximum of 254, due to the internal use of a byte to track the index.
`NUMBUFFERS`	specifies the number of general purpose file buffers allocated by USFiles. Each buffer consumes approximately 530 bytes of RAM. At least 2 buffers are required for operation. More buffers give better performance.
`NUMFATBUFS`	specifies the number of buffers to hold FAT sectors only
`NUMDIRBUFS`	specifies the number of buffers to hold directory sectors only
`NUMINFBUFS`	specifies the number of buffers to hold FAT32 file system information sectors only See also: The section on USFiles Tips for more discussion of configuring buffers.

See also: The section on USFiles Tips for more discussion of configuring buffers.

When using DOS 8.3 file names only, USFiles requires approximately 30 KB of ROM and 6 KB of RAM. The RAM size includes 10 file buffers, which is user-configurable.

The total number of buffers will be the sum of the four buffer numbers specified in **siosrc\sioconf.h.**

Buffer Configuration Guidelines

NUMBUFFERS is used for data sectors, so increase this for large files. If there are many files or subdirectories in a single directory, increase NUMDIRBUFS. If you are using very large files (many clusters per file), increase NUMFATBUFS. Use only one file system information sector buffer for each FAT32 partition.

NUMBUFFERS cannot be zero. All the others can be set to zero, and they are zero as the default. Any sectors that do not have a specific buffer pool assigned (e.g. a FAT sector when NUMFATBUFS is 0) will be placed in the general buffer pool (NUMBUFFERS). If you are only using the general buffer pool, then NUMBUFFERS must be at least 2. If either NUMDIRBUFS or NUMFATBUFS is non-zero, then NUMBUFFERS must be at least 1.

To aid in tuning the buffer numbers, there is a symbol in **siosrc\sioconf.h** called USF_BUF_DEBUG. If this is set to 1, then the ***get_buffer()*** function will track buffer usage. The statistics can be displayed by calling ***buf_dump()***. Please refer to **siosrc\usfbdump.c** for the ***buf_dump()*** function call. Below are the statistics that are tracked for buffers.

calls:	The number of times ***get_buffer()*** was called for this type of buffer
pushes:	The number of times ***get_buffer()*** had to write out a dirty buffer to make room for a new buffer of the given type
reads:	The number of times ***get_buffer()*** had to call the driver read function for this buffer type
unused:	The number of buffers of this type that were allocated but not used
total:	The total number of buffers of this type allocated

In general, if the number of pushes is large increase that buffer pool. If there are unused buffers for a given pool, then decrease its size.

VFAT

For information on using and configuring VFAT, please refer to Appendix C, *VFAT*.

Checking Configuration Parameters

The user can also enable careful input parameter checking. To turn on parameter checking, set `USS_SIO_PCHK` in **siosrc\sioconf.h** to 1. This would most likely only be used during development and not for production code. To remove the extra code that checks the input to functions, set `USS_SIO_PCHK` to 0.

NOTE: When passing in addresses from which USFiles will read (e.g. USFiles reads data from the buffer passed to ***mt_fwrite()*** and writes that data to a file), we do not test for a NULL address. Reading from the NULL address will generally not cause catastrophic failures. When passing in an address to which USFiles will write (e.g. ***mt_fread()***), turning on parameter checking will test that the address is not NULL. This will prevent USFiles from writing data to an obviously incorrect region of memory.

Protecting Resources

In integrating USFiles with an RTOS, there are two levels of resource protection used. The first level is protecting the stream I/O layer, and the second is protecting directory-level access in the file system. The method of resource protection varies between the supported RTOSes. Typically a resource or semaphore is used. The RTOS header file has the definitions for ***LOCK_*** and ***UNLOCK_STREAMIO()***, and ***LOCK_*** and ***UNLOCK_FILESYSTEM()***. If this protection is implemented as a resource, then definitions are made for the symbols `STREAM_RESOURCE` and `PCFM_RESOURCE`. The resource definitions (if any) can be found in the file **rtos.h** in the appropriate **siosrc\<*rtos*> subdirectory**.

NOTE: The ***LOCK_FILESYSTEM()*** call may become nested, so a protection mechanism that allows resource nesting is required.

Table 4-2: Symbols for Locking Stream I/O and File System

RTOS	**STREAM_RESOURCE Default Value**	**PCFM_RESOURCE Default Value**
MultiTask!	N/A	1
TronTask!	1	2
Hitachi ITRON	2	1
RX850	N/A	N/A
RX850 Pro	N/A	N/A

NOTE: The RX850 and RX850 Pro RTOSes do not take static resource ID definitions. They will be dynamically configured, hence there are no default values.

`STREAM_RESOURCE`	specifies the resource number used to lock stream I/O access while the current task is accessing it.
`PCFM_RESOURCE`	specifies the resource number used to lock the file system while the current task is accessing it.

You do not need to use the default values for these ID numbers, but these symbols must be defined to valid resource ID numbers.

Setting Timeouts for Device Drivers

If you are using one of our direct-access device drivers on PC-type hardware without an RTOS, then there is one more parameter that you should be aware of. The **depends.h** file defines the clock frequency as `CLOCKHZ`. If you are testing USFiles on a DOS PC in stand-alone mode, then you should set `CLOCKHZ` to 18. If you leave this value at its default setting (182), then the timeout periods for **lbahddrv.c** and **flopdrv.c** will be exceptionally long. Leaving the default setting will not cause an error, but you may wait for a long time for a timeout to occur. This is because the direct access device drivers use the DOS ticker interrupt, which has a frequency of 18.2 Hz. If `CLOCKHZ` does not match this, then our timeout period will not be what we expect, since we convert from milliseconds to clock ticks based on the value of `CLOCKHZ`.

NOTE: If a board support package (BSP) is being used with USFiles, then the **depends.h** file will not be present. The clock speed will be specified in **config.mak** as `USS_CLOCKS_PER_SEC`. See **siosrc\\<*cpu*>\cpunotes.txt** to determine whether your version of USFiles is using a BSP.

Files Used for Configuration

compiler.mak	Specifies target board (see comments in file for details). Located in **config\\<*cpu*>\\<compiler>** directory.
config.mak	Specifies product installation directory, products to build, CPU, compiler and RTOS used, and possibly `USScCLOCKS_PER_SEC`.
devtab.c	Contains the device table.
depends.h	May specify `CLOCKHZ` if no BSP is used. Located in **siosrc\\<*cpu*>\\<compiler>** directory, if present.
rtos.h	May specify `STREAM_RESOURCE` and `PCFM_RESOURCE`. Located in **siosrc\\<*rtos*>** directory.
sioconf.h	Specifies `NUMSTREAMS, NUMBUFFERS, NUMFATBUFS, NUMDIRBUFS,` and `NUMINFBUFS.` For debugging use, `USF_BUF_DEBUG` and `USS_SIO_PCHK` are set. Also used to specify `RIO` and `USEBIOS`. Located in **siosrc** directory.
sio.mak	Specifies `VFAT` and `FAKEUNICODE` (See Appendix C, *VFAT*, for more details). Located in **config** directory.

The file **siosrc\\<*cpu*>\\cpunotes.txt** might contain more information on configuration for a particular processor.

USFiles Tips

This section provides a few suggestions that should improve the performance of USFiles. If you would like to discuss any of these items, please contact us.

Use Short File Names

If possible use short file names. Building up long file name entries can be a time-consuming process.

Use Unique Long File Names

If long file names are necessary, try to make file names in a given directory unique in the first six characters. This would mean using the names **file01_for_testing.tmp**, **file02_for_testing.tmp**, etc. instead of **testing_file01.tmp**, **testing_file02.tmp**, etc. USFiles will more quickly be able to assign a unique 8.3 name to the file.

Do Not Place Too Many Files in a Directory

Do not keep too many files in a given directory. When searching for a file name, USFiles will have to read each entry in the directory until it finds the desired name. Each entry is 32 bytes, so a directory with 1000 files uses 63 sectors (for 8.3 names). If you use long file names, that number could easily double (see the first item above).

Tune Buffer Usage

Performance can be improved by using the use-specific buffers that have been introduced in USFiles 3.07. Some guidelines for use are:

`NUMBUFFERS`	(Number of Devices) x (Number of sectors per cluster)
`NUMFATBUFS`	Number of files
`NUMDIRBUFS`	Number of files
`NUMINFBUFS`	Number of FAT32 devices

These should be used as starting points. Set `USF_BUF_DEBUG` in **siosrc\sioconf.h** to help fine tune these numbers. See the section

"*Configuring Streams and Buffers*" for a description of the buffer tracking statistics. In general, the more buffers the better, but this can lead to large RAM requirements.

Increase Cluster Size

Using a larger cluster size will limit the number of times USFiles has to access the FAT. If you are typically dealing with large files a larger cluster size can provide a significant performance improvement. When dealing with smaller files, you may end up wasting more disk space than you want. There are utilities like Partition Magic* that will allow you to choose the disk cluster size.

3. Library Reference

Chapter Contents

Overview of USFiles Functions

At the user program level, all I/O devices (streams) are accessed through the familiar ANSI C functions: ***fopen, fread, fwrite, fgetc, fgets, fputc, fputs, printf, fprintf, sprintf, vsprintf, sscanf, fflush,*** and ***fclose***. Disk (`pcfm`) devices also accept the functions ***fgetpos, fsetpos, fseek, ftell, mkdir***, and ***remove***. These latter calls will do nothing on a serial (`sfm`) device other than return an error code. All of these functions are supplied in source form and conform to the ANSI specifications with these exceptions: All devices are unbuffered in the ANSI sense of the word. Interrupt-driven serial devices are actually buffered with separate input and output buffers for each device. This buffering is on the level of the interrupt service routine in the driver and not on the higher level buffering as dealt with by the ANSI ***setvbuf*** function. (This function is therefore not supplied.) Disk devices are buffered by at least a full sector at a time when any access is made. The paths defined as `stdin` and `stdout` are not automatically opened when your application is started; they must be explicitly opened before they can be used.

The first ***fopen()*** function initializes the required device. If initialization takes some time, then you may notice that the first attempt to open a file on a device takes considerably longer than subsequent calls.

The direct access disk drivers (**lbahddrv.c** and **flopdrv.c**) that we provide are interrupt driven. For these devices, the task that sends the command then enters a wait state until the device signals that the command is completed. If an RTOS is being used with USFiles, then other tasks may run while the drive is executing the command. If USFiles is in stand-alone mode, then everything is put on hold until the drive is finished.

Function Names

The I/O functions (***fopen(), fread()***, etc.) are contained in the source files **streamio.c** and **fileio.c**. Each of these functions are defined in the source with a prefix of "***mt_***", i.e.; ***fopen()*** is defined as ***mt_fopen()***, etc. The header file **ussio.h** contains `#defines` to equate the names such as ***mt_fopen()*** to ***fopen()***. These `defines` can be switched off, which would make the names without the ***mt_*** prefix disappear. If you do this it will allow you to use I/O functions from another source (such as the library that came with your compiler), simultaneously with USFiles. In this case, ***fopen()*** would refer to the compiler library version of the function and ***mt_fopen()*** to the USFiles version of the function. In the remainder of this document we will refer to our functions as ***fopen(), fread()***, etc., interchangeably with the names with the ***mt_*** prefix. Keep in mind however that if you switch off the `#defines` in **ussio.h** you will be referring to them as ***mt_fopen()***, ***mt_fread()***, etc. exclusively. With the `#defines` switched off, the file descriptor type for USFiles becomes `MTFILE` rather than `FILE`.

You can switch off the `#defines` by including your compiler library `<stdio.h>` header file before you include the file **ussio.h**. Providing **stdio.h** defines `EOF`, the ***mt_*** defines will be omitted. The paths to `stdin`, `stdout`, and `stderr` are **NOT** automatically opened; they must be opened explicitly with a call to ***fopen()*** before they are used. If you are using your C compiler library I/O in conjunction with the USFiles I/O functions, you must not use ***mt_printf()***, or use the `stdin`, `stdout`, and `stderr` macros with the ***mt_*** functions, since the values defined in **stdio.h** will not be compatible with the USFiles values.

To use any of the stream I/O functions in your code, you must compile and link **streamio.c** and **fileio.c** along with the appropriate *file manager* and *driver* source files to your program. If additional devices are defined, they must be added as entries in the `device_tab` definition in **devtab.c**. All of the ***printf()*** functions are contained in the files **fprintf.c** and **sprintf.c**, and ***sscanf()*** is in the file **sscanf.c**. These files must be compiled and linked to your code to use these functions. If you are using VFAT to record long file names, then you may need to include **sprintf.c**.

The **makefile** provided will compile all necessary modules and build a library containing them if you make any of the test program targets. You need then only link this library with your application code to make the USFiles functions available to your code.

The ***fprintf()*** and ***scanf()*** functions will be generated as integer-only versions (not supporting floats and doubles) unless the label `PF_FLOATS` is defined when you compile these modules (**fprintf.c** and **sscanf.c**).

Using *errno*

Many of the I/O functions may set the variable *`errno`* to a non-zero value. When using an RTOS, we have to be careful how *`errno`* is defined. Chapter 6, *Supported RTOSes*, discusses how *`errno`* is implemented. USFiles operating in stand-alone mode defines *`errno`* by including the compiler library's **errno.h** file.

In either case, once an error code is written into *`errno`* by one of the functions returning an error, it is never cleared unless you clear it in the application. You will need to do this unless you are aborting your program on any error.

The error codes placed into *`errno`* by the USFiles functions are defined in **mtio.h**, and they are listed in Appendix G, *Error Codes*. Some of these may conflict with values defined in your compiler library header **errno.h** if you try to include both of these files in your application.

The library entries for functions that can set *`errno`* list the possible *`errno`* values. Since *`errno`* can be set at either the stream I/O or file manager level, we distinguish between the two. If you are using a file manager other than the PC file manager, then the possible *`errno`* values will differ. Because we are continually developing USFiles, do not regard these as comprehensive lists.

Atomic typedef Names

In addition to the ANSI C type definitions, USFiles specifies additional types (see **depends.h**).

Name	**Description**
byte	unsigned char (8 bits)
int16	signed 16-bit integer
int32	signed 32-bit integer
uint	unsigned integer
uint16	unsigned 16-bit integer
uint32	unsigned 32-bit integer

(The final five names are specified in the draft for ANSI C-99)

User Interface Library Functions

The functions are summarized by type, and then described individually in detail.

Function Summary

File Control Functions

mt_fopen	***Opens a file***
mt_fclose	***Closes a file***
mt_rename	***Renames a file***
mt_remove	***Removes a file***
mt_mkdir	***Creates a directory***
mt_readdir	Reads a directory entry
mt_rmdir	***Removes a directory***
mt_rewind	***Sets file pointer to start of file***
mt_fseek	***Positions file pointer to desired location***
mt_fsetpos	***Positions file pointer to desired location***
mt_ftell	***Reports position of file pointer***
mt_fgetpos	***Reports position of file pointer***

Writing Functions

mt_fwrite	Writes to a file
mt_fputc	Writes a single character to a file
mt_fputs	Writes a string to a file
mt_printf	Writes formatted output to `stdout`
mt_fprintf	Writes formatted output to a file
mt_sprintf	Writes formatted output to a string
mt_vsprintf	Writes formatted output to a string
mt_fflush	Flushes file's output buffer

Reading Functions

mt_fread	***Reads from a file***
mt_fgetc	***Reads a single character from a file***
mt_fgets	***Reads a string from a file***
mt_sscanf	***Converts a string according to specified format***

Error Reporting Functions

mt_feof	***Tests for end of file***
mt_ferror	***Returns file error condition***
mt_clearerr	***Clears file error condition***

Error Recovery Functions

invalidate_streams	***Invalidates all open streams for a given device***
otherFilesOpen	***Checks to see if there are open files on a device***
pcfm_invalidate_buffers	***Invalidates all buffers for a given device***

File Time Functions

getf_date	***Returns file modification date***
getf_day	***Returns file modification day***
getf_hour	***Returns file modification hour***
getf_min	***Returns file modification minute***
getf_month	***Returns file modification month***
getf_sec	***Returns file modification seconds***
getf_time	***Returns file modification time***
getf_year	***Returns file modification year***
mak_fdate	***Converts year, month, day to file date format***
mak_ftime	***Converts hours, minutes, seconds to file time format***
pcfm_chtime	***Changes time and date of file (specified by path)***
pcfm_chtimefp	***Changes time and date of file (specified by path)***

File Attribute Functions

getf_attrib	***Returns file attribute byte***
getf_size	***Returns file size***
pcfm_chmod	***Changes attributes of file (specified by path)***
pcfm_chmodfp	***Changes attributes of file (specified by pointer)***

Miscellaneous Functions

char2uni	***Converts ASCII and Shift-JIS to Unicode***
free_byte_cnt	***Returns number of unallocated bytes on drive***
free_clust_cnt	***Returns number of unallocated clusters on drive***
free_kb_cnt	***Returns number of unallocated kilobytes on drive***
getBigEnd16	***Reads 16-bit integer recorded in Big-Endian mode***
getBigEnd32	***Reads 32-bit integer recorded in Big-Endian mode***
getLitEnd16	***Reads 16-bit integer recorded in Little-Endian mode***
getLitEnd32	***Reads 32-bit integer recorded in Little-Endian mode***
pcfm_chvlabel	***Changes an existing volume label***
putBigEnd16	***Records 16-bit integer in Big-Endian mode***
putBigEnd32	***Records 32-bit integer in Big-Endian mode***
putLitEnd16	***Records 16-bit integer in Little-Endian mode***
putLitEnd32	***Records 32-bit integer in Little-Endian mode***
total_byte_cnt	***Returns total number of bytes on drive***
total_clust_cnt	***Returns total number of clusters on drive***
total_kb_cnt	***Returns total number of kilobytes on drive***
uni2char	***Converts Unicode to ASCII and Shift-JIS***

Function Descriptions

char2uni

Converts ASCII and Shift-JIS to Unicode.

```
uint16 char2uni(uint16 c);
```

c *Character to convert*

The variable `c` is either an 8-bit ASCII character or a Shift-JIS two-byte character. It will return the appropriate Unicode character for the character. If `c` is a character that we do not recognize, then the return value is `0xFFFD`.

NOTE: If you do not need to convert Kanji characters into Unicode, then be sure that the symbol `FAKEUNICODE` is defined as 1 (see *Configuring USFiles*). This will remove the Shift-JIS to Unicode conversion table and simplify the conversion process.

See also: ***uni2char***

Return Value

Unicode character corresponding to `c`

0xFFFD if character *c* is not supported

Example

```
/* MACRO: Returns true if c is first byte of double byte Shift-
JIS char */
#define is_dbc(c)  ((((byte)c >= 0x81) && ((byte)c <= 0x9f)) || \
                     (((byte)c >= 0xe0) && ((byte)c <= 0xfc)))

char filename[10];
char *fptr, namec;
uint16 unichar;

/* Get characters in filename */

fptr = filename;
namec = *fptr;
if(namec >= ‘ ‘){
   fptr++;        /* Next byte */
   if(is_dbc(namec))
      unichar = char2uni((uint16)((((uint16) namec) << 8)
                                    | (byte) (*fptr)));
   else
      unichar = char2uni((unsigned char)namec);
}
```

free_byte_cnt

Returns the number of unallocated bytes available on the drive.

```
uint32 free_byte_cnt(MTFILE *stream);
```

stream   pointer to the stream file descriptor

The number of bytes available to be allocated on the disk drive associated with *stream* is returned. If an error occurs, 0 is returned.

NOTE: If using the FAT32 addition to USFiles, be careful with this function. Since the number of FAT32 clusters can be a 32-bit number, converting that to the number of bytes might overflow a 32-bit unsigned integer.

See also: ***free_clust_cnt, free_kb_cnt, total_byte_cnt, total_clust_cnt, total_kb_cnt***

Return Value

Number of bytes available on disk.

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

ELOCKED	timeout while waiting for file system access

Example

```
FILE *fp;
uint32 freebytes;

  /* open for read/write */
  fp = mt_fopen(“A:\file1”, “r+b”);
  freebytes = free_byte_cnt(fp);
```

free_clust_cnt

Returns the number of unallocated clusters available on the drive.

```
uint32 free_clust_cnt(MTFILE *stream);
```

`stream` pointer to the stream file descriptor

The number of clusters available to be allocated on the disk drive associated with `stream` is returned. If an error occurs, 0 is returned.

See also: ***free_byte_cnt, free_kb_cnt, total_byte_cnt, total_clust_cnt, total_kb_cnt***

Return Value

Number of clusters available on disk.

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

ELOCKED	timeout while waiting for file system access

Example

```
FILE *fp;
uint32 clusters;

  /* open for read/write */
  fp = mt_fopen(“A:\file1”, “r+b”);
  clusters = free_clust_cnt(fp);
```

free_kb_cnt

Returns the number of unallocated kilobytes available on the drive.

```
uint32 free_kb_cnt(MTFILE *stream);
```

`stream` pointer to the stream file descriptor

This function returns the number of kilobytes available to be allocated on the disk drive associated with `stream`. If an error occurs, 0 is returned.

See also: ***free_byte_cnt, free_clust_cnt, total_byte_cnt, total_clust_cnt, total_kb_cnt***

Return Value

Number of bytes available on disk.

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

ELOCKED	timeout while waiting for file system access

Example

```
FILE *fp;
uint32 freekb;
  /* open for read/write */
  fp = mt_fopen(“A:\file1”, “r+b”);
  freekb = free_kb_cnt(fp);
```

getBigEnd16

Reads 16-bit integer recorded in Big-Endian mode.

```
uint16 getBigEnd16(byte **pos);
```

pos	*address of pointer indicating start of Big-Endian integer*

The routine ***getBigEnd16()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 16-bit integer.

See also: ***getBigEnd32, getLitEnd16, getLitEnd32, putBigEnd16, putBigEnd32, putLitEnd16, putLitEnd32***

Return Value

Unsigned 16-bit integer

Example

```
byte buffer[512], *bp;
uint16 number;

/* Point to beginning of 16-bit Big-Endian integer */
bp = &buffer[10];
number = getBigEnd16(&bp);
/* bp will now be at &buffer[12] */
```

getBigEnd32

Reads 32-bit integer recorded in Big-Endian mode.

```
uint32 getBigEnd32(byte **pos);
```

pos	*address of pointer indicating start of Big-Endian integer*

The routine ***getBigEnd32()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 32-bit integer.

See also: ***getBigEnd16, getLitEnd16, getLitEnd32, putBigEnd16, putBigEnd32, putLitEnd16, putLitEnd32***

Return Value

Unsigned 32-bit integer

Example

```
byte buffer[512], *bp;
uint32 number;

/* Point to beginning of 32-bit Big-Endian integer */
bp = &buffer[10];
number = getBigEnd32(&bp);
/* bp will now be at &buffer[14] */
```

getLitEnd16

Reads 16-bit integer recorded in Little-Endian mode.

```
uint16 getLitEnd16(byte **pos);
```

pos *address of pointer indicating start of Little-Endian integer*

The routine ***getLitEnd16()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 16-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd32, putBigEnd16, putBigEnd32, putLitEnd16, putLitEnd32***

Return Value

Unsigned 16-bit integer

Example

```
byte buffer[512], *bp;
uint16 number;

/* Point to start of 16-bit Little-Endian integer */
bp = &buffer[10];
number = getLitEnd16(&bp);
/* bp will now be at &buffer[12] */
```

getLitEnd32

Reads 32-bit integer recorded in Little-Endian mode.

```
uint32 getLitEnd32(byte **pos);
```

pos *address of pointer indicating start of Little-Endian integer*

The routine ***getLitEnd32()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 32-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd16, putBigEnd16, putBigEnd32, putLitEnd16, putLitEnd32***

Return Value

Unsigned 32-bit integer

Example

```
byte buffer[512], *bp;
uint32 number;

/* Point to start of 32-bit Little-Endian integer */
bp = &buffer[10];
number = getLitEnd32(&bp);
/* bp will now be at &buffer[14] */
```

getf_attrib

Returns attribute byte of file.

```
byte getf_attrib(MTFILE *stream);
```

stream *pointer to the stream file descriptor object*

The ***getf_attrib()*** function accesses the attribute byte of an open file's directory entry. The significant bits in the attribute byte for the FAT file system are:

0 Read Only	(`FA_RDONLY`)
1 Hidden File	(`FA_HIDDEN`)
2 System File	(`FA_SYSTEM`)
3 Volume Label	(`FA_LABEL`)
4 Directory	(`FA_DIR`)
5 Archive	(`FA_ARCH`)

See also: ***pcfm_chmod, pcfm_chmodfp***

Return Value

Attribute byte

`0xf`	failure

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte att;
att = getf_attrib(fp);
if( att & FA_RDONLY )
   /* File is read only */
```

getf_date

Returns file modification date.

```
uint16 getf_date(MTFILE *stream);
```

stream *pointer to the stream file descriptor object*

The ***getf_date()*** function accesses the modification date of an open file's directory entry. The date format for the FAT file system combines the year, month, and day in one 16-bit entry. To retrieve each part of the date separately, use the functions ***getf_year***, ***getf_month***, and ***getf_day***.

See also: ***mak_fdate, getf_year, getf_month, getf_day***

Return Value

File modification date

0	failure

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
uint16 date;
date = getf_date(fp);
if( (date & 0x1f) == 10 )
  /* If the modification day is the 10th */
```

getf_day

Returns day file was modified.

```
byte getf_day(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_day()*** function returns the day of the month on which the file was last modified. The day will range from 1 to 31

See also: ***mak_fdate, getf_year, getf_month, getf_date***

Return Value

File modification day (1..31)

`0`	failure

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte day;

day = getf_day(fp);
if( day == 27 )
  /* If the modification day is the 27th */
```

getf_hour

Returns hour file was modified.

```
byte getf_hour(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_hour()*** function returns the hour of the day in which the file was last modified. The hour will range from 0 to 23

See also: ***mak_ftime, getf_min, getf_sec, getf_time***

Return Value

File modification hour (0..23)

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte hour;

hour = getf_hour(fp);
if( hour == 1 )
  /* If the modification hour is 1 */
```

getf_min

Returns minute file was modified.

```
byte getf_min(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_min()*** function returns the minute in the hour in which the file was last modified. The minute will range from 0 to 59.

See also: ***mak_ftime, getf_hour, getf_sec, getf_time***

Return Value

File modification minute (0..59)

errno **Value**

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte minute;

minute = getf_min(fp);
if( minute == 30 )
  /* If the modification minute is 30 */
```

getf_month

Returns month file was modified.

```
byte getf_month(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_month()*** function returns the month in which the file was last modified. The month will range from 1 to 12.

See also: ***mak_fdate, getf_year, getf_day, getf_date***

Return Value

File modification month (1..12)

`0`	failure

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte month;

month = getf_month(fp);
if( month == 6 )
  /* If the modification month is June */
```

getf_sec

Returns seconds at which file was modified.

```
byte getf_sec(MTFILE *stream);
```

stream   *pointer to the stream file descriptor object*

The ***getf_sec()*** function returns the number of seconds in the minute in which the file was last modified. The seconds value will range from 0 to 59.

NOTE: For a file recorded in the FAT file system, the seconds value is stored in 2-second increments.

See also: ***mak_ftime, getf_hour, getf_min, getf_time***

Return Value

File modification seconds (0..58)

errno Value

Stream I/O

EBADFP bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
byte secs;

secs = getf_secs(fp);
if( secs == 22 )
  /* If modified at second 22 of the minute */
```

getf_size

Returns file size.

```
uint32 getf_size(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_size()*** function returns the size of an open file. If the file size cannot be determined, then a file size of zero will be reported.

Return Value

File size

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
uint32 size, max_size;

max_size = free_byte_cnt(fp);
size = getf_size(fp);
if( size > max_size )
  /* Cannot store file copy */
```

getf_time

Returns time at which file was modified.

```
uint16 getf_time(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_time()*** function returns the file's last modification time. For the FAT file system, the 16-bit value has a combination of hour, minute, and seconds at which the file was modified. To get each field separately, use the functions ***getf_hour(), getf_min(), getf_sec()***.

CAUTION: This function cannot be used by USFiles for CD-ROM.

See also: ***mak_ftime, getf_hour, getf_min, getf_sec***

Return Value

File modification time

0	failure

errno Value

Stream I/O

`EBADFP`	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
uint16 time;

time = getf_time(fp);
if( (time & 0x1f) == 15 )
   /* If modified during the 15th 2-second interval */
```

getf_year

Returns year when file was last modified.

```
uint16 getf_year(MTFILE *stream);
```

stream	*pointer to the stream file descriptor object*

The ***getf_year()*** function returns the year in which the file was last modified. For the FAT file system the year ranges from 1980 to 2107.

See also: ***mak_fdate, getf_month, getf_day, getf_date***

Return Value

File modification year (1980..2107, for FAT file)

0	failure

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

None

Example

```
MTFILE *fp;
uint year;

year = getf_year(fp);
if( year == 1997 )
   /* If modified during 1997  */
```

invalidate_streams

Invalidates all streams open on a device.

```
int invalidate_streams(DEVICE *devp);
```

devp	*pointer to device*

The ***invalidate_streams()*** function is provided for error recovery purposes. This function will close all streams for the device specified.

See also: ***otherFilesOpen, pcfm_invalidate_buffers***

Return Value

0	successful completion
`EOF`	error occurred, check *`errno`*

errno Value

Stream I/O

`EMEMERR`	memory release error
`EBADARG`	*`devp`* is `NULL` (only if `USS_SIO_PCHK` is set to 1)

PC File Manager

None

Example

```
DEVICE *devp;

/* Disk has changed */
pcfm_invalidate_buffers(devp);
if(otherFilesOpen(devp))
    invalidate_streams(devp);
else
    /* No open files, so ignore error */
```

mak_fdate

Converts the year, month, and day to DOS date format.

```
uint16 mak_fdate(uint year, byte month, byte day);
```

year	*year in which file was modified (1980 – 2107)*
month	*month in which file was modified (1 – 12)*
day	*day in which file was modified (1 – 31)*

The DOS date format is produced based on the year, month, and day provided. This function is implemented as a macro in **mtio.h**.

CAUTION: This function cannot be used by USFiles for CD-ROM.

Return Value

File modification date in DOS format

Example

```
uint16 dos_date;

/* Convert June 27, 1997 to DOS date */
dos_date = mak_fdate(1997, 06, 27);
```

mak_ftime

Converts the hour, minute, and second to DOS time format.

```
uint16 mak_ftime(byte hour, byte minute,
                 byte second);
```

hour	*hour in which file was modified (0 –23)*
minute	*minute in which file was modified (0 – 59)*
second	*second in which file was modified (0 – 59)*

The DOS time format is produced based on the hour, minute, and second provided. This function is implemented as a macro in **mtio.h**.

CAUTION: This function cannot be used by USFiles for CD-ROM.

Return Value

File modification date in DOS format

Example

```
uint16 dos_time;

/* Convert 11:12:30 am to DOS time */
dos_time = mak_ftime(11, 12, 30);
```

mt_clearerr

Clears the end-of-file and error indicators.

```
void clearerr(MTFILE *stream);
```

stream	pointer to the stream file descriptor

The end-of-file and error indicators associated with *stream* are cleared.

See also: ***rewind, feof, ferror***

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

None

Example

```
FILE *fp;
  /* open for read/write */
  fp = mt_fopen(DEVICE_0, “r+b”);
  mt_clearerr(fp);
```

mt_fclose

Closes an open path to a stream.

```
int mt_fclose(MTFILE *stream);
```

`stream`	pointer to the stream file descriptor object

The ***mt_fclose()*** function returns zero if the `stream` was successfully closed, or `EOF` if any errors were detected. For ***mt_fclose()*** to successfully complete, the stream must be open and accessible by the task making the ***mt_fclose*** call. The stream output buffer will be flushed before the device is closed. The device interrupts are disabled when the device is closed, and any tasks waiting for the device I/O to complete will be reactivated.

Return Value

0	file successfully closed
`EOF`	error (file not open, or not in possession of task making the call)

errno Value

Stream I/O

EBADFP	bad file pointer
EMEMERR	memory release error

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EWRTPRT	attempted write to write-protected disk
driver error	

Example

```
MTFILE *fp;

  fp = mt_fopen("COM1", "r+b");
              /* open for read/write */
  {     /* processing */ }
  if( mt_fclose(fp) ){
        if(errno == EMEMERR)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

NOTE: If USFiles is operating with MultiTask! and the task in possession of an open stream or streams dies, or is killed (by ***klltsk()***), all streams in the possession of that task are closed.

mt_feof

Tests for end-of-file condition.

```
int mt_feof(MTFILE *stream);
```

stream	pointer to the stream file descriptor object

The ***mt_feof()*** function returns non-zero if the *stream* is at end-of-file. Once the EOF flag is set it will only be cleared by calls to ***rewind()***, ***clearerr()***, or successful calls to ***fseek()*** or ***fsetpos()***, which is in accordance with the ANSI C 99 specification. Of course, closing the stream makes the EOF flag invalid as well.

NOTE: This function is implemented as a macro in **ussio.h**.

Return Value

0	file is not at end
!0	file is positioned at end

errno Value

Stream I/O

`EBADFP`	fp is not valid (only if `USS_SIO_PCHK` is set to 1)

Example

```
MTFILE *fp;
int i;

  /* open for read/write */
  fp = mt_fopen(“a:file1”, “r+b”);
  while( !mt_feof(fp) ){
      i = mt_fgetc(fp);
      /* etc. */
```

mt_ferror

Returns the file error indicator.

```
int mt_ferror(MTFILE *stream);
```

`stream`	pointer to the stream file descriptor object

The ***mt_ferror()*** function returns non-zero if the error indicator is set for the stream. The error indicator will be cleared by a ***rewind()***, or ***clearerr()*** function, or by closing the stream. This function is implemented as a macro in **ussio.h**.

NOTE: ANSI C does not specify under what conditions the error indicator for the stream is set. In the current implementation, only the driver level ever sets the error indicator. You should generally rely on the return status of each function to determine errors, and the value of *`errno`*. Note also that *`errno`* is not cleared by any ANSI C function. Once it is set non-zero, it is up to you to clear it.

Return Value

0	No error for file

> 0	Error occurred on file, see Appendix G, *Error Codes*
`EOF`	File pointer not valid (only if `USS_SIO_PCHK` is 1)

errno Value

Stream I/O

`EBADFP`	fp is not valid (only if `USS_SIO_PCHK` is set to 1)

Example

```
MTFILE *fp;
int i;

  /* open for read/write */
  fp = mt_fopen(DEVICE_0, “r+b”);
  while( !mt_feof(fp) ){
      i = mt_fgetc(fp);
      if( mt_ferror(fp) )
          report_error(“File error occurred”);
      /* etc. */
```

mt_fflush

Flushes the output buffer of a stream.

```
int mt_fflush(MTFILE *stream);
```

`stream` pointer to the stream file descriptor object

The ***mt_fflush()*** function will cause the calling task to wait until any remaining data in the stream output buffer has been transmitted to the port or file. If the specified `*stream == NULL` then all open streams are flushed.

A call to ***mt_fflush()*** will update the specified file's directory entry as well as flush data to disk. This allows a file on disk to be consistent after every ***mt_fflush()*** call. This behavior is different from DOS, which does not update the directory entry until the file is closed.

Return Value

0	file successfully flushed
`EOF`	error (file not open, or not accessible from the task making the call)

errno Value

Stream I/O

EBADFP	bad file pointer
EUNINIT	file not initialized

PC File Manager

None

Example

```
MTFILE *fp;

  if( mt_fflush(fp) ){
        if(errno == EUNINIT)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_fgetc

Gets a character from a stream.

```
int mt_fgetc(MTFILE *stream);
```

`stream`	pointer to the stream file descriptor

The ***mt_fgetc()*** function obtains the next character, as an unsigned `char` converted to an `int`, from the input stream pointed to by *`stream`*.

Return Value

character	the next character from the stream
`EOF`	error (stream not opened or not in possession of calling task)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for read

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT sector encountered
EATEOF	at end of file
driver error	

Example

```
MTFILE *fp;    /* open stream pointer */
int     c;

  c = mt_fgetc(fp); /* get character */
  if( c == EOF ){
        if(errno == EACCESS)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_fgetpos

Gets a stream’s current position.

```
int mt_fgetpos(MTFILE *stream, fpos_t *position);
```

stream	pointer to I/O stream
position	position returned by function

The ***mt_fgetpos()*** function fills `position` with a value representing the current position of the file pointer for `stream`. This is usually the byte number from the beginning of the file. In the case of a file open in `text` mode this may not be the same as the actual number of bytes you have read from the file. The `position` returned by ***mt_fgetpos()*** should be used as an argument to ***mt_fsetpos()*** to reposition a file to a former location.

NOTE: On an error condition, `errno` is also set to the return value.

See also: ***mt_fsetpos***

Return Value

0	success
EBADFP	invalid file pointer
EUNSUP	device does not support function

`EBADARG`	*`position`* is `NULL` (only if `USS_SIO_PCHK` is 1)

errno Value

Stream I/O

EBADFP	bad file pointer
EBADARG	*position* is NULL (only if USS_SIO_PCHK is 1)

PC File Manager

None

Example

```
MTFILE *fp;    /* open stream pointer */
fpos_t position;
int i;
double x;

  status = mt_fgetpos(fp,&position);
```

mt_fgets

Gets a string from a stream.

```
char *mt_fgets (char *s, int n, MTFILE *stream);
```

`s`	pointer to character array of at least size *`n`*
`n`	maximum number of characters to read plus one (for the null string terminator)
`stream`	pointer to the stream file descriptor

The ***mt_fgets()*** function reads at most one less than the number of characters specified by *`n`* from the stream pointed to by *`stream`* into the array pointed to by *`s`*. No additional characters are read after the new-line character (which is retained). A null character is written immediately after the last character read into the array.

NOTE: The new-line character is defined in **userio.h** as `EOL_CHAR` and is not necessarily the same as “`\n`” produced by your C compiler. The default value of `EOL_CHAR` is the ASCII carriage return for `sfm` and pipe devices, and “`\n`” for `pcfm` devices.

See also: ***mt_fputs***

Return Value

`s`	operation successful
`NULL`	error: Stream not open, or not in our possession

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for read
`EBADARG`	*`s`* is `NULL` or *`cnt`* < 1 (only if `USS_SIO_PCHK` is 1)

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EATEOF	at end of file
driver error	

Example

```
MTFILE *fp;   /* open stream pointer */
char buf[80];

    if( mt_fgets(buf, 80, fp) ){
          /* we have string */
    }else{
          if(errno == EACCESS)
                /* Handle this error */
          else
                /* Handle other errors */
    }
```

mt_fopen

Opens a path to a stream.

`MTFILE *mt_fopen(const char *name, const char *mode);`

name	pathname to `device:[file]`
mode	type of access permitted

The ***mt_fopen()*** function opens a path to name and returns a pointer to the MTFILE structure controlling the stream. The device component part of name must appear in the device table (**device_tab**). The additional name components if any must conform to the rules for the type of device opened. If the operation fails, a null pointer is returned.

When using short (8.3) file names, USFiles will truncate any long names to become 8.3 names. If there are more than 8 characters preceding the extension, the name will be truncated to 8, unless the 8th byte is the first byte of a two-byte character. In this case, the name will be truncated to use the first 7 bytes only. The same rule holds for extensions that exceed 3 characters. If a file name has multiple occurrences of '.', the last one found marks the extension. For example, the file name "**long.tmp.txt.dat**" is recorded as "**long.dat**".

NOTE: Although * and ? are characters not allowed in file names, USFiles will not reject them. They are recognized as wild card characters, but USFiles does not support matching file names with them. Please avoid using these unless you implement wild card pattern matching.

The mode string specifies the type of access requested as follows:

“r”	open text mode for reading

“w”	create text mode for writing
“a”	append (open/create for write at EOF)
“rb”	open binary mode for reading
“wb”	create or truncate for binary write
“ab”	append binary (open/create for write at EOF)
“r+”	open for update (read and write)
“w+”	truncate or create for update
“a+”	append (update at EOF)
“r+b”	open binary mode for update
“w+b”	truncate or create for binary update
“a+b”	append; open/create for binary update at EOF
“d”	open directory (USFiles only)

CAUTION: USFiles has an incompletely implemented append mode. When opening a file in append mode, USFiles will set the initial position to the end of the file. This is the extent of supporting append mode. USFiles does <u>not</u> force the file position to EOF before any write, as is required by the ANSI specification.

See also: ***mt_fclose***

Return Value

`MTFILE *`	the file descriptor pointer to be used as a “handle” argument for all subsequent I/O calls for the device.
`NULL`	unable to open the device, possibly because the device name was invalid, or the device is already in the possession of another task. The value of the global *errno* may contain additional error status. See **mtio.h** for the error codes returned in *errno*.

errno Value

Stream I/O

EBADARG	bad value in *mode*
ENOPATH	device not found (see PC File Manager *errno* codes)
ECAPERR	device does not support open for *mode* specified
ENMFILE	no entries available in open streams array
EBADARG	*name* is NULL, empty, or filled with blanks

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path not found (see Stream I/O *errno* codes)
ENOENT	file not found in directory
ENOTDIR	path contains a file name (instead of a directory)
EACCESS	trying to access a file as a directory or vice versa
ERDONLY	opening read only file for write
EBIGPATH	path length exceeds VFAT restrictions

ERDFULL	root directory is full
EISOPEN	attempted to open a file multiple times, when not all opens are for reading only
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;

  fp = mt_fopen(“A:\\temp.txt”, “r+b”);
              /* open for r/w binary mode */
  if( !fp ){
        if(errno == ECAPERR)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

mt_fprintf

Writes formatted output to a stream.

```
int mt_fprintf(MTFILE *stream, const char *format, ...);
```

stream	the output stream file descriptor pointer
format	format specification string
...	arguments to be formatted for output

The ***mt_fprintf()*** function writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored. The ***fprintf*** function returns when the end of the `format` string is encountered.

The format shall be a multi-byte character sequence composed of zero or more directives. A directive is one or more white-space characters, ordinary characters (not %) which are copied unchanged to the output stream, or a conversion specification. A conversion specification is introduced by the character %, and has this format:

```
%[flags][width][precision][mod]type
```

flags	`-`	Left-justify result
	`+`	Always prefix with + or -
	`space`	Prefix with a blank if non-negative
	`#`	Alternate form conversion
width	*n*	Prints at least *n* characters, pad with spaces
	`0`*n*	Prints at least *n* characters, pad with zeros
	`*`	The next argument, which must be type `int`; it is consumed from the `args` list and used as the width specifier.
precision		(default) =1 for `d,i,o,u,x,X` =6 for `e,E,f`
	`.0`	No decimal point for `e,E,f`
	`.`*n*	*n* decimal places or characters are printed
mod	`h`	Short `int` for types: `d,i,o,u,x`
	`l`	Long `int` for types: `d,i,o,u,x` double for types: `e,f,g`
	`L`	Same as `l`
type	`c`	`Int` converted to unsigned `char` printed
	`d`	Signed decimal `int`
	`e`	Signed exponential
	`f`	Signed floating point
	`g`	Same as `e` or `f` based on value and precision
	`i`	Signed decimal `int`
	`n`	Argument is a pointer to `int` into which is written the number of `chars` written to

	stream so far
o	Octal unsigned int
p	Pointer
s	String
u	Decimal unsigned int
x	Hexadecimal unsigned int (a..f)
X	Hexadecimal unsigned int (A..F)

Return Value

+n	the number of characters written
EOF	output error (stream not open or not accessible)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for write

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;    /* open stream pointer */
int count,i,j;
double x,y;

  count = mt_fprintf(fp,”i = %d,  (%04X hex),
                    x=%e\r\n”,i,i,x);
  if(count == EOF){
        if(errno == EACCESS)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_fputc

Writes a character to a stream.

```
int mt_fputc(int c, MTFILE *stream);
```

`c`	the character to be output
`stream`	pointer to the stream file descriptor object

The ***mt_fputc()*** function writes the character specified by `c` (converted to an unsigned `char`) to the output stream pointed to by `stream`. The ***mt_fputc()*** function returns the character written unless an error occurs, in which case it returns `EOF`.

See also: ***mt_fgetc***

Return Value

character	the character written to the stream
`EOF`	error (stream not opened or not in possession of calling task)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for write

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;   /* open stream pointer */
int     c;

  if( mt_fputc(c,fp) == EOF ){
        if(errno == EACCESS)
```

```
                /* Handle this error */
          else
                /* Handle other errors */
    }
```

mt_fputs

Writes a string to a stream.

```
int mt_fputs(const char *s, MTFILE *stream);
```

s	pointer to the string to write
stream	pointer to the stream file descriptor

The ***mt_fputs()*** function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null of `s` is not written. The number of characters written is returned unless a write error occurs, in which case `EOF` is returned. (Note that the ANSI C standard specifies only that a non-negative value is returned in the normal case.)

See also: ***mt_fgets***

Return Value

count	the number of characters written
`EOF`	error (stream not opened for write or not in possession of calling task)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for write

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full

driver error

Example

```
MTFILE *fp;    /* open stream pointer */

  if( mt_fputs("Hello there",fp) == EOF ) ){
        if(errno == EACCESS)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_fread

Reads bytes from a stream.

```
size_t  mt_fread(void *ptr, size_t size, size_t nmemb,
                 MTFILE *stream);
```

ptr	pointer to the buffer to receive data
size	the size in bytes of each element
nmemb	the number of elements
stream	the stream object pointer

The ***mt_fread()*** function attempts to read *nmemb* elements of *size* bytes into the array pointed to by *ptr*, from *stream*. The actual number of elements read is returned. Note that the number of elements returned will be equal to *nmemb* unless the EOF is reached or some error occurs.

One method of reading blocks of data is to call ***mt_fread(****`buffer, blocksize, numblocks, fp`****)**, but we recommend calling ***mt_fread(****`buffer, 1, blocksize*numblocks, fp`****)**. In the first case, the return value will be the number of blocks read. If some part of a block is read, but not the entire block, then there is no way to know how many additional bytes were read, and therefore, the file position is unknown. The second method described reports how many bytes are read.

NOTE: It is not recommended that ***mt_fread()*** be used for files opened in text mode. It will not cause difficulties, but it may return unexpected values.

See also: ***mt_fwrite***

Return Value

+n	the number of elements actually read
0	no bytes read, typically indicating an error

errno Value

Stream I/O

`EBADFP`	bad file pointer
`EACCESS`	file not opened for read
`EBADARG`	*`ptr`* is `NULL` or *`nmemb`* `*` *`size`* exceeds `UINT_MAX` (only if `USS_SIO_PCHK` is 1)

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EATEOF	at end of file

driver error

Example

```
MTFILE *fp;    /* open stream pointer */
char buf[80];

  if( mt_fread(buf, 1, 80, fp) != 80 ) ){
        if(errno == EACCESS)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_fseek

Repositions a file pointer.

`int mt_fseek(MTFILE *stream, long offset, int location);`

stream	pointer to I/O stream
offset	number of bytes to offset from *location* to determine new file pointer position
location	file position from which to add *offset* SEEK_SET (0) - Beginning of file SEEK_CUR (1) - Current file pointer position SEEK_END (2) - End of file

The ***mt_fseek()*** function repositions the file pointer for *stream* by *offset* bytes from *location*. If the stream is text mode, offset should be 0 or the value returned by ***mt_ftell()***. The value in *location* should be SEEK_SET for beginning of file, SEEK_CUR for current file pointer position, or SEEK_END for end of file. A successful seek will clear the EOF flag, in accordance with the ANSI C 99 specification.

NOTE: USFiles supports seeking past EOF in a file that is opened for write access. This is only allowed if SEEK_END is specified as the *location*.

See also: ***mt_ftell***

Return Value

0	file pointer successfully repositioned
errno	*see* errno *values below*

errno Value

Stream I/O

EBADFP	bad file pointer
EBADARG	bad value of *location*
EBADPOS	illegal *offset*

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;    /* open stream pointer */
int status;

/* Note: second arg below is 30”ell” */
  status = mt_fseek(fp,30l,SEEK_SET);
```

mt_fsetpos

Sets a stream’s current position (byte offset from beginning of file).

```
long int mt_fsetpos(MTFILE *stream, const fpos_t *pos);
```

stream	pointer to I/O stream
pos	new position to set

The ***mt_fsetpos()*** function sets the file pointer associated with *`stream`* to the new position *`pos`*. The new position is the value obtained by a previous call to ***mt_fgetpos()*** on that stream. The reason for the existence of ***fgetpos*** and ***fsetpos*** (in addition to ***fseek***) is that if you want to position to a file in text mode, you cannot necessarily find a position by counting the characters you have written out, since text mode translation may change that number. In this case you can only use ***fgetpos*** to find a current position and then return there later with ***fsetpos***.

See also: ***mt_fgetpos***

Return Value

0	success
errno	*see* errno *values below*

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;   /* open stream pointer */
fpos_t offset;
int i;
```

```
      status = mt_fsetpos(fp, &offset);
```

mt_ftell

Gets current file position.

```
long int mt_ftell(MTFILE *stream)
```

stream	pointer to I/O stream

The ***mt_ftell()*** function returns the value of the file pointer for *stream*. The file pointer contains a value that specifies the current position of the file as the byte offset from the beginning of the file.

See also: ***mt_fseek***

Return Value

offset	value of file pointer on success byte offset from beginning of file
-1	*errno* set positive on failure

errno Value

Stream I/O

EBADFP	bad file pointer
EUNSUP	function unsupported for this file

PC File Manager

None

Example

```
MTFILE *fp;    /* open stream pointer */
long offset;
int i;
double x;

  offset = mt_ftell(fp);
  if(offset == -1L){
        if(errno == EUNSUP)
              /* Handle this error */
        else
              /* Handle other errors */
  }
  status = mt_fseek(fp,offset,SEEK_SET);
```

mt_fwrite

Writes to a stream.

```
size_t mt_fwrite(void *ptr, size_t size, size_t nmemb,
                  MTFILE *stream);
```

ptr	pointer to the data to write
size	the size of each data item
nmemb	the number of data items
stream	pointer to the stream file descriptor

The ***mt_fwrite()*** function writes, from the array pointed to by *ptr*, up to *nmemb* elements of *size* bytes each, to *stream*. The number of elements actually written is returned, which will be less than *nmemb* only if an error occurred. If the stream is not open or not accessible to the calling task, `EOF` will be returned.

See also: ***mt_fread***

Return Value

count	the number of items written

`EOF`	error (stream not opened for write or otherwise not accessible by the calling task)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for write

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EDSKFUL	disk is full
driver error	

Example

```
MTFILE *fp;    /* open stream pointer */
int count;
int data[10];

  count = mt_fwrite(data, sizeof(int), 10, fp);
  if( count < 10 ) {
        if(errno == EACCESS)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_mkdir

Creates a new directory.

```
int mt_mkdir(const char *path)
```

path	the complete pathname of the directory to create

The ***mt_mkdir()*** function creates a new directory from the given pathname *path*.

Return Value

0	successful
EOF	error, and global variable *errno* set to a non-zero error code (*errno* codes are defined in **mtio.h**)

errno Value

Stream I/O

EEXIST	directory already exists
ECAPERR	device cannot be written to
ENOPATH	part of directory path not found
ENMFILE	no available entries in open streams array
EMEMERR	memory release error

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes

ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path not found (see Stream I/O *errno* codes)
ENOTDIR	path contains a file name (instead of a directory)
EBIGPATH	path length exceeds VFAT restrictions
ERDFULL	root directory is full
EDSKFUL	disk is full
EWRTPRT	trying to write to a write-protected disk
driver error	

Example

```
int status

  status = mt_mkdir(“a:\\thisdir/thatdir/newdir”);
  if( status ){
        if(errno == EEXIST)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

mt_printf

Writes formatted output to `stdout` stream.

```
int mt_printf(const char *format, ...);
```

`format`	format specification string
`...`	arguments to be formatted for output

The ***mt_printf()*** function writes output to the `stdout` stream, under control of the string pointed to by *`format`* that specifies how subsequent arguments are converted for output. The ***mt_printf*** function behaves exactly like an ***mt_fprintf*** call with `stdout` specified as the stream, and indeed it is implemented as this. See ***mt_fprintf*** for further information on the *`format`* specification. The definition of `stdout` is in the file **ussio.h** and may be modified by the user to be any device. Note that `stdout` is not automatically opened.

Return Value

+n	the number of characters written
`EOF`	output error (stream not open or not the owner)

errno Value

Stream I/O

EBADFP	bad file pointer
EACCESS	file not opened for write

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered

EDSKFUL	disk is full

driver error

Example

```
FILE *fp;      /* open stream pointer */
int count;
int i,j;
double x,y;

  count = mt_printf(“i = %d,  (%04X hex),
                  x=%e\r\n”,i,i,x);
  if(count == EOF){
        if(errno == EBADFP)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

mt_readdir

Reads a directory entry.

```
int mt_readdir(MTFILE *stream, MT_DIRENT *dirp);
```

stream	Pointer to I/O stream (open directory)
dirp	Pointer to directory entry structure

The ***mt_readdir()*** function reads the next directory entry in the opened directory, and stores the data in the structure pointed to by `dirp`. When the end of the directory is reached the end-of-file indicator will be set. The `MT_DIRENT` structure is defined in **ussio.h** as:

```
typedef struct mt_dirent{
   uint32   size;            /* File size */
   uint16   year;            /* File mod year */
   byte           month,           /* File mod month */
            day,             /* File mod day */
            hour,            /* File mod hour */
            minute,           /* File mod minute */
            second,           /* File mod second */
            dir_flag;         /* =1 if directory */
#if VFAT
   byte       name[_MAX_FILENAME+1];  /* VFAT file name */
#else
   byte       name[13];               /* 8.3 file name */
#endif
}MT_DIRENT;
```

Return Value

0	successful
EOF	entry not read; *errno* or end of directory reached

errno Value

Stream I/O

EBADFP	bad file pointer
ENOTDIR	file specified is not a directory

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered
EATEOF	positioned at end of file
driver error	

NOTE: In a multitasking system, be careful not to modify a directory while you are reading its entries. This might result in an incorrect listing.

Example

```
MT_DIRENT entry;
int status;
  iprintf("\nListing Root Directory on A:\n");
  fp = mt_fopen("A:","d");
  while ( !mt_feof(fp) ) {
        if(!mt_readdir(fp, &entry))
              iprintf(" %s\n",entry.name);
        else if(errno != 0)
              iprintf("mt_readdir errno = %d",errno);
  }
  mt_clearerr(fp);        /* Clear EOF indicator */
  mt_fclose(fp);
```

mt_remove

Deletes a file.

```
int mt_remove(const char *pathname);
```

pathname	the complete pathname to the file

The ***mt_remove()*** function deletes a file specified by *pathname*. A complete pathname including the device name must be specified.

Return Value

0	successful
EOF	error, with the global variable *errno* set to the specific error code

errno Value

Stream I/O

ECAPERR	device cannot be written to
ENOPATH	device not found

ENMFILE	no available entries in open streams array

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path not found (see Stream I/O *errno* codes)
ENOTDIR	path contains a file name (instead of a directory)
EBIGPATH	path length exceeds VFAT restrictions
ERDFULL	root directory is full
EWRTPRT	trying to write to a write-protected disk
driver error	

Example

```
if( mt_remove(“a:\\subdir\\thisfile.txt”) )
    printf(“errno = %d\n”, errno);
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

mt_rename

Renames (or moves) a file or subdirectory.

```
int mt_rename(const char *oldname, const char *newname);
```

oldname	pathname to an existing file
newname	new pathname to give file

The ***mt_rename()*** function changes the name of the file `oldname` to `newname`. A complete pathname must be given for both, which must be on the same device (drive). Subdirectories can be renamed. The `newname` does not need to be in the same directory as `oldname`. The effect in this case is that of moving the file to the new directory (and possibly renaming it in the process). Attempting to rename a directory to be its own subdirectory is not allowed and generates an `EACCESS` error.

Return Value

0	successful
`EOF`	error, with the global variable `errno` set to the specific error code.

errno Value

Stream I/O

ECAPERR	device cannot be written to
ENOPATH	part of directory path not found
ENMFILE	no available entries in open streams array
EEXIST	file (or directory) with *newname* already exists
EWRGDEV	trying to rename file to a different device

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path not found (see Stream I/O *errno* codes)
ENOENT	file not found in directory
ENOTDIR	path contains a file name (instead of a directory)
ERDONLY	opening read only file for write
EBIGPATH	path length exceeds VFAT restrictions
ERDFULL	root directory is full
EDSKFUL	disk is full
EACCESS	attempting to rename a directory to its own subdirectory
EWRTPRT	trying to rename file on write-protected disk
driver error	

Example

```
     if( mt_rename(“a:\\file1.txt”,”a:\\file2.txt”) )
         printf(“errno = %d\n”, errno);
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

mt_rewind

Repositions file pointer to start of file.

```
void mt_rewind(MTFILE *stream);
```

stream	pointer to I/O stream

The ***mt_rewind()*** function will position the file pointer to the start of the file, and clear any end-of-file and error indicators associated with the stream.

See also: ***mt_fsetpos, mt_fseek***

Return Value

None

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
EBADFAT	bad FAT encountered driver error

Example

```
      MTFILE *fp;
      if(mt_feof(fp))
            mt_rewind(fp);
```

mt_rmdir

Removes (deletes) a subdirectory.

```
int mt_rmdir(const char *pathname);
```

pathname	the complete pathname of the directory to delete

The ***mt_rmdir()*** function removes the directory specified by `pathname` from the file system. The directory must be empty or an error is returned. An attempt to remove the root directory returns an error.

See also: ***mt_mkdir()***

Return Value

0	successful
`EOF`	error, and global variable *errno* set to a non-zero error code

errno Value

Stream I/O

ENOPATH	device not found
ENMFILE	no available entries in open streams array
EACCESS	*pathname* is a file, not a directory
ENOTMT	directory is not empty
EWRTPRT	trying to remove directory on write-protected disk

PC File Manager

ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path not found (see Stream I/O *errno* codes)
ENOTDIR	path contains a file name (instead of a directory)
EBIGPATH	path length exceeds VFAT restrictions
ERDFULL	root directory is full
EWRTPRT	trying to write to a write-protected disk
driver error	

Example

```
int status

  status = mt_rmdir(“a:\\thisdir/thatdir/newdir”);
  if( status ){
        if(errno == ENOTMT)
              /* Handle this error */
        else
              /* Handle other errors */
  }
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

mt_sprintf

Writes formatted output to a string.

```
int mt_sprintf(char *s, const char *format, ...);
```

s	pointer to string to receive output
format	format specification string
...	arguments to be formatted for output

The ***mt_sprintf()*** function writes output to the string pointed to by *s*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. The ***mt_sprintf()*** function behaves exactly like an ***mt_fprintf*** call except that the output is written to the string *s* rather than a stream.

See also: ***mt_fprintf*** for further information on the *format* specification.

Return Value

+n	the number characters written
EOF	output error (stream not open or not accessible)

errno Value

Stream I/O

EBADARG	*s* is NULL (only if USS_SIO_PCHK is 1)

Example

```
MTFILE *fp;    /* open stream pointer */
int count;
int i,j;
double x,y;
char *line[100];

   count = mt_sprintf(line,
          ”i = %d,  (%04X hex),x=%e\r\n”,i,i,x);
```

mt_sscanf

Converts a string, using the specified format.

```
int mt_sscanf(const char *s, const char *format,
              ...);
```

s	pointer to string containing input characters
format	format specification string
...	pointers to objects to receive input items

The ***sscanf()*** function reads input from the string pointed to by *s*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the `format`, the behavior is undefined. If the *format* is exhausted while arguments remain, the excess arguments are ignored.

The *format* shall be a multibyte character sequence composed of zero or more directives. A directive is one or more white-space characters, an ordinary character (neither a % nor a white-space), or a conversion specification. A conversion specification is introduced by the character % and has this format:

%[*flags*][*width*][*mod*]*type*

flags	*****	Suppresses assignment of next field

width	*n*	Maximum number of characters that will be read
mod	h	Short `int` for types: `d,i,o,u,x`
	l	Long `int` for types: `d,i,o,u,x` double for types: `e,f,g`
	L	Same as `l`
type	c	Single character
	d	Signed decimal `int`
	e	Signed exponential
	f	Signed floating point
	g	Same as `e` or `f` based on value and precision
	i	Signed decimal, octal, or hex `int` (e.g. `123, 0123, 0x123`)
	[abc]	Matches characters in set or ...
	[^ab]	Matches characters NOT in set
	n	`Int` to receive count of `chars` consumed so far
	p	Pointer
	s	String

Return Value

+n	the number of input items assigned {`0..`*n*}
`EOF`	failure

Example

```
MTFILE *fp;    /* open stream pointer */
char buf[80];
int count, arg[4];

    mt_fgets(buf, 80, fp);          /* read string */
    count = mt_sscanf(buf,”%d %d %d %d”,&arg[0],
                         &arg[1], &arg[2],&arg[3]);
```

mt_vsprintf

Writes formatted output to a string.

```
#include <stdarg.h>

int mt_vsprintf(char *s, const char *format,
                va_list arg);
```

s	pointer to string to receive output
format	format specification string
arg	list of arguments to be formatted for output

The ***mt_vsprintf()*** function is equivalent to ***mt_sprintf***, with the variable argument list replaced by *arg*, which shall have been initialized by the ***va_start*** macro (and possibly subsequent ***va_arg*** calls). The ***mt_vsprintf()*** function returns the number of characters written in the array, not counting the terminating null character.

Return Value

+n	the number characters written
EOF	output error (stream not open or not accessible)

errno Value

Stream I/O

EBADARG	*s* is NULL (only if USS_SIO_PCHK is 1)

Example

```
MTFILE *fp;    /* open stream pointer */
int count;
int i;
double x;
void *args[3];

   args[0] = &i;
   args[1] = &i;
   args[2] = &x;

   count = mt_vsprintf(line,”i = %d, (%04X hex),
                         x=%e\r\n”,&args[0]);
```

otherFilesOpen

Tests to see if files are open on the specified device.

```
int otherFilesOpen(DEVICE *devp);
```

devp	*pointer to device*

The ***otherFilesOpen()*** function is provided for error recovery purposes. If a disk change error is sensed, then we can call ***otherFilesOpen()*** to see if files on the device are open.

The ***mt_fopen()*** function sets a pointer in the device structure when a file is opened. This pointer is only used by stream devices and not by disk devices. Once ***pcfm_open()*** successfully opens the file, it clears this pointer. This is done to handle the case when a disk change is sensed during a file open.

Imagine the situation where you close all open files, change the disk, and then open a file on a new disk. The drive will have sensed that a disk has changed. If we simply scan to see if there are any files open on the disk, then we will find the file we are trying to open! By using the pointer mentioned above, we know that it is safe to continue with the file open. This is why the function name refers to “other” files. An examination of the code is useful for understanding this.

See also: ***pcfm_invalidate_buffers, invalidate_streams***

Return Value

0	no files open
1	files open on device

Example

```
DEVICE *devp;

/* Disk has changed */
pcfm_invalidate_buffers(devp);
if(otherFilesOpen(devp))
    invalidate_streams(devp);
else
    /* No open files, so ignore error */
```

pcfm_chmod

Changes file attributes (uses path).

```
int pcfm_chmod(const char *pathname, int attribute);
```

pathname	the pathname to the file
attribute	new file attribute value

The ***pcfm_chmod()*** function will change the *attribute* associated with the file specified by *pathname*. The attributes must be one or more of the following:

FA_NORMAL	Normal file (no attributes)
FA_RDONLY	Read-only file
FA_HIDDEN	Hidden file (does not affect accessibility)
FA_SYSTEM	System file
FA_ARCH	Archive bit (file changed since bit cleared)

See also: ***pcfm_chmodfp***

Return Value

+n	new *attribute*
EOF	failure

errno Value

Stream I/O

None

PC File Manager

ENMFILE	no entries available in open streams array
EACCESS	illegal attribute change
EBADARG	requested attribute is illegal
ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	part of directory path or device not found
ENOENT	file not found in directory
ENOTDIR	path contains a file name (instead of a directory)
EACCESS	trying to access a directory
EBIGPATH	path length exceeds VFAT restrictions
EWRTPRT	trying to modify file on write-protected disk
driver error	

Example

```
int att;

/* Set file attributes to “system” and “read-only” */
att = pcfm_chmod(“a:\\myfile.bin”, FA_SYSTEM|FA_RDONLY);
if( att == EOF )
   fatal(“Chmod error, a:\\myfile.bin\n”);
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

pcfm_chmodfp

Changes file attributes (uses pointer).

```
int pcfm_chmodfp(MTFILE *fp, int function, int
                  attribute);
```

fp	file descriptor pointer to open file
function	`0` = return current, `1` = set new attribute
attribute	new file attribute value

The ***pcfm_chmodfp()*** function will either return, or change the attributes of the open file specified by *fp*. If *function* = `0`, then the current file attributes are returned. If *function* = `1`, then the file attributes are set to *attribute*. The `FA_DIR` attribute cannot be changed by this function. The new attributes will have no effect until the file is closed and reopened (e.g., if the file is currently open for write, and is made read-only by this function, writes to the file are still permitted until the file is closed and reopened).

`FA_NORMAL`	Normal file (no attributes)
`FA_RDONLY`	Read-only file
`FA_HIDDEN`	Hidden file (does not affect accessibility)
`FA_SYSTEM`	System file
`FA_ARCH`	Archive bit (file changed since bit cleared)
`FA_DIR`	File is a subdirectory

See also: ***pcfm_chmod***

Return Value

+n	current attribute of the file
`EOF`	failure

errno Value

Stream I/O

None

PC File Manager

EBADFP	bad file pointer
EACCESS	illegal attribute change
EBADARG	requested attribute is illegal, or function is not 0 or 1 (only if USS_SIO_PCHK is 1)

Example

```
MTFILE *fp;
fp = mt_fopen("A:\\MYFILE.BIN", "r+b");
att = pcfm_chmodfp(fp, FA_SYSTEM|FA_RDONLY);
if( att == EOF )
  fatal("Chmodfp error, a:\\myfile.bin\n");
```

pcfm_chtime

Changes file date and time (uses path).

```
int pcfm_chtime (const char *pathname, uint16 ftime,
                  uint16 fdate);
```

pathname	full pathname to the file
ftime	the new file time to set
fdate	the new file date to set

The ***pcfm_chtime()*** function changes the file's modification date and time fields in its directory entry to the new values given. The *`ftime`* and *`fdate`* values are in the binary encoded format that is stored in the directory (as returned by the driver ***timestamp*** function). You can use the macros ***mak_fdate*** and ***mak_ftime***, which are defined in **mtio.h**, to convert year, month, day into the *`fdate`* format, and hour, minute, second into the *`ftime`* format.

See also: ***pcfm_chtimefp***

Return Value

0	OK
`EOF`	file not found or not accessible

errno Value

Stream I/O

None

PC File Manager

ENMFILE	no entries available in open streams array
ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters

ENOPATH	part of directory path or device not found
ENOENT	file not found in directory
ENOTDIR	path contains a file name (instead of a directory)
EBIGPATH	path length exceeds VFAT restrictions
EWRTPRT	trying to change time on write-protected disk
driver error	

Example

```
pcfm_chtime(“a:\\myfile.bin”, mak_ftime(12,30,00),
                mak_fdate(1997,1,1));
```

NOTE: USFiles accepts either ‘\’ or ‘/’ characters as name separators interchangeably.

pcfm_chtimefp

Changes file date and time (uses pointer).

```
int pcfm_chtimefp(MTFILE *fp, uint16 ftime, uint16
                  fdate);
```

fp	file descriptor pointer to open file
ftime	the new file time to set
fdate	the new file date to set

The ***pcfm_chtimefp()*** function changes the file’s modification date and time fields in its directory entry to the new values given. The `ftime` and `fdate` values are in the binary encoded format that is stored in the directory (as returned by the driver ***timestamp*** function). You can use the macros ***mak_fdate*** and ***mak_ftime,*** which are defined in **mtio.h,** to convert year, month, day into the `fdate` format, and hour, minute, second into the `ftime` format.

See also: ***pcfm_chtime***

Return Value

0	OK
EBADFP	bad file pointer
ELOCKED	timeout waiting for access to file system

Example

```
MTFILE *fp;
fp = mt_fopen("A:\\MYFILE.BIN", "r+b");
pcfm_chtimefp(fp, mak_ftime(12,30,00),
        mak_fdate(1997,1,1));
```

pcfm_chvlabel

Changes an existing volume label.

```
int pcfm_chvlabel(const char *drivename,
                  char *oldlabel, const char *newlabel);
```

drivename	name of drive to alter label on (e.g. "**A:**")
oldlabel	pointer to where to return old label
newlabel	the new label string to set

The ***pcfm_chvlabel()*** function returns the existing volume label of the specified drive in *oldlabel*. If no volume label currently exists, *oldlabel* will be set to an empty string. If *newlabel* does not equal NULL, then the *newlabel* string is made the current volume label.

Return Value

0	OK
errno	*see* errno *values below*

errno Value

Stream I/O

None

PC File Manager

ENMFILE	no entries available in open streams array
ELOCKED	timeout while waiting for file system access
ENOBUF	no buffers available
ECTLFAIL	device controller failed
EWRGFMT	sector size not 512 bytes
ENOMEM	no memory for file structure allocation
EBADPART	sector does not contain partition table
EDSKCHG	disk has changed
EBADNAM	file name too long or has bad characters
ENOPATH	device not found
ERDFULL	root directory is full
EDSKFUL	disk is full
EWRTPRT	trying to modify write-protected disk
driver error	

Example

```
char olda[12], oldb[12];
if(pcfm_chvlabel(“A:”,olda,NULL))
              /* get drive a: label */
    error_stop(1);
if(pcfm_chvlabel(“b:”,oldb,”New Volume”))
```

```
                    /* set drive b: label */
        error_stop(2);
```

pcfm_invalidate_buffers

Invalidates all buffers on a device.

```
int pcfm_invalidate_buffers(DEVICE *devp);
```

devp	*pointer to device*

The ***pcfm_invalidate_buffers()*** function is provided for error recovery purposes. We can use this function to mark all buffers for the device as unused. If there are dirty buffers found for the device, then 1 is returned. These buffers are still invalidated. We simply report that some data will be lost. If no dirty buffers are found for the device, then 0 is returned.

See also: ***otherFilesOpen, invalidate_streams***

Return Value

0	no dirty buffers for device
1	dirty buffers found for device
`EOF`	*`devp`* is `NULL` (only if `USS_SIO_PCHK` is 1)

errno Value

Stream I/O

None

PC File Manager

EBADARG	*devp* is NULL (only if USS_SIO_PCHK is 1)

Example

```
DEVICE *devp;

/* Disk has changed */
pcfm_invalidate_buffers(devp);
if(otherFilesOpen(devp))
    invalidate_streams(devp);
else
    /* No open files, so ignore error */
```

putBigEnd16

Records 16-bit integer in Big-Endian mode.

```
void putBigEnd16(uint16 value, byte **pos);
```

value	*number to store*
pos	*address of pointer indicating where to store Big-Endian integer*

The routine ***putBigEnd16()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 16-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd16, getLitEnd32, putBigEnd32, putLitEnd16, putLitEnd32***

Return Value

None

Example

```
byte buffer[512], *bp;
uint16 number;

/* Point to beginning of 16-bit Big-Endian integer */
bp = &buffer[10];
number = 0xFACE;
```

```
putBigEnd16(number, &bp);
/* bp will now be at &buffer[12] */
```

putBigEnd32

Records 32-bit integer in Big-Endian mode.

```
void putBigEnd32(uint32 value, byte **pos);
```

value	*number to store*
pos	*address of pointer indicating where to store Big-Endian integer*

The routine ***putBigEnd32()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 32-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd16, getLitEnd32, putBigEnd16, putLitEnd16, putLitEnd32***

Return Value

None

Example

```
byte buffer[512], *bp;
uint32 number;

/* Point to beginning of 32-bit Big-Endian integer */
bp = &buffer[10];
number = 0x12FACE32;
putBigEnd32(number, &bp);
/* bp will now be at &buffer[14] */
```

putLitEnd16

Records 16-bit integer in Little-Endian mode.

```
void putLitEnd16(uint16 value, byte **pos);
```

value	*number to store*
pos	*address of pointer indicating where to store Little-Endian integer*

The routine ***putLitEnd16()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 16-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd16, getLitEnd32, putBigEnd16, putBigEnd32, putLitEnd32***

Return Value

None

Example

```
byte buffer[512], *bp;
uint16 number;

/* Point to start of 16-bit Little-Endian integer */
bp = &buffer[10];
number = 0xFACE;
putLitEnd16(number, &bp);
/* bp will now be at &buffer[12] */
```

putLitEnd32

Records 32-bit integer in Little-Endian mode.

```
void putLitEnd32(uint32 value, byte **pos);
```

value	*number to store*
pos	*address of pointer indicating where to store Little-Endian integer*

The routine ***putLitEnd32()*** is primarily an internal routine, but it may prove useful in some applications. The pointer will be incremented to the next byte following the 32-bit integer.

See also: ***getBigEnd16, getBigEnd32, getLitEnd16, getLitEnd32, putBigEnd16, putBigEnd32, putLitEnd16***

Return Value

None

Example

```
byte buffer[512], *bp;
uint32 number;

/* Point to start of 32-bit Little-Endian integer */
bp = &buffer[10];
number = 0x12FACE32;
putLitEnd32(number, &bp);
/* bp will now be at &buffer[14] */
```

total_byte_cnt

Returns the number of bytes on the drive.

```
uint32 total_byte_cnt(MTFILE *stream);
```

stream pointer to the *stream* file descriptor

The number of bytes on the disk drive associated with stream is returned. If an error occurs, 0 is returned.

NOTE: If using FAT32 support, the total byte count may exceed the limits of a 32-bit unsigned integer.

See also: ***total_clust_cnt, total_kb_cnt, free_byte_cnt, free_clust_cnt, free_kb_cnt***

Return Value

Number of bytes on disk.

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

None

Example

```
FILE *fp;
uint32 totalbytes;
   /* open for read/write */
   fp = mt_fopen(“A:\file1”, “r+b”);
   totalbytes = total_byte_cnt(fp);
```

total_clust_cnt

Returns the number of clusters on the drive.

```
uint32 total_clust_cnt(MTFILE *stream);
```

stream	pointer to the *stream* file descriptor

The number of clusters on the disk drive associated with stream is returned. If an error occurs, 0 is returned.

See also: ***total_byte_cnt, total_kb_cnt, free_byte_cnt, free_clust_cnt, free_kb_cnt***

Return Value

Number of clusters on disk.

errno Value

Stream I/O

EBADFP	bad file pointer

PC File Manager

None

Example

```
FILE *fp;
uint32 totalclusts;
   /* open for read/write */
   fp = mt_fopen("A:\file1", "r+b");
   totalclusts = total_clust_cnt(fp);
```

total_kb_cnt

Returns the number of kilobytes on the drive.

```
uint32 total_kb_cnt(MTFILE *stream);
```

stream	pointer to the *stream* file descriptor

The number of kilobytes on the disk drive associated with stream is returned. If an error occurs, 0 is returned.

See also: ***total_byte_cnt, total_clust_cnt, free_byte_cnt, free_clust_cnt, free_kb_cnt***

Return Value

Number of kilobytes on disk.

***errno* Value**

Stream I/O

EBADFP	bad file pointer

PC File Manager

None

Example

```
FILE *fp;
uint32 totalkb;
  /* open for read/write */
  fp = mt_fopen(“A:\file1”, “r+b”);
  totalkb = total_kb_cnt(fp);
```

uni2char

Converts a Unicode character to ASCII or Shift-JIS.

`void uni2char(char **asciiPos, uint16 uniChar);`

asciiPos	address of pointer where ASCII character will be stored
uniChar	Unicode character to convert

A Unicode character may be converted to either a single-byte (ASCII) or double-byte (Shift-JIS) character. The ***uni2char()*** function will move the *asciiPos* pointer to indicate where the next character should be recorded. It will be incremented by either one or two bytes.

This function is not included with the default settings of USFiles. No USFiles functions make use of this utility, but it is provided for application use. To include the ***uni2char()*** function, please see the comments in **usfutil.c**.

USFiles supports a limited set of Unicode characters. The file **uni2jis.c** provides details of which characters are allowed. If an unsupported

Unicode character is passed into ***uni2char()***, then we set `**asciiPos` equal to the ASCII replacement character (`0x1A`).

See also: ***char2uni***

Return Value

none

Example

```
uint16 uniString[20];
char saveString[20],*pAscii;
int i;

pAscii = saveString;
for(i=0; i<20; i++)
  if(uniString[i]){
        uni2char(&pAscii, uniString[i]);
        /* pAscii will point to next available
        ** position */
  }else{
        *pAscii = 0;      /* NULL Terminate */
        break;
  }
```

4. Supported RTOSes

Chapter Contents

Using Stream I/O from Multiple Tasks

USFiles will allow multiple tasks to use the file system simultaneously when used with an RTOS. There is no record locking on individual files, however, so any file opened for modification (write, rename, or delete) cannot be opened by another task. An attempt to do so will result in the second open returning a `NULL` file pointer and setting `errno` to `EISOPEN`. There will be no conflict between any accesses to separate files, or multiple read-only accesses to the same file if that file is not opened for modification by any task.

The ***LOCK_FILESYSTEM()*** macro used to acquire the `PCFM_RESOURCE` when using USFiles with an RTOS locks the file system for the duration of each read or write operation. This will insure the operation is complete before another task gets control of the file system. You should therefore be able to have two or more tasks appending records to the end of the same file, as long as they are using the same file handle (i.e., one task should open the file only, and the other task make use of the same `FILE *`).

After a write operation, the data may not be immediately transferred to disk, but may remain in an internal buffer until either the buffer is needed, or the stream is flushed with ***fflush()***, or closed by ***fclose()***.

Multitasking with *errno*

Customers must pay special attention to *errno*, especially with uITRON RTOSes, which generally do not implement protections for *errno*. Many libraries (floating point, TCP/IP, file system, etc.) could theoretically use *errno*. If you are using several libraries that utilize *errno*, then you must implement a system-wide task-safe *errno*.

For USFiles only, an example is provided below to implement a task-safe *errno* on RTOSes that do not already protect it. This example assumes a uITRON RTOS.

In **rtossup.c:**

```
ID my_task_id(void){
       ID myid;
       get_tid(&myid);
       return myid;
}
```

In **rtos.h:**

```
int errno_array[ NUMTASKS ];

#define errno   errno_array[ my_task_id() ]
```

This will set up an array that stores *errno* for each task.

Supported RTOSes

At present USFiles has been integrated with the following RTOSes:

- None (does not require an RTOS)
- MultiTask!
- TronTask! (both versions 2.x and 3.x)
- Hitachi SH-7, SH-77, and HI7750
- RX850
- RX850 Pro
- PPSM and PPSM GT

When integrating USFiles with an RTOS, these items must be considered:

- Protecting stream I/O
- Protecting the file system
- Dynamic memory allocation
- Defining `errno`

The following sections will describe how each of the supported RTOSes handles these issues as well as discussing other items of interest concerning the RTOS. The tables in each section map a USFiles call to an RTOS call (or to an intermediate function that uses an RTOS call) and list the files that are related to this integration. Please examine the files, because some of the calls may change. All USFiles RTOS-specific information has been placed in the files **rtos.h** and **rtossup.c**, which are found in the **siosrc\\<*rtos*>** subdirectory.

The file system has four cases where it requests memory allocation:

- A single 512-byte block is requested when reading the BPB sector. This is returned immediately after accessing the data in that sector.

- Each time ***mt_fopen*** is called a file handle structure (MTFILE) is allocated. This will be released when the file is closed.
- Before the file handle allocation is done, ***mt_fopen*** will request allocation of space to copy the filename argument passed to the open call. This is used to parse the device name from the device table, and will be released as soon as this process is finished.
- When using the ***mt_readdir*** function with long file names (VFAT), USFiles dynamically allocates space to temporarily hold the Unicode name before it is converted to ASCII. This requires 260 bytes, which are released before the function return.

Stand-alone Mode

Table 6-1: Stand-alone Mode Calls

USFiles Call	**RTOS Call**
Protecting Stream I/O	
N/A	
Protecting the File System	
N/A	
Dynamic Memory Allocation	
alloc_mem()	***calloc()***
dealloc_mem()	***free()***

errno

USFiles in stand-alone mode uses the compiler library’s **errno.h** file.

MultiTask!

Table 6-2: MultiTask! Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	*`mt busy++`*
UNLOCK_ STREAMIO()	***MTqproc()***
Protecting the File System	
LOCK_ FILESYSTEM()	***getres*** *`(PCFM_RESOURCE, PCFM TIMEOUT)`*
UNLOCK_ FILESYSTEM()	***relres*** *`(PCFM_RESOURCE)`*
Dynamic Memory Allocation	
alloc _mem()	***reqmem()***
dealloc _mem()	***relmem()***

errno

MultiTask! defines *`errno`* for each task. Please refer to the MultiTask! documentation and source code for details.

Stack Size

The file **depends.h** specifies a minimum stack size required for the MultiTask! test programs to run. USFiles requires a larger stack size, so in the test programs, we define macros `XTRA_STACK` and `XTRA_MEM` to account for this difference.

If the MultiTask! test programs and **usftest** in stand-alone mode all run successfully, but **usftest** fails when running with MultiTask!, then you may want to try increasing the stack size.

NOTE: When using a board support package (BSP), the default stack size will be specified in a BSP header file. Please see **siosrc\<*cpu*>\cpunotes.txt** to determine whether a BSP is being used.

Dynamic Task Loading with fruntsk

With USFiles for the 80x86 platform we have added the capability to dynamically load and start a MultiTask! task or several tasks from a separately compiled ***.exe** file. This requires a task that is already running to initiate the load and startup of the task in the ***.exe** file by making a call to ***fruntsk()***.

We will call the currently running portion of your application the "*static part*" and the part you will be loading from the file the "*overlay*" (although it is not overwriting any code).

The static part must be compiled as `huge` model, and must contain all of the MultiTask! operating system as well as USFiles or equivalent file access library and the new modules:

fruntsk.c
dyload.c
dytable.asm

The overlay is compiled as either `large` or `huge` model (Microsoft C), or `huge` model (Borland C) with the label `MT_OVERLAY` defined (usually by adding `-DMT_OVERLAY` to the *`CFLAGS`* variable in the makefile). The overlay module is then linked with **dyentry.asm** instead of the usual startup module (i.e., `dyentry` replaces the compiler C startup routine which calls `main`).

The model restrictions are necessary in order to relocate the code in the overlay file and be able to dynamically link it to the system services in the static part. `Dyentry` and `dytable` set up a jump table in the overlay with a jump to each MultiTask! system function in the static part. The commonly needed MultiTask! global variables are redefined for the overlay as functions returning pointers to the variable. This is all transparent to the overlay, and all coding in the overlay task is identical on the user level to coding for the static part.

If you need to add access for the overlay to an additional function or variable which resides in the static part, you can do so by adding a table entry to the file **dyconf.asm** and recompiling both the static and overlay parts.

Each item requiring a dynamic link appears in the **dyconf.asm** file as a line with the item name preceded by either the ***funclnk*** macro for functions or the ***datalnk*** macro for data items. Each data item also requires a `#define` and `extern` declaration in the C overlay file. An example of this is shown in the **dyconf.asm** file. The MultiTask! data items already defined in this file have their `#define` and `extern` declarations already in place in **mtdata.h** where they are conditionally included when `MT_OVERLAY` is defined.

The makefile contains a target program, **dytest.c**, which builds a static part that loads in the **coretest** program as an overlay and runs it. Refer to this as an example of using the dynamic load capability.

NOTE: Dynamic task loading is now considered unsupported, and the source files can be found in the **siosrc\unsupp** directory.

fruntsk

Loads and runs a task from a file.

```
int fruntsk(uint priority, char *fname, uint stksiz,
                ...);
```

priority	task priority to be assigned to the loaded task
**fname*	the complete pathname to code being loaded
stksiz	the size of stack to assign to the loaded task
...	up to 4 arguments to be passed to task

The ***fruntsk()*** function is similar to ***runtsk*** except the task is first loaded from a file whose pathname is given by *fname*. The file must be a relocatable type as used by **dyload.c**. (For 80x86 this will be a ***.exe** file.) Memory is allocated by a call to ***reqmem()*** to hold the code in the module loaded. The code is located and loaded into memory, and the stack space `stksiz` bytes is allocated with a call to ***reqmem().*** The ***main()*** function of the loaded file is started as the first (and possibly only) task in the file. If there was more than one task contained in the file, each of these needs to be started by the first task (***main()***) in that file. Be aware that all code memory is attached to the first task of that file and will be deallocated when that task dies. Because of this you must ensure that it will be the last task to die of the group loaded in that module.

NOTE: The ***fruntsk*** function is part of USFiles and is currently only supported on the 80x86 real mode platform.

Return Value

+n	`TASK_ID` (slot number) of loaded task
`E_IOERR`	Error reading file (more information may be contained in *`errno`*)
`E_NOSLOT`	All task slots in configuration (*`NUMTSK`*) are in use
`E_NORAM`	insufficient memory (***reqmem***)

Example

```
TASK_ID slot;

  slot = fruntsk(100, “a:\\bin\\ovltask.exe”, 1000);
  if( slot < 0 )
     { error }
```

Table 6-3: TronTask! Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***ussStreamForbid()***
UNLOCK_ STREAMIO()	***ussStreamPermit()***
Protecting the File System	
LOCK_ FILESYSTEM()	***ussFileForbid()***
UNLOCK_ FILESYSTEM()	***ussFilePermit()***
Dynamic Memory Allocation	
alloc _mem()	***alloc_mem()***
dealloc _mem()	***relmem()***

Since tasks cannot 'own' a resource governed by TronTask!'s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** function in **rtossup.c** calls the kernel ***reqmem()*** function.

errno

TronTask! defines `errno` for each task. Please refer to the TronTask! documentation and source code for details.

Initializing USFiles

To gain access to stream I/O and the file system, the semaphores protecting these must be created (for TronTask! 3.x only) and signaled. This should be done at the start of your application by calling the function ***ussSIOInit()***, which is found in the file **rtossup**.**c**. This routine will create the semaphores, if necessary, and signal them so that access can be granted to tasks. For an example of initializing USFiles with TronTask!, please see **usftest.c**.

Stack Size

The file **depends.h** specifies a minimum stack size required for the TronTask! test programs to run. USFiles requires a larger stack size, so in the test programs, we define macros `XTRA_STACK` and `XTRA_MEM` to account for this difference.

If the TronTask! test programs and **usftest** in stand-alone mode all run successfully, but **usftest** fails when running with TronTask!, then you might want to try increasing the stack size.

NOTE: When using a board support package (BSP), the default stack size will be specified in a BSP header file. Please see **siosrc\<cpu>\cpunotes.txt** to determine whether a BSP is being used.

Table 6-4: Hitachi ITRON Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***wai_sem***(`STREAM_RESOURCE`)
UNLOCK_STREAMIO()	***sig_sem***(`STREAM_RESOURCE`)
Protecting the File System	
LOCK_FILESYSTEM()	***usfitron_getres()***
UNLOCK_FILESYSTEM()	***usfitron_relres()***
Dynamic Memory Allocation	
alloc_mem()	***alloc_mem()***
dealloc_mem()	***dealloc_mem()***

Since tasks cannot ‘own’ a resource governed by Hitachi ITRON’s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_FILESYSTEM()*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** and ***dealloc_mem()*** calls are also in **rtossup.c**. We make use of the RTOS ***get_blk()*** and ***rel_blk()*** calls, but we also tried to do some optimization. Please see the **rtossup.c** file for details.

errno

We include the compiler library’s **errno.h** file in **rtos.h**.

Test Environment

Refer to the **config\sh\hitachi\compiler.mak** file for instructions on configuring USFiles for a particular SH processor. Depending on your RTOS, you might have to modify another makefile in **config\sh\hitachi** to specify the path to the RTOS directory. The appropriate makefile is:

SH7	**sh7dos.mak**
SH77	**sh77dos.mak**
HI7750	**hi7750.mak**

The test is downloaded to RAM in the board via the serial port, and then run. The test sends text out the serial port from the board to a terminal attached to this port. The debug monitor is assumed to have initialized the serial port for the test. All test output goes through the function ***putch()*** in the file **getput?.c**, which can be found in the **siosrc\sh** directory. To direct the text displayed by the program to another location, you can replace this function. Note that there are multiple versions of this file specified by replacing **?** with the `TARGET` number selected by setting **compiler.mak** appropriately.

Using Library Header Files

stdlib.h

Compiling the file system requires that three lines be added to the compiler header file **stdlib.h** in order to compensate for nested inclusion of this file.

Near the top of **stdlib.h** (before any statements that are not comments) add these two lines:

```
#ifndef  stdlib_h
#define  stdlib_h
```

After the last line of the file add:

```
#endif
```

These will prevent any problem if the file is included more than once, which is the case with the file system code. This is difficult to avoid because of the way the header files are used for other systems.

Check the contents of **stdarg.h** to see if similar lines already exist (the exact name defined is not important, but the structure is). This is common practice for most C compilers, and may already be present in later releases of the Hitachi compiler.

stdio.h

Using the Hitachi compiler library **stdio.h** with USFiles may produce a linker error, due to the name of the USFiles module **sprintf.c**. This problem can be remedied by following these steps:

1. Rename **sprintf.c** to **usprintf.c.**
2. Edit **siosrc\makefile** to change **sprintf** to **usprintf.**
3. Rebuild the library and application.

The USFiles modules and test programs do not use **stdio.h**, so this fix is only necessary if your application needs **stdio.h**.

The depends.h File

You should make changes only to the following statements, when appropriate.

```
  #define MASK_INTS()     set_imask(0xf)
```

This line defines a macro used to mask interrupts in the system. Normally this sets the interrupt mask value to the highest possible setting to disable all interrupts. This may be set to some other level for some circumstances if you understand the usage.

```
#define UNMASK_INTS()     set_imask(0)
```

This macro returns interrupt mask setting to zero. For the case of using USFiles without an RTOS, there is only one place where these macros are used, for a very brief time, in **streamio.c**.

Configuration Files

When using SH7 or SH77, customized configuration files are provided in the **siosrc\<*rtos*>** subdirectories. Other configuration files needed for these RTOSes are the default files. Their location is identified by the `MTPTH` symbol in **sh7dos.mak** or **sh77dos.mak** (see **siosrc\<*rtos*>\makefile**). The customized configuration files are:

suptbl7.c	Variation of **hisuptbl.c** for SH1 or SH2
suptbl77.c	Variation of **hisuptbl.c** for SH3
tstsup2.c	SH2 version of above
tstsup3.c	SH3 version of above

When using HI7750, we do not provide any customized configuration files. The user must configure the RTOS files according to the instructions found in **siosrc\sh\cpunotes.txt**.

Interface

The file **rtossup.c** provides the functions to interface the file system to Hitachi ITRON.

The only ITRON functions used by the file system are:

get_tid
get_blk
rel_blk
sig_sem
wai_sem

In addition to these, the test program **usftest.c** uses the following functions, and depends upon the timer interrupt handler calling `irot_rdq(2)`:

clr_flg
set_flg
wai_flg
sta_tsk
slp_tsk
wai_tsk
wup_tsk

The event flag used by SH7 and SH77 in **usftest** is defined in **usftest.c** as XEVT, which has a value of 10. HI7750 dynamically assigns the event flag ID number.

Various Makefiles

compiler.mak	Used to select the board
makefile.sh2	Specific information for DVE-7604 board
makefile.sh3	Specific information for DVE-7708 board
makefile.sh4	Specific information for Hitachi SH4 Solution Engine
sh7dos.mak	Specific rules for SH7 RTOS
sh77dos.mak	Specific rules for SH77 RTOS
hi7750.mak	Specific rules for HI7750 RTOS

Table 6-5: RX850 and RX850 Pro Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***wai_sem***(`STREAM_RESOURCE`)
UNLOCK_STREAMIO()	***sig_sem***(`STREAM_RESOURCE`)
Protecting the File System	
LOCK_FILESYSTEM()	***usfitron_getres()***
UNLOCK_FILESYSTEM()	***usfitron_relres()***
Dynamic Memory Allocation	
alloc_mem()	***alloc_mem()***
dealloc_mem()	***dealloc_mem()***

Since tasks cannot ‘own’ a resource governed by RX850’s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_FILESYSTEM()*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** and ***dealloc_mem()*** calls are also in **rtossup.c**. We make use of the RTOS ***get_blf()*** and ***rel_blf()*** (***get_blk()*** and ***rel_blk()*** for RX850 Pro) calls, but we also tried to do some optimization. Please see the **rtossup.c** file for details.

errno

We include the compiler library’s **errno.h** file in **rtos.h**

Test Environment

The provided makefiles are configured to compile USFiles with the Green Hills compiler located in the directory specified by `PTH` in **config\v8xx\green\compiler.mak**. In addition, the directory containing support for the RTOS is indicated by `NUCLEUS_TOP` in the same **compiler.mak**.

We have tested our software using the Kyoto Micro Computer Partner-ET Extended Target Debugger ROM emulator. You should be able to simply start the Partner software, load the compiled test, and run. Output is directed to the JRS232C port on the RTE-V850E/MS1-PC board. If you connect this to a monitor, the test output will be displayed.

If you are using a different board, then you will have to modify the ***putchr()*** routine in **getput1.850**, which is found in **siosrc\v8xx**

Board Revisions

We have tested our software with two different revisions of the RTE-V850E/MS1-PC board. The only difference that you need to be aware of is the clock speed. The file **siosrc\v8xx\serial.c** provides the serial driver for the evaluation boards. If you are using rev. 3.0 of the evaluation board, then be sure that `BRGC0` = 65 to indicate a 40MHz clock. If using rev. 3.1 of the evaluation board, then `BRGC0` should be 54, indicating a 33 MHz clock. One way to check for the proper clock speed is that output will appear garbled if the wrong clock speed is specified.

Configuration Files

The following files are used to configure **usftest** to run as a task with RX850 (Pro). These are modified versions of several of the original RX850 sample configuration files.

sit850.cf	RX850 configuration file in **siosrc\rx850** directory
sit850p.cf	RX850 Pro configuration file in **siosrc\rx850pro** directory

These files define all the RTOS items that are required (tasks, semaphores, memory pools, etc.). When the `STREAM_RESOURCE` and `PCFM_RESOURCE` semaphores are defined, they are initialized to 1, meaning that they are available to be accessed immediately. Please see the RX850 (Pro) documentation to see how to properly configure your particular application.

Interface

The file **rtossup.c** provides the functions to interface the file system to RX850 and RX850 Pro.

The only ITRON functions used by the file system are:

get_tid	
get_blk	(RX850 Pro)
rel_blk	(RX850 Pro)
get_blf	(RX850)
rel_blf	(RX850)
sig_sem	
wai_sem	

In addition to these, the test program **usftest.c** uses the following functions:

clr_flg
set_flg
wai_flg
sta_tsk
slp_tsk
dly_tsk
wup_tsk

The event flag used by **usftest** is *XEVT*, the value of which is determined by the configuration file.

For RX850, memory allocation is done with fixed-size blocks from pool `USFILES_MEMPOOL`. The block size is 608 bytes (defined in **sit850.cf**) and `USFILES_MEMPOOL` is defined in **siosrc\rx850\rtos.h**.

For RX850 Pro, memory allocation is done with variable-size blocks from pool `USFILES_MEMPOOL`, which is defined in **siosrc\rx850pro\rtos.h**.

5. Porting Guide

Chapter Contents

Porting USFiles Stand-alone Mode

There is little in USFiles that is processor dependent, so porting mostly involves identifying the correct libraries. USFiles makes use of several string functions like ***strcmp()*** and memory routines like ***memcpy()*** or ***memset()***. The easiest way to determine what library from your tool chain is required is to compile and link the **usftest** program and see which symbols remain unresolved.

We are assuming that you have been able to run a simple “Hello, World” program on your hardware before you start to port USFiles.

Setting Up Makefiles

To develop makefiles for a new processor and/or compiler, we recommend using existing makefiles as a starting point. If you are porting to a new CPU, you will have to add ***CPU*** and ***CPU\compiler*** subdirectories to the **config** and **siosrc** directories. Copy the existing **makefile** and **compiler.mak** files into these directories and edit them.

You will also have to add your new CPU and compiler names to the appropriate lists in **config.mak**. Make sure that the names in **config.mak** match the directory names that you have created.

NOTE: If a board support package (BSP) has been provided for your CPU, then porting is most easily done in the BSP model. Please see BSP documentation for porting instructions.

As an example, consider porting to a CPU called ***NewCpu***, and a compiler called ***NewComp***. These are the steps for porting:

1. Create these directories:

   siosrc\*NewCpu*
   siosrc\*NewCpu*\*NewComp*
   config\*NewCpu*
   config\*NewCpu*\*NewComp*

2. Copy the makefiles:
    a. Copy **siosrc\i8086\makefile** to **siosrc\NewCpu\makefile**
    b. Copy **siosrc\i8086\borland\makefile** to **siosrc\NewCpu\NewComp\makefile**
    c. Copy **config\i8086\borland\compiler.mak** to **siosrc\NewCpu\NewComp\compiler.mak**
3. Edit the makefiles:
    a. The **siosrc\*NewCpu*\makefile** should contain USFiles device drivers and may contain character I/O. Character I/O may be found in the **bspsrc** directory.
    b. The **siosrc\*NewCpu*\NewComp\makefile** will most likely not build anything.
    c. The **config\*NewCpu*\NewComp\compiler.mak** file has the necessary flags for tool chains, the paths to the tool chains, the rules for building, and the target-specific information
    d. In the **config.mak** file, add and select these lines:
    ```
    CPU = NewCpu
    COMPILER = NewComp
    ```

Of course, you will need to modify the code to support drivers and character output on a new board. Interrupts are only dealt with by the drivers themselves for USFiles. The RAM disk driver does not need any interrupts, but the i8086 hard disk and diskette drivers utilize interrupts. Please pay attention to interrupts when porting.

NOTE: When using a BSP, the BSP will handle the interrupt level.

When building a USFiles library or application in stand-alone mode, the files indicated in **siosrc\makefile**, **siosrc\NewCpu\makefile**, **siosrc\NewCpu\NewComp\makefile**, and **siosrc\none\makefile** will be compiled and added to the library according to the rules found in **config\NewCpu\NewComp\compiler.mak**. The library is then placed in the **lib** directory and the application (in **appsrc**) is compiled and linked with the library.

Editing Header Files

Copy the existing header files **depends.h** and **usstypes.h** to the **siosrc\NewCpu\NewComp** directory, and edit these files as needed.

NOTE: If a BSP is being used, then these files are unnecessary. The BSP header files replace these.

Porting Drivers

The drivers provided with USFiles, with the exception of the RAM disk driver, have been specifically developed for operation on PC hardware. Please regard these as samples only. Unless you are using PC hardware, they will require modification to work with your hardware. If you choose to use one of our drivers as a sample, we recommend avoiding the diskette driver (**flopdrv.c**). It has some peculiarities that will be discussed later. The BIOS driver (**biosdrv.c**) or the hard disk driver (**lbahddrv.c**) are clearer.

RAM Disk Driver

The RAM disk driver (**ramdrv.c**) should operate on any hardware with at least 256 KB of memory available. The size of the RAM disk can be configured in **ramdef.c**.

BIOS Driver

The BIOS driver (**biosdrv.c**) requires a BIOS to operate. It uses functions provided by the Microsoft and Borland tools to access the BIOS. This driver is representative of the structure required in a USFiles driver, but its utility is limited.

Hard Disk Driver

The hard disk driver (**lbahddrv.c**) provides direct access to an IDE hard drive. The drive can either operate in logical block addressing (LBA) mode, or in cylinder, head, sector (CHS) mode. Each drive's unit number in the device table determines how it is accessed. See Chapter 4, *Configuring USFiles*, for more details.

The driver initialization installs an interrupt service routine (ISR) in the expected DOS vector for IRQ 14. When operating in stand-alone mode, an ISR is also installed into the DOS timer interrupt vector to allow drive commands to timeout. These ISRs will require attention when porting to new hardware. In stand-alone mode, you may not wish to use interrupts. You could simply let the driver enter a spin loop until the drive operation completes.

You will also need to modify the hard drive communication port definitions near the top of the file. The file **diskio.asm** has the ***get_sector()*** and ***put_sector()*** routines called by the driver. There are samples of how these functions can be replicated in C code, but you may wish to provide assembly routines to improve speed.

Finally, you might wish to redefine the macro ***ptr_norm()***. This is used to normalize the 80x86 real mode buffer pointer. Other CPUs should not need this, so it can be redefined to do nothing.

Diskette Driver

The diskette driver does the same sort of initialization as the hard disk driver. It installs an ISR in the DOS interrupt vector for a floppy disk drive controller, and it will install the timer ISR in stand-alone mode, if it is not already installed.

The diskette driver has four ports defined, which will require modification to match your hardware. It also uses DMA. If your hardware does not have DMA, then this driver will require significant modification.

Memory Alignment

When developing or porting drivers, be aware of the memory alignment requirements of your CPU. This may become an issue when we directly transfer data from a disk to the user's buffer (bypassing internal USFiles buffers). Imagine the following situation:

1. We open a file and read 511 bytes from it.
2. We then call ***mt_fread()*** to read 2000 bytes into an application buffer.
3. The read routine realizes that we have to read one byte from an internal buffer. It does that and then increments the pointer to the application buffer by one.
4. The read routine then tries to directly transfer 1536 bytes from the disk to the application buffer. The application buffer is now positioned on an odd byte, so accessing the buffer this way may fail.

A solution to this problem is to have a 512-byte buffer in the driver that can be used to temporarily hold sectors. The read and write routines will have to test whether `bufp->userbuf` has the proper alignment. If it does not, transfer to the temporary buffer, and then to the application buffer (for read).

Another solution is to make sure that reads and writes are always done to maintain the proper alignment (e.g. a minimum of two bytes at a time).

Porting USFiles to a New RTOS

Chapter 6, *Supported RTOSes*, describes the issues that must be considered when integrating USFiles with an RTOS. These will be discussed in a bit more detail here. If you are porting USFiles to a processor and/or compiler for which we do not provide makefiles, you should first port USFiles to this environment in stand-alone mode, which is discussed in the previous section. This will insure that USFiles is operating properly for your development environment, and then the RTOS integration can be performed.

Integrating an RTOS with USFiles

When integrating USFiles with an RTOS, you must consider these items:

- Protecting stream I/O
- Protecting the file system
- Dynamic memory allocation
- Defining *`errno`*

Integrating Files

We have attempted to keep the files involved in RTOS integration to a minimum. For supported RTOSes, these files are **rtos.h** and **rtossup.c**. They are located in the appropriate **siosrc\<*rtos*>** directory. You will need to create a subdirectory in **siosrc** for your new RTOS and create **rtos.h** and **rtossup.c** files there. Make sure that you update the RTOS list in **config.mak** to include your new RTOS.

RTOS Header File

A good starting point for your RTOS header file is copying **rtos.h** from **siosrc\none** and editing that. This **rtos.h** file includes the definitions required by USFiles, and you can substitute the appropriate functions for your RTOS.

In this RTOS header file, you will need to define these macros:

LOCK_STREAMIO()
UNLOCK_STREAMIO()
LOCK_FILESYSTEM()
UNLOCK_FILESYSTEM()

These are typically implemented as resources (e.g. ***LOCK_STREAMIO()*** requests a resource and ***UNLOCK_STREAMIO()*** releases a resource). You will then have to define the ID numbers for the resources that protect the stream I/O and file system. This can be done directly in the header support file.

NOTE: Calls to ***LOCK_FILESYSTEM()*** may be nested. The implementation must allow a single task to call ***LOCK_FILESYSTEM()*** twice consecutively without an ***UNLOCK_FILESYSTEM()*** call in between. The counting semaphores of µITRON do not allow this, so we must wrap the semaphore calls with our own code. If your RTOS has similar behavior, please see the **rtos.h** and **rtossup.c** files in **siosrc\tt3** for an example of how to handle this.

Here you will also need to define how `errno` is implemented. For our MultiTask! and TronTask! RTOSes, each task can have an error code associated with it, so we use a macro to map `errno` to the task error code. With other supported RTOSes, we include the compiler library's **errno.h** file. Be sure that the `errno` is safe for a multitasking environment before using this.

RTOS Support File

The RTOS support file (**rtossup.c**) will contain the functions for dynamic memory allocation. Again, a good starting point is the **rtossup.c** file in the **siosrc\none** directory. Copy this to your new RTOS directory and edit as needed.

USFiles calls the functions ***alloc_mem()*** and ***dealloc_mem()*** to acquire and release heap memory. These functions should be defined in the RTOS support file. Since USFiles often allocates a `PCFM_BUFFER` structure, you may find it useful to set aside a block of memory with this size. When a function requests a block that is `sizeof(PCFM_BUFFER)`, you can return the address of this dedicated space. This is safe, because all the calls that would request this block are protected, so only one task at a time will access it. Here is an example that implements this technique:

```
/* defined as long to get address alignment */
uint32 usfblock[(sizeof(PCFM_BUFFER) + 3) / 4];

int dealloc_mem(void *reladr)
{                    /* release memory */

    if (reladr != usfblock) {
        /* Do RTOS memory free */
    } else {
        /* ignore release of PCFM_BUFFER block */
        return 0;
    }
}

void *alloc_mem(int reqsize)
{
    if (reqsize == sizeof(PCFM_BUFFER)) {
        /* return pointer to 512 byte block */
        return (usfblock);
    }
    else
         /* Do RTOS memory allocation */
     /* Return pointer to memory */
}
```

NOTE: The memory returned by a call to **alloc_mem()** must be initialized to zero for USFiles to function properly.

Building Your Application

When you build an application with USFiles and a new RTOS, you will obviously need to specify the RTOS library so that your application and USFiles can link with it. You can either do this in the **config\<*cpu*>\<*compiler*>\compiler.mak** file or by adding the RTOS library to the `USER_LIBS` list in **config.mak**.

Porting Drivers

If you have not already read the discussion on porting drivers for USFiles in stand-alone mode, you should do so now. Most of the issues discussed there are still applicable when using an RTOS. This section will only deal with issues specific to an RTOS.

Generally there are two items that must be handled when integrating a USFiles driver with an RTOS:

- Putting a task to sleep
- Waking up the task from an interrupt service routine

The **lbahddrv** driver is a good example to use. It typically:

- Saves the ID of the task that is running (so it can be woken up)
- Sends the proper command to the drive
- Waits with a timeout specified (here other tasks can operate)

When the interrupt is received from the drive, the ISR then wakes up the task so it can continue operation. This is usually all that must be handled when integrating our drivers with a new RTOS. Unfortunately, the diskette driver is an exception to this.

Diskette Driver

The complication with the diskette driver is that we need to keep track of the motor operation. We dedicate a task to turning the motor off when necessary. This is done in the ***pcfdrv_init()*** function. The device table entry for diskette drives has a field to specify the ID number of an event that indicates when the drive motor should be turned off. Be careful that you do not reuse this event ID number.

The ***motor_off_task()*** is started as a very high priority task. It only checks to see if the motor should be turned off. If so, it executes the ***MotorOff()*** function. When the ***motor_on()*** function is called we determine at what time the motor should be shut off by adding the present system time to the timeout period passed into ***motor_on()***. Signaling the ***motor_off_task()*** must be implemented so that if there is a second call to ***motor_on()*** before the current motor off time is reached, the old time is ignored, and the new time is recognized.

6. Supported RTOSes

Chapter Contents

Using Stream I/O from Multiple Tasks

USFiles will allow multiple tasks to use the file system simultaneously when used with an RTOS. There is no record locking on individual files, however, so any file opened for modification (write, rename, or delete) cannot be opened by another task. An attempt to do so will result in the second open returning a `NULL` file pointer and setting `errno` to `EISOPEN`. There will be no conflict between any accesses to separate files, or multiple read-only accesses to the same file if that file is not opened for modification by any task.

The ***LOCK_FILESYSTEM()*** macro used to acquire the `PCFM_RESOURCE` when using USFiles with an RTOS locks the file system for the duration of each read or write operation. This will insure the operation is complete before another task gets control of the file system. You should therefore be able to have two or more tasks appending records to the end of the same file, as long as they are using the same file handle (i.e., one task should open the file only, and the other task make use of the same `FILE *`).

After a write operation, the data may not be immediately transferred to disk, but may remain in an internal buffer until either the buffer is needed, or the stream is flushed with ***fflush()***, or closed by ***fclose()***.

Multitasking with *errno*

Customers must pay special attention to `errno`, especially with uITRON RTOSes, which generally do not implement protections for `errno`. Many libraries (floating point, TCP/IP, file system, etc.) could theoretically use `errno`. If you are using several libraries that utilize `errno`, then you must implement a system-wide task-safe `errno`.

For USFiles only, an example is provided below to implement a task-safe `errno` on RTOSes that do not already protect it. This example assumes a uITRON RTOS.

In **rtossup.c:**

```
ID my_task_id(void){
       ID myid;
       get_tid(&myid);
       return myid;
}
```

In **rtos.h:**

```
int errno_array[ NUMTASKS ];

#define errno   errno_array[ my_task_id() ]
```

This will set up an array that stores `errno` for each task.

Supported RTOSes

At present USFiles has been integrated with the following RTOSes:

- None (does not require an RTOS)
- MultiTask!
- TronTask! (both versions 2.x and 3.x)
- Hitachi SH-7, SH-77, and HI7750
- RX850
- RX850 Pro
- PPSM and PPSM GT

When integrating USFiles with an RTOS, these items must be considered:

- Protecting stream I/O
- Protecting the file system
- Dynamic memory allocation
- Defining `errno`

The following sections will describe how each of the supported RTOSes handles these issues as well as discussing other items of interest concerning the RTOS. The tables in each section map a USFiles call to an RTOS call (or to an intermediate function that uses an RTOS call) and list the files that are related to this integration. Please examine the files, because some of the calls may change. All USFiles RTOS-specific information has been placed in the files **rtos.h** and **rtossup.c**, which are found in the **siosrc\\<*rtos*>** subdirectory.

The file system has four cases where it requests memory allocation:

- A single 512-byte block is requested when reading the BPB sector. This is returned immediately after accessing the data in that sector.

- Each time ***mt_fopen*** is called a file handle structure (MTFILE) is allocated. This will be released when the file is closed.
- Before the file handle allocation is done, ***mt_fopen*** will request allocation of space to copy the filename argument passed to the open call. This is used to parse the device name from the device table, and will be released as soon as this process is finished.
- When using the ***mt_readdir*** function with long file names (VFAT), USFiles dynamically allocates space to temporarily hold the Unicode name before it is converted to ASCII. This requires 260 bytes, which are released before the function return.

Table 6-1: Stand-alone Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
N/A	
Protecting the File System	
N/A	
Dynamic Memory Allocation	
alloc_mem()	***calloc()***
dealloc_mem()	***free()***

errno

USFiles in stand-alone mode uses the compiler library’s **errno.h** file.

MultiTask!

Table 6-2: MultiTask! Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	`mt busy++`
UNLOCK_ STREAMIO()	***MTqproc()***
Protecting the File System	
LOCK_ FILESYSTEM()	***getres*** `(PCFM_RESOURCE, PCFM TIMEOUT)`
UNLOCK_ FILESYSTEM()	***relres*** `(PCFM_RESOURCE)`
Dynamic Memory Allocation	
alloc _mem()	***reqmem()***
dealloc _mem()	***relmem()***

errno

MultiTask! defines `errno` for each task. Please refer to the MultiTask! documentation and source code for details.

Stack Size

The file **depends.h** specifies a minimum stack size required for the MultiTask! test programs to run. USFiles requires a larger stack size, so in the test programs, we define macros `XTRA_STACK` and `XTRA_MEM` to account for this difference.

If the MultiTask! test programs and **usftest** in stand-alone mode all run successfully, but **usftest** fails when running with MultiTask!, then you may want to try increasing the stack size.

NOTE: When using a board support package (BSP), the default stack size will be specified in a BSP header file. Please see **siosrc\<*cpu*>\cpunotes.txt** to determine whether a BSP is being used.

Dynamic Task Loading with fruntsk

With USFiles for the 80x86 platform we have added the capability to dynamically load and start a MultiTask! task or several tasks from a separately compiled ***.exe** file. This requires a task that is already running to initiate the load and startup of the task in the ***.exe** file by making a call to ***fruntsk()***.

We will call the currently running portion of your application the "*static part*" and the part you will be loading from the file the "*overlay*" (although it is not overwriting any code).

The static part must be compiled as `huge` model, and must contain all of the MultiTask! operating system as well as USFiles or equivalent file access library and the new modules:

fruntsk.c
dyload.c
dytable.asm

The overlay is compiled as either `large` or `huge` model (Microsoft C), or `huge` model (Borland C) with the label `MT_OVERLAY` defined (usually by adding `-DMT_OVERLAY` to the *`CFLAGS`* variable in the makefile). The overlay module is then linked with **dyentry.asm** instead of the usual startup module (i.e., `dyentry` replaces the compiler C startup routine which calls `main`).

The model restrictions are necessary in order to relocate the code in the overlay file and be able to dynamically link it to the system services in the static part. `Dyentry` and `dytable` set up a jump table in the overlay with a jump to each MultiTask! system function in the static part. The commonly needed MultiTask! global variables are redefined for the overlay as functions returning pointers to the variable. This is all transparent to the overlay, and all coding in the overlay task is identical on the user level to coding for the static part.

If you need to add access for the overlay to an additional function or variable which resides in the static part, you can do so by adding a table entry to the file **dyconf.asm** and recompiling both the static and overlay parts.

Each item requiring a dynamic link appears in the **dyconf.asm** file as a line with the item name preceded by either the ***funclnk*** macro for functions or the ***datalnk*** macro for data items. Each data item also requires a `#define` and `extern` declaration in the C overlay file. An example of this is shown in the **dyconf.asm** file. The MultiTask! data items already defined in this file have their `#define` and `extern` declarations already in place in **mtdata.h** where they are conditionally included when `MT_OVERLAY` is defined.

The makefile contains a target program, **dytest.c**, which builds a static part that loads in the **coretest** program as an overlay and runs it. Refer to this as an example of using the dynamic load capability.

NOTE: Dynamic task loading is now considered unsupported, and the source files can be found in the **siosrc\unsupp** directory.

fruntsk

Loads and runs a task from a file.

```
int fruntsk(uint priority, char *fname, uint stksiz,
                ...);
```

priority	task priority to be assigned to the loaded task
**fname*	the complete pathname to code being loaded
stksiz	the size of stack to assign to the loaded task
...	up to 4 arguments to be passed to task

The ***fruntsk()*** function is similar to ***runtsk*** except the task is first loaded from a file whose pathname is given by *fname*. The file must be a relocatable type as used by **dyload.c**. (For 80x86 this will be a ***.exe** file.) Memory is allocated by a call to ***reqmem()*** to hold the code in the module loaded. The code is located and loaded into memory, and the stack space *`stksiz`* bytes is allocated with a call to ***reqmem().*** The ***main()*** function of the loaded file is started as the first (and possibly only) task in the file. If there was more than one task contained in the file, each of these needs to be started by the first task (***main()***) in that file. Be aware that all code memory is attached to the first task of that file and will be deallocated when that task dies. Because of this you must ensure that it will be the last task to die of the group loaded in that module.

NOTE: The ***fruntsk*** function is part of USFiles and is currently only supported on the 80x86 real mode platform.

Return Value

+n	`TASK_ID` (slot number) of loaded task
`E_IOERR`	Error reading file (more information may be contained in *`errno`*)
`E_NOSLOT`	All task slots in configuration (*`NUMTSK`*) are in use
`E_NORAM`	insufficient memory (***reqmem***)

Example

```
TASK_ID slot;

   slot = fruntsk(100, “a:\\bin\\ovltask.exe”, 1000);
   if( slot < 0 )
      { error }
```

Table 6-3: TronTask! Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***ussStreamForbid()***
UNLOCK_ STREAMIO()	***ussStreamPermit()***
Protecting the File System	
LOCK_ FILESYSTEM()	***ussFileForbid()***
UNLOCK_ FILESYSTEM()	***ussFilePermit()***
Dynamic Memory Allocation	
alloc _mem()	***alloc_mem()***
dealloc _mem()	***relmem()***

Since tasks cannot ‘own’ a resource governed by TronTask!’s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** function in **rtossup.c** calls the kernel ***reqmem()*** function.

errno

TronTask! defines `errno` for each task. Please refer to the TronTask! documentation and source code for details.

Initializing USFiles

To gain access to stream I/O and the file system, the semaphores protecting these must be created (for TronTask! 3.x only) and signaled. This should be done at the start of your application by calling the function ***ussSIOInit()***, which is found in the file **rtossup**.**c**. This routine will create the semaphores, if necessary, and signal them so that access can be granted to tasks. For an example of initializing USFiles with TronTask!, please see **usftest.c**.

Stack Size

The file **depends.h** specifies a minimum stack size required for the TronTask! test programs to run. USFiles requires a larger stack size, so in the test programs, we define macros `XTRA_STACK` and `XTRA_MEM` to account for this difference.

If the TronTask! test programs and **usftest** in stand-alone mode all run successfully, but **usftest** fails when running with TronTask!, then you might want to try increasing the stack size.

NOTE: When using a board support package (BSP), the default stack size will be specified in a BSP header file. Please see **siosrc\<cpu>\cpunotes.txt** to determine whether a BSP is being used.

Table 6-4: Hitachi ITRON Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***wai_sem***`(STREAM_RESOURCE)`
UNLOCK_STREAMIO()	***sig_sem***`(STREAM_RESOURCE)`
Protecting the File System	
LOCK_FILESYSTEM()	***usfitron_getres()***
UNLOCK_FILESYSTEM()	***usfitron_relres()***
Dynamic Memory Allocation	
alloc_mem()	***alloc_mem()***
dealloc_mem()	***dealloc_mem()***

Since tasks cannot ‘own’ a resource governed by Hitachi ITRON’s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_FILESYSTEM()*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** and ***dealloc_mem()*** calls are also in **rtossup.c**. We make use of the RTOS ***get_blk()*** and ***rel_blk()*** calls, but we also tried to do some optimization. Please see the **rtossup.c** file for details.

errno

We include the compiler library’s **errno.h** file in **rtos.h**.

Test Environment

Refer to the **config\sh\hitachi\compiler.mak** file for instructions on configuring USFiles for a particular SH processor. Depending on your RTOS, you might have to modify another makefile in **config\sh\hitachi** to specify the path to the RTOS directory. The appropriate makefile is:

SH7	**sh7dos.mak**
SH77	**sh77dos.mak**
HI7750	**hi7750.mak**

The test is downloaded to RAM in the board via the serial port, and then run. The test sends text out the serial port from the board to a terminal attached to this port. The debug monitor is assumed to have initialized the serial port for the test. All test output goes through the function ***putch()*** in the file **getput?.c**, which can be found in the **siosrc\sh** directory. To direct the text displayed by the program to another location, you can replace this function. Note that there are multiple versions of this file specified by replacing **?** with the `TARGET` number selected by setting **compiler.mak** appropriately.

Using Library Header Files

stdlib.h

Compiling the file system requires that three lines be added to the compiler header file **stdlib.h** in order to compensate for nested inclusion of this file.

Near the top of **stdlib.h** (before any statements that are not comments) add these two lines:

```
#ifndef  stdlib_h
#define  stdlib_h
```

After the last line of the file add:

```
#endif
```

These will prevent any problem if the file is included more than once, which is the case with the file system code. This is difficult to avoid because of the way the header files are used for other systems.

Check the contents of **stdarg.h** to see if similar lines already exist (the exact name defined is not important, but the structure is). This is common practice for most C compilers, and may already be present in later releases of the Hitachi compiler.

stdio.h

Using the Hitachi compiler library **stdio.h** with USFiles may produce a linker error, due to the name of the USFiles module **sprintf.c**. This problem can be remedied by following these steps:

4. Rename **sprintf.c** to **usprintf.c.**
5. Edit **siosrc\makefile** to change **sprintf** to **usprintf.**
6. Rebuild the library and application.

The USFiles modules and test programs do not use **stdio.h**, so this fix is only necessary if your application needs **stdio.h**.

The depends.h File

You should make changes only to the following statements, when appropriate.

```
  #define MASK_INTS()     set_imask(0xf)
```

This line defines a macro used to mask interrupts in the system. Normally this sets the interrupt mask value to the highest possible setting to disable all interrupts. This may be set to some other level for some circumstances if you understand the usage.

```
#define UNMASK_INTS()     set_imask(0)
```

This macro returns interrupt mask setting to zero. For the case of using USFiles without an RTOS, there is only one place where these macros are used, for a very brief time, in **streamio.c**.

Configuration Files

When using SH7 or SH77, customized configuration files are provided in the **siosrc\<*rtos*>** subdirectories. Other configuration files needed for these RTOSes are the default files. Their location is identified by the `MTPTH` symbol in **sh7dos.mak** or **sh77dos.mak** (see **siosrc\<*rtos*>\makefile**). The customized configuration files are:

suptbl7.c	Variation of **hisuptbl.c** for SH1 or SH2
suptbl77.c	Variation of **hisuptbl.c** for SH3
tstsup2.c	SH2 version of above
tstsup3.c	SH3 version of above

When using HI7750, we do not provide any customized configuration files. The user must configure the RTOS files according to the instructions found in **siosrc\sh\cpunotes.txt**.

Interface

The file **rtossup.c** provides the functions to interface the file system to Hitachi ITRON.

The only ITRON functions used by the file system are:

get_tid
get_blk
rel_blk
sig_sem
wai_sem

In addition to these, the test program **usftest.c** uses the following functions, and depends upon the timer interrupt handler calling `irot_rdq(2)`:

clr_flg
set_flg
wai_flg
sta_tsk
slp_tsk
wai_tsk
wup_tsk

The event flag used by SH7 and SH77 in **usftest** is defined in **usftest.c** as XEVT, which has a value of 10. HI7750 dynamically assigns the event flag ID number.

Various Makefiles

compiler.mak	Used to select the board
makefile.sh2	Specific information for DVE-7604 board
makefile.sh3	Specific information for DVE-7708 board
makefile.sh4	Specific information for Hitachi SH4 Solution Engine
sh7dos.mak	Specific rules for SH7 RTOS
sh77dos.mak	Specific rules for SH77 RTOS
hi7750.mak	Specific rules for HI7750 RTOS

Table 6-5: RX850 and RX850 Pro Mode Calls

USFiles Call	RTOS Call
Protecting Stream I/O	
LOCK_ STREAMIO()	***wai_sem***(`STREAM_RESOURCE`)
UNLOCK_STREAMIO()	***sig_sem***(`STREAM_RESOURCE`)
Protecting the File System	
LOCK_FILESYSTEM()	***usfitron_getres()***
UNLOCK_FILESYSTEM()	***usfitron_relres()***
Dynamic Memory Allocation	
alloc_mem()	***alloc_mem()***
dealloc_mem()	***dealloc_mem()***

Since tasks cannot ‘own’ a resource governed by RX850’s semaphores, we had to implement our own routines to interface with the resources to allow nested ***LOCK_FILESYSTEM()*** calls. These are found in the file **rtossup.c**.

The ***alloc_mem()*** and ***dealloc_mem()*** calls are also in **rtossup.c**. We make use of the RTOS ***get_blf()*** and ***rel_blf()*** (***get_blk()*** and ***rel_blk()*** for RX850 Pro) calls, but we also tried to do some optimization. Please see the **rtossup.c** file for details.

errno

We include the compiler library’s **errno.h** file in **rtos.h**

Test Environment

The provided makefiles are configured to compile USFiles with the Green Hills compiler located in the directory specified by `PTH` in **config\v8xx\green\compiler.mak**. In addition, the directory containing support for the RTOS is indicated by `NUCLEUS_TOP` in the same **compiler.mak**.

We have tested our software using the Kyoto Micro Computer Partner-ET Extended Target Debugger ROM emulator. You should be able to simply start the Partner software, load the compiled test, and run. Output is directed to the JRS232C port on the RTE-V850E/MS1-PC board. If you connect this to a monitor, the test output will be displayed.

If you are using a different board, then you will have to modify the ***putchr()*** routine in **getput1.850**, which is found in **siosrc\v8xx**

Board Revisions

We have tested our software with two different revisions of the RTE-V850E/MS1-PC board. The only difference that you need to be aware of is the clock speed. The file **siosrc\v8xx\serial.c** provides the serial driver for the evaluation boards. If you are using rev. 3.0 of the evaluation board, then be sure that `BRGC0` = 65 to indicate a 40MHz clock. If using rev. 3.1 of the evaluation board, then `BRGC0` should be 54, indicating a 33 MHz clock. One way to check for the proper clock speed is that output will appear garbled if the wrong clock speed is specified.

Configuration Files

The following files are used to configure **usftest** to run as a task with RX850 (Pro). These are modified versions of several of the original RX850 sample configuration files.

sit850.cf	RX850 configuration file in **siosrc\rx850** directory
sit850p.cf	RX850 Pro configuration file in **siosrc\rx850pro** directory

These files define all the RTOS items that are required (tasks, semaphores, memory pools, etc.). When the `STREAM_RESOURCE` and `PCFM_RESOURCE` semaphores are defined, they are initialized to 1, meaning that they are available to be accessed immediately. Please see the RX850 (Pro) documentation to see how to properly configure your particular application.

Interface

The file **rtossup.c** provides the functions to interface the file system to RX850 and RX850 Pro.

The only ITRON functions used by the file system are:

get_tid	
get_blk	(RX850 Pro)
rel_blk	(RX850 Pro)
get_blf	(RX850)
rel_blf	(RX850)
sig_sem	
wai_sem	

In addition to these, the test program **usftest.c** uses the following functions:

clr_flg
set_flg
wai_flg
sta_tsk
slp_tsk
dly_tsk
wup_tsk

The event flag used by **usftest** is *`XEVT`*, the value of which is determined by the configuration file.

For RX850, memory allocation is done with fixed-size blocks from pool `USFILES_MEMPOOL`. The block size is 608 bytes (defined in **sit850.cf**) and `USFILES_MEMPOOL` is defined in **siosrc\rx850\rtos.h**.

For RX850 Pro, memory allocation is done with variable-size blocks from pool `USFILES_MEMPOOL`, which is defined in **siosrc\rx850pro\rtos.h**.

7. Porting Guide

Chapter Contents

Porting USFiles Stand-alone Mode

There is little in USFiles that is processor dependent, so porting mostly involves identifying the correct libraries. USFiles makes use of several string functions like ***strcmp()*** and memory routines like ***memcpy()*** or ***memset()***. The easiest way to determine what library from your tool chain is required is to compile and link the **usftest** program and see which symbols remain unresolved.

We are assuming that you have been able to run a simple “Hello, World” program on your hardware before you start to port USFiles.

Setting Up Makefiles

To develop makefiles for a new processor and/or compiler, we recommend using existing makefiles as a starting point. If you are porting to a new CPU, you will have to add ***CPU*** and ***CPU\compiler*** subdirectories to the **config** and **siosrc** directories. Copy the existing **makefile** and **compiler.mak** files into these directories and edit them.

You will also have to add your new CPU and compiler names to the appropriate lists in **config.mak**. Make sure that the names in **config.mak** match the directory names that you have created.

NOTE: If a board support package (BSP) has been provided for your CPU, then porting is most easily done in the BSP model. Please see BSP documentation for porting instructions.

As an example, consider porting to a CPU called ***NewCpu***, and a compiler called ***NewComp***. These are the steps for porting:

4. Create these directories:

   siosrc\*NewCpu*
   siosrc\*NewCpu*\*NewComp*
   config\*NewCpu*
   config\*NewCpu*\*NewComp*

5. Copy the makefiles:
    d. Copy **siosrc\i8086\makefile** to **siosrc\NewCpu\makefile**
    e. Copy **siosrc\i8086\borland\makefile** to **siosrc\NewCpu\NewComp\makefile**
    f. Copy **config\i8086\borland\compiler.mak** to **siosrc\NewCpu\NewComp\compiler.mak**
6. Edit the makefiles:
    e. The **siosrc\*NewCpu*\makefile** should contain USFiles device drivers and may contain character I/O. Character I/O may be found in the **bspsrc** directory.
    f. The **siosrc\*NewCpu*\NewComp\makefile** will most likely not build anything.
    g. The **config\*NewCpu*\NewComp\compiler.mak** file has the necessary flags for tool chains, the paths to the tool chains, the rules for building, and the target-specific information
    h. In the **config.mak** file, add and select these lines:
    ```
    CPU = NewCpu
    COMPILER = NewComp
    ```

Of course, you will need to modify the code to support drivers and character output on a new board. Interrupts are only dealt with by the drivers themselves for USFiles. The RAM disk driver does not need any interrupts, but the i8086 hard disk and diskette drivers utilize interrupts. Please pay attention to interrupts when porting.

NOTE: When using a BSP, the BSP will handle the interrupt level.

When building a USFiles library or application in stand-alone mode, the files indicated in **siosrc\makefile**, **siosrc\NewCpu\makefile**, **siosrc\NewCpu\NewComp\makefile**, and **siosrc\none\makefile** will be compiled and added to the library according to the rules found in **config\NewCpu\NewComp\compiler.mak**. The library is then placed in the **lib** directory and the application (in **appsrc**) is compiled and linked with the library.

Editing Header Files

Copy the existing header files **depends.h** and **usstypes.h** to the **siosrc\NewCpu\NewComp** directory, and edit these files as needed.

NOTE: If a BSP is being used, then these files are unnecessary. The BSP header files replace these.

Porting Drivers

The drivers provided with USFiles, with the exception of the RAM disk driver, have been specifically developed for operation on PC hardware. Please regard these as samples only. Unless you are using PC hardware, they will require modification to work with your hardware. If you choose to use one of our drivers as a sample, we recommend avoiding the diskette driver (**flopdrv.c**). It has some peculiarities that will be discussed later. The BIOS driver (**biosdrv.c**) or the hard disk driver (**lbahddrv.c**) are clearer.

RAM Disk Driver

The RAM disk driver (**ramdrv.c**) should operate on any hardware with at least 256 KB of memory available. The size of the RAM disk can be configured in **ramdef.c**.

BIOS Driver

The BIOS driver (**biosdrv.c**) requires a BIOS to operate. It uses functions provided by the Microsoft and Borland tools to access the BIOS. This driver is representative of the structure required in a USFiles driver, but its utility is limited.

Hard Disk Driver

The hard disk driver (**lbahddrv.c**) provides direct access to an IDE hard drive. The drive can either operate in logical block addressing (LBA) mode, or in cylinder, head, sector (CHS) mode. Each drive's unit number in the device table determines how it is accessed. See Chapter 4, *Configuring USFiles*, for more details.

The driver initialization installs an interrupt service routine (ISR) in the expected DOS vector for IRQ 14. When operating in stand-alone mode, an ISR is also installed into the DOS timer interrupt vector to allow drive commands to timeout. These ISRs will require attention when porting to new hardware. In stand-alone mode, you may not wish to use interrupts. You could simply let the driver enter a spin loop until the drive operation completes.

You will also need to modify the hard drive communication port definitions near the top of the file. The file **diskio.asm** has the ***get_sector()*** and ***put_sector()*** routines called by the driver. There are samples of how these functions can be replicated in C code, but you may wish to provide assembly routines to improve speed.

Finally, you might wish to redefine the macro ***ptr_norm()***. This is used to normalize the 80x86 real mode buffer pointer. Other CPUs should not need this, so it can be redefined to do nothing.

Diskette Driver

The diskette driver does the same sort of initialization as the hard disk driver. It installs an ISR in the DOS interrupt vector for a floppy disk drive controller, and it will install the timer ISR in stand-alone mode, if it is not already installed.

The diskette driver has four ports defined, which will require modification to match your hardware. It also uses DMA. If your hardware does not have DMA, then this driver will require significant modification.

Memory Alignment

When developing or porting drivers, be aware of the memory alignment requirements of your CPU. This may become an issue when we directly transfer data from a disk to the user's buffer (bypassing internal USFiles buffers). Imagine the following situation:

5. We open a file and read 511 bytes from it.
6. We then call ***mt_fread()*** to read 2000 bytes into an application buffer.
7. The read routine realizes that we have to read one byte from an internal buffer. It does that and then increments the pointer to the application buffer by one.
8. The read routine then tries to directly transfer 1536 bytes from the disk to the application buffer. The application buffer is now positioned on an odd byte, so accessing the buffer this way may fail.

A solution to this problem is to have a 512-byte buffer in the driver that can be used to temporarily hold sectors. The read and write routines will have to test whether `bufp->userbuf` has the proper alignment. If it does not, transfer to the temporary buffer, and then to the application buffer (for read).

Another solution is to make sure that reads and writes are always done to maintain the proper alignment (e.g. a minimum of two bytes at a time).

Porting USFiles to a New RTOS

Chapter 6, *Supported RTOSes*, describes the issues that must be considered when integrating USFiles with an RTOS. These will be discussed in a bit more detail here. If you are porting USFiles to a processor and/or compiler for which we do not provide makefiles, you should first port USFiles to this environment in stand-alone mode, which is discussed in the previous section. This will insure that USFiles is operating properly for your development environment, and then the RTOS integration can be performed.

Integrating an RTOS with USFiles

When integrating USFiles with an RTOS, you must consider these items:

- Protecting stream I/O
- Protecting the file system
- Dynamic memory allocation
- Defining *`errno`*

Integrating Files

We have attempted to keep the files involved in RTOS integration to a minimum. For supported RTOSes, these files are **rtos.h** and **rtossup.c**. They are located in the appropriate **siosrc\\<*rtos*>** directory. You will need to create a subdirectory in **siosrc** for your new RTOS and create **rtos.h** and **rtossup.c** files there. Make sure that you update the RTOS list in **config.mak** to include your new RTOS.

RTOS Header File

A good starting point for your RTOS header file is copying **rtos.h** from **siosrc\none** and editing that. This **rtos.h** file includes the definitions required by USFiles, and you can substitute the appropriate functions for your RTOS.

In this RTOS header file, you will need to define these macros:

LOCK_STREAMIO()
UNLOCK_STREAMIO()
LOCK_FILESYSTEM()
UNLOCK_FILESYSTEM()

These are typically implemented as resources (e.g. ***LOCK_STREAMIO()*** requests a resource and ***UNLOCK_STREAMIO()*** releases a resource). You will then have to define the ID numbers for the resources that protect the stream I/O and file system. This can be done directly in the header support file.

NOTE: Calls to ***LOCK_FILESYSTEM()*** may be nested. The implementation must allow a single task to call ***LOCK_FILESYSTEM()*** twice consecutively without an ***UNLOCK_FILESYSTEM()*** call in between. The counting semaphores of µITRON do not allow this, so we must wrap the semaphore calls with our own code. If your RTOS has similar behavior, please see the **rtos.h** and **rtossup.c** files in **siosrc\tt3** for an example of how to handle this.

Here you will also need to define how *`errno`* is implemented. For our MultiTask! and TronTask! RTOSes, each task can have an error code associated with it, so we use a macro to map *`errno`* to the task error code. With other supported RTOSes, we include the compiler library's **errno.h** file. Be sure that the *`errno`* is safe for a multitasking environment before using this.

RTOS Support File

The RTOS support file (**rtossup.c**) will contain the functions for dynamic memory allocation. Again, a good starting point is the **rtossup.c** file in the **siosrc\none** directory. Copy this to your new RTOS directory and edit as needed.

USFiles calls the functions ***alloc_mem()*** and ***dealloc_mem()*** to acquire and release heap memory. These functions should be defined in the RTOS support file. Since USFiles often allocates a `PCFM_BUFFER` structure, you may find it useful to set aside a block of memory with this size. When a function requests a block that is `sizeof(PCFM_BUFFER)`, you can return the address of this dedicated space. This is safe, because all the calls that would request this block are protected, so only one task at a time will access it. Here is an example that implements this technique:

```
/* defined as long to get address alignment */
uint32 usfblock[(sizeof(PCFM_BUFFER) + 3) / 4];

int dealloc_mem(void *reladr)
{                   /* release memory */

    if (reladr != usfblock) {
        /* Do RTOS memory free */
    } else {
        /* ignore release of PCFM_BUFFER block */
        return 0;
    }
}

void *alloc_mem(int reqsize)
{
    if (reqsize == sizeof(PCFM_BUFFER)) {
        /* return pointer to 512 byte block */
        return (usfblock);
    }
    else
         /* Do RTOS memory allocation */
     /* Return pointer to memory */
}
```

NOTE: The memory returned by a call to **alloc_mem()** must be initialized to zero for USFiles to function properly.

Building Your Application

When you build an application with USFiles and a new RTOS, you will obviously need to specify the RTOS library so that your application and USFiles can link with it. You can either do this in the **config\<*cpu*>\<*compiler*>\compiler.mak** file or by adding the RTOS library to the `USER_LIBS` list in **config.mak**.

Porting Drivers

If you have not already read the discussion on porting drivers for USFiles in stand-alone mode, you should do so now. Most of the issues discussed there are still applicable when using an RTOS. This section will only deal with issues specific to an RTOS.

Generally there are two items that must be handled when integrating a USFiles driver with an RTOS:

- Putting a task to sleep
- Waking up the task from an interrupt service routine

The **lbahddrv** driver is a good example to use. It typically:

- Saves the ID of the task that is running (so it can be woken up)
- Sends the proper command to the drive
- Waits with a timeout specified (here other tasks can operate)

When the interrupt is received from the drive, the ISR then wakes up the task so it can continue operation. This is usually all that must be handled when integrating our drivers with a new RTOS. Unfortunately, the diskette driver is an exception to this.

Diskette Driver

The complication with the diskette driver is that we need to keep track of the motor operation. We dedicate a task to turning the motor off when necessary. This is done in the ***pcfdrv_init()*** function. The device table entry for diskette drives has a field to specify the ID number of an event that indicates when the drive motor should be turned off. Be careful that you do not reuse this event ID number.

The ***motor_off_task()*** is started as a very high priority task. It only checks to see if the motor should be turned off. If so, it executes the ***MotorOff()*** function. When the ***motor_on()*** function is called we determine at what time the motor should be shut off by adding the present system time to the timeout period passed into ***motor_on()***. Signaling the ***motor_off_task()*** must be implemented so that if there is a second call to ***motor_on()*** before the current motor off time is reached, the old time is ignored, and the new time is recognized.

A. Handling Disk Changes

Overview

Handling a disk change is the responsibility of the application designer. We are not able to anticipate all the situations under which an application can experience a disk change. Therefore, USFiles does not automatically provide for all possible methods of handing a disk change, which is largely the work of the driver. In the case of the PC drivers that we provide (**flopdrv.c** and **biosdrv.c**), the ***error_handler()*** routine deals with the disk change recovery.

The diskette drive disk change is checked in three places: During a raw read, during a raw write, and during a file open. The sector read and write routines call the raw read and write routines, so these will also pass through the disk change test. The sector reads and writes will then call the error handler if an error is encountered. The raw reads and writes do not, which is why USFiles almost exclusively uses the logical sector reads and writes from **pcfm**. When ***pcfm_open()*** is called, it also checks for a disk change and will branch to the error handler if one is sensed. All disk change errors should pass through the error handler.

Continuing with the New Disk

One method of handling a disk change error is to simply continue with the new disk that was inserted. This is the default method used in the USFiles drivers. The precise method that we have implemented in these drivers will not corrupt any diskettes. The worst that might happen is that data meant to be stored on the original disk is lost.

In ***pcfdrv_error_handler()***, case 0x06 is the disk change condition. The code there is:

```
*status = EDSKCHG;        /* our error code */
/* read sector 0 */
tmpstatus = pcfm_get_bpb(bufp->devp);
if(tmpstatus){
    bufp->error_status = tmpstatus;
    break;                      /* no retrys */
}
/*
** some recovery action is possible here
*/

/*
** Invalidate buffers regardless of whether files are
** open.  This is necessary to clear out FAT buffers.
*/
pcfm_invalidate_buffers(bufp->devp);
/*
** If there are any open files for the device,
** invalidate the streams for this device with
** no attempt at error recovery.
*/
if(otherFilesOpen(bufp->devp))
    invalidate_streams(bufp->devp);
else
    *status = 0;     /* No open files, ignore error */
retstatus = 0;      /* Default is do not retry */
break;
```

The first thing we try to do is read the BPB of the new disk. If that produces an error, then we simply give up and report an error. Once the BPB is successfully read, then we can test to see if the disk truly changed by comparing the old disk serial number found in *`bufp->serial_no`* to the new one found in *`bufp->devp->devparm.pcd->serial_no`*. We do not do this by default. If the serial numbers match, then we do not really have an error.

If the disk has truly changed (or if we do not bother checking), then we need to invalidate all buffers for the device, which will clear out FAT buffers as well. Then we check to see if files are open on the device. If there are, then we must also invalidate those streams.

The ***otherFilesOpen()*** routine only returns 1 if files <u>other</u> than the one being opened are found for the device. This allows handling the following situation seamlessly:

- Open file
- Close file
- Change disk
- Open new file

If we only checked for <u>any</u> open files, then we would receive a disk change error on the second file open.

Putting Back the Old Disk

The other common method for handling a disk change is to expect the original disk to be inserted again. One complicating factor with this method is that putting the original disk back in will signal another disk change. This will have to be accounted for in your error handling. One method is to call the driver ***diskchange()*** function to make sure that a disk has changed, and then make sure that it's the original (e.g. by checking serial numbers). You do not want to call a ***diskchange()*** function that will call the ***error_handler()*** again. This will result in nested ***error_handler()*** calls that may produce odd results.

A possible way to achieve this recovery method with **flopdrv** is:

```
*status = EDSKCHG;      /* our error code */
/* read sector 0 */
tmpstatus = pcfm_get_bpb(bufp->devp);
if(tmpstatus){
    bufp->error_status = tmpstatus;
    break;              /* no retrys */
}
/*
** some recovery action is possible here
*/
iprintf("Please insert original disk and press any key to
          continue\n");
getchr();
/* If disk has changed again */
if( internal_pcfdrv_diskchange() ){
   tmpstatus = pcfm_get_bpb(bufp->devp);
   if(tmpstatus){
       bufp->error_status = tmpstatus;
       break;                  /* no retrys */
   }
   if( bufp->serial_no ==
          bufp->devp->devparm.pcd->serial_no ){
      *status = 0;      /* No error necessary */
      retstatus = 0;    /* Do not retry */
      break;            /* Get out */
   }
}
/*
** If we get here, either the disk was not changed again,
** or the disk that was put in was not the original.
** Invalidate buffers regardless of whether files are
** open.  This is necessary to clear out FAT buffers.
*/
pcfm_invalidate_buffers(bufp->devp);
/*
** If there are any open files for the device,
** invalidate the streams for this device with
** no attempt at error recovery.
*/
if(otherFilesOpen(bufp->devp))
    invalidate_streams(bufp->devp);
else
    *status = 0;        /* No open files, ignore error */
retstatus = 0;          /* Default is do not retry */
break;
```

Other Situations

We want to emphasize that there are likely other methods to handle a changed disk. The two methods that we outlined above should prove useful in devising your own recovery scheme. If you need assistance with this matter, please contact U S Software

B. 386 Protected Mode

Supported Compilers

The USFiles 386 protected mode port may be built with the Microsoft, Borland, and CAD-UL tools. When using Borland or Microsoft be sure that the file **siosrc\i386\runtime.c** is added to the library. See **siosrc\i386\makefile**. For CAD-UL, this file should NOT be placed in the library.

Memory Allocation

USFiles operates in 386 Protected Mode with MultiTask! and TronTask! for the Microsoft, Borland, Watcom, MetaWare, and CAD-UL tools. Due to deficiencies in the libraries provided by Microsoft and Borland, these tool chains do not support USFiles in stand-alone mode.

The MetaWare and Watcom tool chains require some additions to allow USFiles to operate in stand-alone mode. Typically USFiles stand-alone mode maps ***alloc_mem()*** to the library ***calloc()*** routine and ***dealloc_mem()*** to ***free()***, but the ***calloc()*** provided by these libraries is not embeddable.

To enable support of 386 Protected Mode, USFiles uses the MultiTask! memory allocation routines ***reqmem()*** and ***relmem()***. This requires some initialization before it can be used. The test program **usftest.c** has this included, and the steps are:

1. Define a global array that will be used for memory allocation:
   `uint32 free_memory[ (MEMORY_SIZE +7)/4 ];`
2. Initialize the memory array in the ***main()*** function:
   `MTmeminit(&free_memory[1], MEMORY_SIZE);`

In this example, `MEMORY_SIZE` is the amount of memory that can be dynamically allocated.

NOTE: The CAD-UL support also uses the MultiTask! ***reqmem()*** and ***relmem()*** functions for consistency.

MTmeminit

Adds memory blocks to dynamic memory pool.

```
int MTmeminit(void *memory_ptr, mem_size_t size);
```

memory_ptr	address of memory to add to pool
size	number of bytes to add

The system will add *size* contiguous bytes of memory starting at *memory_ptr* to the pool of free memory managed by ***reqmem()*** and ***relmem(). MTmeminit()*** can accept multiple blocks of memory, but they must be provided in either ascending or descending order. On certain tool chains, you may be able to have the linker and/or startup code pass the actual available memory into your code. Otherwise, use an array of type `long`.

Return Value

SUCCESS	memory added to dynamic pool
E_RELMEM	corrupt memory block header (should never happen)

Example

```
long free_mem[2048]; /* allocate aligned memory */

MTmeminit(free_mem, sizeof(free_mem));

/* absolute memory designated */
MTmeminit((MTmem_t)0xc000, 0x4000);
```

Libraries

The Watcom compiler requires use of the library **$(PTH)\lib386\dos\clib3s.lib**, and MetaWare uses **$(PTH)\flat\hc386.lib**. Here `PTH` is the path to the compiler directory and is specified in the **makefile**.

C. VFAT

Overview

Long file names can be supported with an extension to the standard FAT file system called VFAT. This appendix explains VFAT and how it is implemented by USFiles.

How VFAT Works

VFAT was introduced to allow for longer file names recorded in Unicode. It makes use of the 32-byte directory entry structures, but several entries are strung together to make a VFAT directory entry. The short file name entry also has some fields that are not used by standard FAT12/16 entries. The extended directory entry is shown in Table C-1, with the new fields shown in boldface.

Table C-1: VFAT Short File Name Directory Entry

Relative Byte Position (hex [decimal])	**Field Description**	**Comments**
`00-07 [0-7]`	File name	Base of short file name
`08-0A [8-10]`	File extension	Extension of short file name
`0B [11]`	File attribute	See Table 2-5
`0C [12]`	Reserved	
`0D [13]`	**Creation time in 4-millisecond units**	**VFAT only**
`0E-11 [14-17]`	**Time and date created**	**VFAT only**
`12-13 [18-19]`	**Date of last access**	**VFAT only**
`14-15 [20-21]`	Reserved	
`16-19 [22-25]`	Time and date created	
`1A-1B [26-27]`	First cluster for file	
`1C-1F [28-31]`	File size	

A directory entry is marked as part of the long file name by setting the attribute byte to `0Fh` (i.e., read only, hidden, system, and label all set). The directory entries containing the long file name are stored in reverse order; the end of the file name will be encountered first, the beginning is near the end, and the DOS 8.3 version of the file name is last. Table C-2 shows how a directory entry is used as part of a long file name.

Table C-2: VFAT Long File Name Directory Entry

Relative Byte Position (hex [decimal])	**Field Description**	**Comments**
`0 [00]`	ID	Entry number for given file. ID > `40h` indicates end of long file name
`01-0A [1-10]`	First 5 characters of name	Unicode uses two bytes per character.
`0B [11]`	File attribute	`0Fh`
`0C [12]`	Reserved	`00h`
`0D [13]`	Alias Checksum	Checksum of DOS 8.3 name
`0E-19 [14-25]`	Next 6 characters of name	
`1A-1B [26-27]`	Reserved	00h
`1C-1F [28-31]`	Last 2 characters of name	

It is probably easiest to see how VFAT works by studying an example. Let us consider a file named "**This is a really long file name.temporary**", and see how it would be handled by USFiles.

First, the DOS 8.3 version of the file name is created. Any characters that are allowed by VFAT but not by DOS are replaced by an underscore ('_'). The exception is the space. If a space is encountered, it is just ignored. If illegal characters are found, or if the name is longer than DOS allows, then the file name will have a **~#** attached to it. The **#** will be a number to make the 8.3 file name unique. Our sample long file name will be converted to the DOS 8.3 file name "**THISIS~1.TEM**". DOS file names only use capital letters.

USFiles converts the ASCII long file name characters to Unicode. Every character is converted, including the '**.**' separating the file name from the extension. Each directory entry can hold 13 characters of the long file name, and we calculate how many slots are needed to hold the long file name. Our example has 41 characters, so it requires four slots. We need to find five consecutive available slots in the directory where the file will be stored (four for the long file name plus one for the DOS 8.3 file name). Once we have done this, we can store the directory entry. Table C-3 shows a representation of how these entries would appear in the directory. Be aware that for the long file name entries, the characters indicated will be stored in Unicode, not in ASCII.

Table C-3: Storage of Long File Names

Directory Entry Number	**Slot ID**	**Characters Stored**	**Comments**
`N`	`44h`	"ry"	Slot 4, but end of file name, so ID = `44h`
`N=1`	`03h`	'name "tempora"	
`N=2`	`02h`	"lly long file"	
`N=3`	`01h`	"this is a rea"	
`N=4`	N/A	"THISIS~1" "TEM"	DOS 8.3 does not use slot ID and does not store '.'

Since files with long names require several directory entries, one must be careful with the root directory. It has a limited number of entries available, and the number of files (if long file names are used) will be less than this.

Restrictions on VFAT

Allowed VFAT Characters

VFAT allows these characters in addition to the DOS characters:

- a to z (lower case)
- [ ] ; , = + <Space>

VFAT records long file names in Unicode, and it does make a distinction between lower and upper case letters. In addition to the long file name, VFAT creates a DOS 8.3 name. For the DOS 8.3 name any lower case letters are converted to upper case, any spaces are removed, and any of the characters allowed by VFAT but not by DOS are replaced by an underscore (‘_’).

VFAT file names may also contain Shift-JIS characters, but USFiles may only support converting a limited set of these into Unicode. The file **jis2uni.c** can be examined to determine which specific characters are supported.

File Name Lengths

The longest file name that VFAT allows contains 256 characters. This includes the ‘.’, the extension, and the `NULL` character at the end.

Path Lengths

When VFAT is used, a limit of 260 characters is imposed on the total path length. This includes the `NULL` character. Therefore, if you use the maximum 256 characters for a filename, that leaves you with three characters to specify a directory name, and one for the separator character (‘\’).

Number of Directory Entries

USFiles places a limit on the number of directory entries that a directory can hold when VFAT is being used. To keep track of available slots, we use a 16-bit unsigned integer to count slots, and therefore we are limited to $2^{16} = 65536$ slots for any directory except the root directory. The number of entries available to a FAT 12/16 root directory is determined when the disk is formatted.

Recall that long file name entries generally occupy several directory entries. If all file names occupy two slots (one for the DOS 8.3 name and one for the long file name), then that leaves us with 32768 files. The first two slots are reserved as aliases for the current directory and its parent, so that limits us to 32766 files in any directory (other than the root directory). Please be aware of this limitation.

Using VFAT

To enable VFAT, the symbol VFAT in **config\sio.mak** must be set to 1. The following tables show the character set configuration options for USFiles.

Access	means the file is created with USFiles, and read by USFiles.
Import	means the file is created by Windows95 or WindowsNT (or DOS for short file names), and may be properly found and opened by USFiles.
Export	means the file is created by USFiles, and read by either Windows95, WindowsNT or DOS.

In all cases where long file name access is not permitted, USFiles can still access the 8.3 name that corresponds to the long name, and access the file this way. The symbols `VFAT` and `FAKEUNICODE` are both found in **config\sio.mak**.

When `FAKEUNICODE = 0`, code size requirement is increased by approximately 14K for the necessary Unicode translation tables.

Table C-4: Access Configuration Options

Configuration Option	**`VFAT = 0`**	**`VFAT = 1 FAKEUNICODE = 0`**	**`VFAT = 1 FAKEUNICODE = 1`**
Access ASCII 8.3 (short name) files	Yes	Yes	Yes
Access Kanji (Shift-JIS) 8.3 short name files	Yes	Yes	Yes
Access ASCII long name files	No	Yes	Yes
Access Kanji (Shift-JIS) long name files	No	Yes	Yes

Table C-5: Import Configuration Options

Configuration Option	**`VFAT = 0`**	**`VFAT = 1 FAKEUNICODE = 0`**	**`VFAT = 1 FAKEUNICODE = 1`**
Import ASCII 8.3 (short name) files	Yes	Yes	Yes
Import Kanji (Shift-JIS) 8.3 short name files	Yes	Yes	Yes
Import ASCII long name files	No	Yes	Yes
Import Kanji (Shift-JIS) long name files	No	Yes	No

Table C-6: Export Configuration Options

Configuration Option	**`VFAT = 0`**	**`VFAT = 1 FAKEUNICODE = 0`**	**`VFAT = 1 FAKEUNICODE = 1`**
Export ASCII 8.3 (short name) files	Yes	Yes	Yes
Export Kanji (Shift-JIS) 8.3 short name files	Yes	Yes	Yes
Export ASCII long name files	No	Yes	Yes
Export Kanji (Shift-JIS) long name files	No	Yes	No

The Unicode translation table is actually mostly initialized data, but is counted as code space since it is ROMable. The figures below do not include the driver layer. The RAM disk driver consumes approximately 1.2K of code space. Data size includes 10 file buffers (number is configurable).

Table C-7: Approximate Code Sizes on 80x86 (Real-mode) Compiled with Borland C, Large Model

NOTE: All numbers with Borland C/C++ 5.0, `NUMBUFFERS` = 10.

Configuration	**Code Size (in Kbytes)**	**Data Size (BSS, in Kbytes)**
`VFAT = 0`	31	6
`VFAT = 1 FAKEUNICODE = 1`	36	7
`VFAT = 1 FAKEUNICODE = 0`	53	7

Case Sensitivity

When only using DOS 8.3 file names, USFiles automatically converts any lower case characters to upper case. With long file names enabled, this is not necessarily done. The file **usfutil.c** contains the symbol `CASE_INSENSITIVE`, which is 1 by default. If `CASE_INSENSITIVE` is 1, any lower case Unicode characters will be converted to upper case. This means that the file name “**AbCd.txt**” is the same as “**ABCD.TXT**”. If you will be using file names that differ only by case, then you will need to set `CASE_INSENSITIVE` to 0, which will preserve case sensitivity.

NOTE: Case sensitivity only works for files that do not meet the DOS 8.3 length limit or those that have characters in their names that DOS 8.3 names do not permit.

Dynamic Memory Use

When calling ***mt_readdir()*** and using long file names, a 256-byte block of memory is dynamically allocated to build up the directory name. This block is immediately freed after being used.

Files Used for Configuring VFAT

In addition to the files listed in Chapter 4, *Configuring USFiles*, the following file determines how VFAT is used:

makefile	Defines `VFAT` and `FAKEUNICODE`. Found in **siosrc** directory.
usfutil.c	Specifies `CASE_INSENSITIVE`

NOTE: The VFAT and FAKEUNICODE symbols may be found in **config\sio.mak**.

D. USFiles for CompactFlash

Installing CompactFlash

USFiles for CompactFlash* (USFCF) is delivered on a single disk. To install USFCF: Insert the disk, change to the drive with the disk, and type **install**. Follow the instructions for installation. Install USFCF into the same directory where USFiles is installed.

Text Files

Along with source files, we provide several text files with important information. Please read all files in your installation directory that end in **.txt**. Information in these files is likely more recent than that found in the manual. Some files that may be of particular interest are:

vsnlog3.txt	USFiles for CompactFlash version information. Located in the **siosrc** directory.
enable.txt	Notes on enabling USFiles for CompactFlash. Located in the **siosrc** directory.
appnote.txt	Topics that should be considered when developing an application. Can be found in the **siosrc** or **siosrc\\<*cpu*>** directory.

This is not a comprehensive list of the possible text files, and not all releases have each of these files.

Overview of CompactFlash

USFiles for CompactFlash (USFCF) provides a driver for CompactFlash cards that integrates with the USFiles product. The driver has been specifically developed on two platforms:

- AMD Elan SC400
- RTE-V850E/MS1-PC with FB2215a CompactFlash Interface Board

The driver supports CompactFlash Cards in ATA mode or True-IDE mode as well as ATA Flash Cards. The driver developed for the Elan board has code to initialize a PCMCIA controller, which is compatible with the Intel 82365 controller. This initialization is done when the first file is opened on the device.

After this initialization, the card is then accessed via the **lbahddrv** functions.

Configuration

In order for USFiles to recognize and use the CompactFlash driver, you must edit **config.mak** to include `usf` and `usfcf` in the `PRODLIST`. The CompactFlash driver requires direct hardware access, so when using the i8086 driver, be sure to have `#define USEBIOS` commented out in **siosrc\sioconf.h**. When using another processor, `USEBIOS` has no effect.

You will need to add a device to the device table to represent your CF card. To specify the CF card as device “**C:**”, you can use:

```
&pcparmC,                  /* device dependent data */
”C”,                       /* name */
FM_PCFM,                   /* device type = PC device */
0xf,                       /* bits: text write read */
0x80,                      /* unit# */
0,                         /* partition */
(DRIVER *)&lbadrv_s,       /* pointer to driver */
&pcfm,                     /* pointer to file manager */
NULL,                      /* pointer to FILE */
0,                         /* flags */
0,                         /* # open paths (RAM) */
```

Be sure that these lines are present in **devtab.c**:

```
PCFM_PARM pcparmC;
extern struct driver_p const lbadrv_s;
```

Testing

Once configuration has been done and USFiles is rebuilt, the first partition on the CompactFlash card will be recognized as C: (if you use the device table entry above). In order to test the CompactFlash card on the V850 board using **usftest**, ***main()*** must be modified in **usftest.c**. The line `defpath = “r:”;` should be changed to `defpath = “c:”;` so that the CompactFlash card is used instead of the RAM drive. Running **usftest** on the Elan board allows you to specify which drive to test at the command line.

See also: Chapter 1, *Getting Started*, for more details.

After **usftest** is run, it leaves all of its files on the CompactFlash card. In order to run it again, these files must be deleted. A utility called **wipe** is included with USFCF to delete all the files on **c:**, the default CompactFlash card. If you change this mapping and still want to use **wipe**, you should change **wipe**, lest you erase the wrong drive.

Not Supported

Support for PCMCIA on a Personal Computer uses various software layers, which are designed specifically to operate with the Windows operating system. USFiles does not support these layers, which include socket services, card services, and hardware drivers.

Other PCMCIA memory cards may be compatible with USFiles, but they will require customization by the user. For example, a customer may want to configure a PCMCIA controller to allow memory-mapped access to an SRAM PCMCIA card with battery backup. Once the PCMCIA controller is configured, the USFiles **ramdrv.c** driver can be used to access the SRAM card.

Linear flash cards are used with Flash Translation Layer (FTL) software that allows applications to access the linear flash card as a standard ATA disk drive. The FTL software takes care of the special access requirements of the linear flash cards, such as the requirement that the flash memory be erased in blocks. USFiles does not support the FTL capability. Therefore, a customer wanting to use linear flash cards with USFiles would have to write a driver for the device.

E. USFiles for CD-ROM

Installing USFiles for CD-ROM

USFiles for CD-ROM (USFCD) is delivered on a single disk. To install USFCD insert the disk, change to the drive with the disk, and type **`install`**. Follow the instructions for installation. Install USFCD into the same directory where USFiles is installed

Source Files

USFiles for CD-ROM consists of primarily two files: **cdfm.c** (in **siosrc**), the CD-ROM file system manager, and **cdromdrv.c** (in **siosrc\i8086**), the ATAPI CD-ROM driver.

Text Files

Along with source files, we provide several text files with important information. Please read all files in your installation directory that end in **.txt**. Information in these files is likely more recent than that found in the manual. Some files that may be of particular interest are:

vsnlog4.txt	USFiles for CD-ROM version information. Found in the **siosrc** directory.
cdreadme.txt	Information specific to the USFiles for CD-ROM release. Found in the **siosrc** directory.
appnote.txt	Topics that you should consider when developing an application. May be in either the **siosrc** or **siosrc\i8086** directory.

This is not a comprehensive list of the possible text files, and not all releases have each of these files.

Overview of CD-ROM

There are several varieties of CD-ROM recording standards. USFiles for CD-ROM supports the CD-ROM format (as opposed to the CD-ROM/XA format, for example). The supported format only has sectors with 2048 bytes of user data.

CD-ROM Driver

The CD-ROM driver (**cdromdrv.c**) that is provided works for ATAPI devices. These are connected via IDE cables. No other CD-ROM drives (e.g. SCSI) will work with this driver. At initialization we instruct the CD-ROM drive to use its default PIO transfer mode.

We have tested the driver with a Hitachi CDR-7730 4x drive and with Matsushita CR-583 8x and 40x drives. We have encountered problems with a BTC 40SB drive, which we have not yet resolved. If there are difficulties with your CD-ROM device and our driver, please contact us.

CD-ROM File Manager

USFiles for CD-ROM file manager (**cdfm.c**) supports ISO 9660 CD-ROMs recorded at interchange level 1 (each file contains only one file section, and file names comply with the DOS 8.3 convention). In addition, we support CDs recorded using Microsoft Joliet Extensions. These extensions allow longer paths, file names, and the use of Unicode characters.

Multisession CD-ROMs

The CD-ROM driver code handles multisession disks. It is configured to only read from the last recorded session on the disk. With modifications another session could be selected.

Basics of the ISO 9660 File System

The ISO 9660 file system differs considerably from the DOS FAT file system, which is why a new file manager had to be developed. This section will outline the basic items found in the ISO 9660 file system. A complete description of the ISO 9660 file system can be purchased from ISO or ANSI.

Volume Descriptors

The volume descriptors are analogous to the DOS file system BPB. These define the layout and size of the CD-ROM. There are five defined volume descriptors, but only three are recognized by **cdfm.c**. These are:

- Primary Volume Descriptor
- Supplementary Volume Descriptor
- Volume Descriptor Set Terminator

Primary Volume Descriptor

The Primary Volume Descriptor (PVD) lays out the CD-ROM with the ISO 9660 file system. It specifies the size of the CD-ROM, the location of the Path Tables, the root directory record, and various other items that are largely unused by USFiles for CD-ROM.

Supplementary Volume Descriptor

The Supplementary Volume Descriptor (SVD) provides the same details as the PVD, but it allows for variations to the ISO 9660 specification. In particular, the SVD can be used to specify a
CD-ROM that uses the Microsoft Joliet Extensions. The Path Table and root directory entry that the SVD point to will record names in Unicode if Joliet Extensions are used. A particular field in the SVD (Escape Sequences) identifies the file system used by a specific SVD.

Volume Descriptor Set Terminator

The Volume Descriptor Set Terminator is used to indicate the end of the sectors containing volume descriptors. After this sector follow the remaining file system structures (Path Tables, directories, and files).

Path Table

The Path Table specifies in which sector each directory begins. This speeds up searching for a file, because it limits the number of sector reads required. A Path Table Entry includes (among other things):

- Location (sector) of directory
- Parent directory number
- Directory name

The parent directory number is needed to differentiate between directories with the same name but different parents. For example, the directories **TEST1\SUBDIR** and **TEST2\SUBDIR** would have the same name in the Path Table, but they would have different parent directory numbers.

NOTE: A Path Table Entry can cross a sector boundary.

Directory Records

A directory record is used to define each file or directory. The directory record for the root directory is specified in the volume descriptor. A directory is composed of the directory records for each file or directory contained within it. The directory record contains these items, as well as others:

- Location (sector)
- Size
- Flags
- Name
- System use field

The length of the name is not known in advance; it is also specified in the directory record. USFiles dynamically allocates the space to hold the file name when a file is opened. USFiles releases that space when a file is closed, but you should be aware that if you have many files with long names open simultaneously, you may require a large heap.

The same thing is true for the system use field. This field can contain anything, but USFiles for CD-ROM does nothing with it. It will save this field in the `CD_DIR_ENTRY` structure associated with `CDFM_FSP`. These structures are defined in **mtio.h**, and the system use field can be accessed via the ***cdfm_read_su()*** function.

The only bit in the flags field that we make use of is bit 1 (where bit 0 is the lowest). If bit 1 is set to 1, then the directory record describes a directory. Otherwise it represents a file.

NOTE: A Directory Record cannot cross a sector boundary.

Navigating the File System

When a file on a CD-ROM device is opened by USFiles, the steps used to find the file are:

1. Read the volume descriptor.
   If using Joliet Extensions, try SVD first, then PVD (if no SVD present).
   If not using Joliet Extensions, only try PVD.
2. Find the proper directory in the Path Table.
3. Find the proper directory entry in the directory.
4. Go to the sector indicated by the directory entry.

The USFiles Implementation

There are several restrictions imposed by the ISO 9660 file system, but since USFiles does not record CD-ROMs, we tend to ignore some of these.

Allowed ISO Characters

The ISO 9660 file system allows these characters:

- A to Z (upper case)
- 0 to 9 (numerals)
- _ (underscore)

Allowed Joliet Characters

Joliet Extensions permit the use of all Unicode characters except control characters and those listed below, but USFiles expects names to be specified in ASCII and/or Shift-JIS. We are limited to characters that can be represented by these means. If Joliet Extensions are being used, we translate the ASCII and Shift-JIS characters to Unicode. The characters **not** allowed with Joliet Extensions are:

* / : ; ? \

File Name Lengths

Without Joliet Extensions, USFiles conforms to Interchange Level 1 of the ISO standard, which means that 8.3 file names are used. If Joliet Extensions are in use, then the length of the file name plus the length of the extension shall not exceed 128 bytes (64 Unicode characters).

Directory Name Lengths

Without Joliet Extensions an 8-character directory name is the limit. If Joliet Extensions are in use, then the length of the directory name shall not exceed 128 bytes (64 Unicode characters).

Extensions for Directory Names

When using Joliet Extensions, directories may have an extension. For example, **dir.tmp** is an allowed directory name with Joliet Extensions, but not for ISO 9660 alone.

Directory Levels

ISO 9660 allows for only eight levels of directories. For example `G:\D1\D2\D3\D4\D5\D6\D7\D8\FILE.TXT` is allowed, but no subdirectories are allowed in `D8`. USFiles does not test for this. When using Joliet Extensions, this limitation is removed. In both cases, the sum of the following items must be less than 255:

- Length of the file name
- Length of all relevant directory names
- Number of relevant directories

Again, USFiles does not test for this condition.

Other Items

We do not make use of the following items covered under the ISO standard:

- Files recorded in interleaved mode
- Use of extended attribute records
- Boot record volume descriptors
- Volume partition volume descriptors
- Directory structures recorded over multiple disks (Volume sets)
- CDs recorded with sector sizes other than 2048 bytes

This seems to be largely consistent with Microsoft’s handling of these options with MSCDEX (Microsoft CD Extensions) for DOS, with the possible exception of CDs recorded with different sector sizes.

Configuring USFiles for CD-ROM

Including CD-ROM Support

To include the CD-ROM support, the `PRODLIST` in **config.mak** must include `usfcd`. To avoid conflicts with a BIOS, you must be sure that no device in the device table uses the BIOS driver (**biosdrv_s**). The ISO 9660 Level 1 standard requires that file names on CDs comply with the DOS 8.3 convention. In addition to this, USFiles supports the use of Joliet Extensions, which allow (among other things) the use of long file names on the CD. Enabling long file names for the CD file manager is the same as enabling long file names for the PC file manager. This is accomplished by setting VFAT to 1 in **siosrc\makefile**.

See also: Appendix C, under *Using VFAT*, for information on Kanji character support.

Devices

To make use of the CD-ROM device, an appropriate entry must be made in the `device_tab[]` found in **devtab.c**. This is a sample entry:

```
(PCFM_PARM *)&cdparmG,  /* device parameter table pointer */
”G”,                  /* Device name */
FM_CDFM,              /* device type */
0x5,                  /* capabilities, 0x5 = read + text mode */
0,                    /* unit number 0=master, 1=slave */
0,                    /* partition number */
(DRIVER *)&cdromdrv_s,  /* Pointer to driver structure */
&cdfm,                /* Address of filemanager structure */
NULL,                 /* Unused for CD */
0,                    /* flags */
0                     /* number of open paths */
```

The device name (in this example `G`) is determined by your configuration (i.e. the number of hard disks, partitions, and CD-ROM drives). Also make sure that the variable `cdparmG` (or some other appropriate name) is defined globally in **devtab.c**. This is done with:

```
CDFM_PARM cdparmG;
```

Each CD-ROM device requires its own unique parameter variable.

In addition to specifying that CD-ROM support is included, the user must indicate on which IDE channel the CD-ROM drive resides. The default setting is for the primary IDE channel, but this can be changed by uncommenting this line in **siosrc\sioconf.h:**

```
#define CD_SEC
```

If the CD-ROM is used in conjunction with a hard drive, the hard drive must be on the primary IDE channel, and we recommend that the hard drive be the master device.

Table E-1 describes the possible CD-ROM drive configurations.

Table E-1: CD-ROM Drive Configuration

IDE Cable	**Master/Slave**	**Unit Number (userio.h)**	**CD_IDE (makefile)**
Primary	Master	0	1
Primary	Slave	1	1
Secondary	Master	0	2
Secondary	Slave*	1	2

* We have not tested the CD-ROM drive as a slave on the secondary IDE cable, nor have we tested more than one CD-ROM drive on a system.

Buffers

The user can determine how many buffers to use with the CD file manager. With the CD sector size of 2048 bytes, having many buffers may not be feasible for certain applications. The number of buffers is determined by `NUMCDBUFS`, which is in the file **siosrc\sioconf.h.** The value of `NUMCDBUFS` is set to 3 as the default.

Memory

In addition to statically defined sector buffers, there are a few CDFM routines that dynamically allocate memory to read in specific CD sectors (such as the sector containing the Volume Descriptor). This memory is freed after the necessary information has been acquired, but be aware that a suitable amount of memory must be available for this use. Each sector is 2048 bytes, and if multitasking is used, only one task will be allocating this amount of memory at a time. Memory for directory entry file names is also dynamically allocated. Be aware of this if you will have many files open at one time.

Mixed-case File Names

We have encountered some CD-ROM disks that use only a Primary Volume Descriptor (meaning that they are presumably ISO 9660 compliant), but have file and directory names that were recorded in mixed-case. By default, we change all file and directory names referred to by a PVD to upper case. If you do not want all names changed to upper case, then comment out the line `#define ALL_UPPER` in the file **cdfm.c**. Doing this will make your PVD file and directory names case sensitive.

If you are using Secondary Volume Descriptor information (long Unicode file names described by the Joliet Extensions), we can still turn on or off case-sensitivity. To allow for case-sensitive names for the long file names, the line `#define CASE_INSENSITIVE 1` (in **usfutil.c**) must be changed to `#define CASE_INSENSITIVE 0`. These options are summarized in Table E-2.

Table E-2: Case-sensitivity Options

Name Type	**Case Sensitive**	**Case Insensitive**
Long File Names (**usfutil.c**)	#define CASE_INSENSITIVE 0	#define CASE_INSENSITIVE 1
Short (ISO) File Names (**cdfm.c**)	/* #define ALL_UPPER */	#define ALL_UPPER

The reason that we have two methods here is that `CASE_INSENSITIVE` also governs how the long file names on diskettes or hard disks are handled. `ALL_UPPER` only affects CD-ROM files referred to by the Primary Volume Descriptor.

Files Used to Configure USFiles for CD-ROM

In addition to those files and configuration parameters mentioned in Chapter 4, *Configuring USFiles*, the following files are used to configure USFiles for CD-ROM:

cdfm.c	Has `ALL_UPPER.`
config.mak	Set `usfcd` and `usf` in **config.mak.**
devtab.c	Has `device_tab[].`
makefile	Has `VFAT` and `FAKEUNICODE`. Found in **siosrc** directory.
sioconf.h	Has `CD_SEC`. Found in **siosrc** directory.

NOTE: In future releases, `VFAT` and `FAKEUNICODE` may be in **config\sio.mak**.

Testing (cdfmtest

We tested a Hitachi CD-ROM drive with model number CDR-7730 under the following configurations:

- CD-ROM Master on Primary IDE channel (no slave)
- CD-ROM Slave on Primary IDE channel (Hard drive master)
- CD-ROM Master on Secondary IDE channel (no slave)

Our test program copies specified files from a CD-ROM disk to a recordable disk. We have used both diskettes and hard drives for testing purposes. The files on the CD-ROM are then compared to the copied file with various read methods. The stream I/O commands are also tested on the CD-ROM files to ensure that acceptable commands function properly and unacceptable commands return the appropriate errors.

This test program is provided for your use as well. The source code is found in **cdfmtest.c**. To use it with a CD-ROM of your own, follow these steps:

1. Select a few files on the CD to open as text files. Enter their full path names (excluding drive letter) in array *`*textFile[]`* in **cdfmtest.c.**
2. Select a few files on the CD to open as binary files. Enter their full path names (excluding drive letter) in array *`*binFile[]`* in **cdfmtest.c**.

NOTE: Be careful in choosing files. Since we will copy files to a recordable disk, you must be aware of file sizes so the disk does not fill up.

3. Select a directory from the CD. Enter its full path name (excluding drive letter) in *`directory[]`* in **cdfmtest.c.**
4. Save the file.
5. Configure the device table in **userio.h** to match your hardware.
6. Compile **cdfmtest**. If using Opus **make**, you may use **omake cdfmtest.**

7. Run **cdfmtest** by entering **`cdfmtest <cd> <dest>`**. The symbol **`<cd>`** represents the CD-ROM drive as configured in **userio.h**. The default is **G:**. Specifying **`<dest>`** indicates to which drive the files will be copied. The default is **R:** if RAM disk is included or **A:** if not. Entering **cdfmtest** with no arguments will use the default drives.

WARNINGS: The **cdfmtest** requires a lot of memory, and may not work with the RAM disk. If you indicate a hard drive as the destination drive, you must reboot your machine after the test completes to ensure that DOS does not corrupt your hard drive. This test cannot be run in a DOS window in Windows95/98. It can only be run from DOS. It is recommended that you reboot the machine after the test in any case. We have noticed that the hardware may get confused after the test is completed.

Initialization

In testing the CD-ROM driver and file manager, we noticed that particular CD-ROM drives took a while to initialize (30 seconds or more). Initialization is done when the first file on the device is opened, so you may notice that it takes quite some time to open the first file on a CD-ROM device, but afterwards it is much faster. We have not determined why this happens. Some devices do not have this behavior.

If you consistently receive the `ETONRDY` error when the CD-ROM drive is being initialized, then you may want to increase the value of `CDROM_RETRY_TIMEOUT` in **cdromdrv.c**.

Additional Functions

These additional functions are provided with the CD file manager:

cdfm_invalidate_buffers	Invalidates all buffers on a device
cdfm_esc_codes	Retrieves escape codes field from Supplementary Volume Descriptor
cdfm_len_su	Returns length of system use field from file's directory entry
cdfm_read_ear	Reads Extended Attribute Record for specified file
cdfm_read_su	Returns contents of system use field from file's directory entry
cdfm_vol_info	Retrieves the indicated field from Volume Descriptor

Three of these (***cdfm_esc_codes()***, ***cdfm_vol_info()***, and ***cdfm_read_ear()***) are commented out at the end of **cdfm.c**. If you wish to use these, you must uncomment this section of the file.

cdfm_esc_codes

Retrieves escape codes field from Supplementary Volume Descriptor.

```
int cdfm_esc_codes(DEVICE *devp, byte *retBuf);
```

devp	*pointer to device*
retBuf	*address of return buffer*

The escape codes field of a Supplementary Volume Descriptor is 32 bytes long. The ***cdfm_esc_codes()*** function reads the SVD and copies that field to `retBuf`. The escape codes field is used to identify which volume type (e.g. Joliet Extensions) the SVD describes.

WARNING: This function has not been tested!

Return Value

0	success
EUNSUP	disk only has a PVD
EBADARG	sector read is not a volume descriptor
ENOMEM	no memory for buffer
ENOTJOLIET	there is an SVD, but it is not a Joliet SVD
ENODESC	volume descriptor not found
driver error	

Example

```
DEVICE *devp;
char codes[32];

if( cdfm_esc_codes(devp, (byte *)codes) )
   /* Some error */
else
   /* Do something with codes */
```

cdfm_invalidate_buffers

Invalidates all buffers on a device.

```
int cdfm_invalidate_buffers(DEVICE *devp);
```

devp	*pointer to device*

The ***cdfm_invalidate_buffers()*** function is provided for error recovery purposes. We can use this function to mark all buffers for the device as unused. This function differs from the ***pcfm_invalidate_buffers()*** function in that it can never return the value 1, since a CD-ROM disk cannot be written to.

See also: ***otherFilesOpen, invalidate_streams***

Return Value

0	success

Example

```
DEVICE *devp;

/* Disk has changed */
cdfm_invalidate_buffers(devp);
if(otherFilesOpen(devp))
    invalidate_streams(devp);
else
    /* No open files, so ignore error */
```

cdfm_len_su

Returns length of system use field from file’s directory entry.

```
int cdfm_len_su(MTFILE *fp);
```

fp	pointer to file

The purpose of the system use field is not defined by the ISO 9660 standard. The length of the system use field is not predefined, so this function provides a means of determining the amount of memory necessary to handle the contents of the system use field.

WARNING: This function has not been tested!

Return Value

Length of system use field

`EOF`	bad file pointer

errno Value

CD File Manager

EBADFP	file fp has not been opened

Example

```
MTFILE *fp;
  char *sysUse;
  int length;
  fp = mt_fopen(“G:\\test.txt”,”r”);
  length = cdfm_len_su(fp);
  if(length == EOF)
        /* Process error */
  else
        /* Allocate length bytes of memory
        ** to sysUse */
  if(cdfm_read_su(fp, sysUse))
        /* error reading */
  else
        /* Do something with sysUse */
```

cdfm_read_ear

Reads Extended Attribute Record for specified file.

```
int cdfm_read_ear(MTFILE *fp, CD_EXT_ATTR *record);
```

fp	*pointer to file*
record	*address of record storage*

The `CD_EXT_ATTR` structure is defined in **mtio.h** as:

```
typedef struct cd_ext_attr{
  CD_VOL_TIME   creDate;  /* File creation date and time */
  CD_VOL_TIME   modDate;  /* File modification date and time */
  CD_VOL_TIME   expDate;  /* File expiration date and time */
  CD_VOL_TIME   effDate;  /* File is valid after date & time */
  byte         *appUse;   /* Pointer to application use field */
  byte         *escSeq;   /* Pointer to escape sequences field */
  uint16        ownerID;  /* File owner number */
  uint16        groupID;  /* Owner’s group number */
  uint16        permissions;  /* File permissions */
  uint16        recLen;        /* Record length */
  byte          recFormat;    /* Record format */
  byte          recAttr;      /* Record attributes */
  byte          systemID[32]; /* System identifier */
  byte          systemUse[64];/* System use field */
  byte          version;      /* Ext. attr. rec. version */
  byte          escLen;       /* Length of escape sequences */
  byte          appLen;       /* Len. of application use field */
} CD_EXT_ATTR;
```

The ***cdfm_read_ear()*** function will fill a `CD_EXT_ATTR` structure for a specified file, if the file has an extended attribute record. The *`appUse`* and *`escSeq`* fields do not have predefined lengths, so we dynamically allocate them. Be sure to free those memory areas if your structure is deleted.

Details of the Extended Attribute Record are not provided here. If you need more details on this record, please refer to the ISO 9660 specification.

WARNING: This function has not been tested!

Return Value

0	success
EBADFP	file *fp* has not been opened
ENOMEM	no memory for buffer
driver read error	

errno Value

CD File Manager

EBADFP	bad file pointer

Example

```
MTFILE *fp;
CD_EXT_ATTR ear;

fp = mt_fopen(“G:\\test.txt”,”r”);
if( cdfm_read_ear(fp, &ear) )
```

```
     /* Some error */
   else
     /* Do something with ear */
```

cdfm_read_su

Returns contents of system use field from file’s directory entry.

```
int cdfm_read_su(MTFILE *fp, char *buf);
```

fp	pointer to file
buf	pointer to buffer for system use storage

The purpose of the system use field is not defined by the ISO 9660 standard. The length of the system use field is not predefined. You can use ***cdfm_len_su()*** to determine the necessary amount of memory for storage. The ***cdfm_read_su()*** function returns the contents of the system use field to the address indicated by `buf`.

WARNING: This function has not been tested!

Return Value

0	successful completion
EOF	bad file pointer

errno Value

CD File Manager

EBADFP	file *fp* has not been opened

Example

```
MTFILE *fp;
char *sysUse;
int length;
```

```
fp = mt_fopen(“G:\\test.txt”,”r”);
length = cdfm_len_su(fp);
if(length == EOF)
  /* Process error */

else
  /* Allocate length bytes of memory
  ** to sysUse */
if(cdfm_read_su(fp, sysUse))
  /* error reading */
else
  /* Do something with sysUse */
```

cdfm_vol_info

Retrieves the indicated field from Volume Descriptor.

```
int cdfm_vol_info(DEVICE *pDev, byte *retBuf,
               enum idFields fieldType);
```

devp	*pointer to device*
retBuf	*address of return buffer*
fieldType	*file name to read*

The `idFields` enumeration is defined in **mtio.h** as:

```
enum idFields { ussCDVolSetID
                ussCDCopyRtID,
                ussCDAbsID,
                ussCDBibID
};
```

Specifying *`ussCDVolSetID`* will not return a file name, but it will return the name of the Volume Set of which the CD-ROM is a member. This field is 128 bytes long.

The other three *`fieldTypes`* specify file names. Each one of these fields is 37 bytes. The symbol `ussCDCopyRtID` specifies the copyright file, `ussCDAbsID` indicates the abstract file, and `ussCDBibID` points to the bibliography file. These files are mentioned in the ISO 9660 specification.

WARNING: This function has not been tested!

Return Value

0	success
`EBADARG`	sector read is not a volume descriptor, or improper *`fieldType`*
`ENOMEM`	no memory for buffer
`EBADBPB`	there is an SVD, but it is not a Joliet SVD
driver read error	

Example

```
DEVICE *devp;
char volID[128],crID[37];

if( cdfm_vol_info(devp, (byte *)(volID), ussCDVolSetID) )
   /* Some error */
else
   /* Do something with volID */

if( cdfm_vol_info(devp, (byte *)(crID), ussCDCopyRtID) )
   /* Some error */
else
   /* Do something with crID */
```

Additional *errno* Values

When using the CD-ROM file manager, certain stream I/O function calls may set `errno` differently than the PC file manager. Table E-3 lists the functions for which the CD-ROM file manager may set `errno` differently than the PC file manager.

Table E-3: CD-ROM File Manager *errno* Codes

Function	***errno* Values**	**Description**
mt_fclose()	driver error	
mt_fgetc()	`ELOCKED`	Timeout waiting for access to file system
	`ENOBUF`	No buffer for sector.
	`driver error`	
mt_fgets()	`ELOCKED`	Timeout waiting for access to file system.
	`ENOBUF`	No buffer for sector.
mt_fopen()	`ELOCKED`	Timeout waiting for access to file system.
	`EBADARG`	Unsupported descriptor type accessed.
	`ENOMEM`	No memory for sector storage.
	`ENOTJOLIET`	SVD exists, but not Joliet compliant.
	`ENODESC`	Specified volume descriptor not found.
	`ENOTPT`	Path table sector not found.
	`EACCESS`	Trying to open a file as a directory, or vice versa.
	`ENOENT`	No entry for file found in directory.
	`ENOPATH`	Part of directory path not found.
	Driver error	

Table E-3 (continued): CD-ROM File Manager *errno* codes

Function	***errno* Values**	**Description**
mt_fprintf()	`ECAPERR`	Device not available for write.
mt_fputc()	`ECAPERR`	Device not available for write.
mt_fputs()	`ECAPERR`	Device not available for write.

mt_fread()	`ELOCKED`
	`ENOBUF`
	driver error
		Timeout waiting for access to file system.
		No buffer for sector.

mt_remove()	`ECAPERR`	Device not available for write.
mt_rename()	`ECAPERR`	Device not available for write.
mt_rmdir()	`ECAPERR`	Device not available for write.

Global Variables

These additional global variables are used when the CD-ROM file manager is added to USFiles:

`CDFM_BUFFER` *`cdfm_buf[`*`NUMCDBUFS`*`]`*	CD-ROM sector buffers
`byte` *`cdfm_agescale`*	Indicates when buffer age parameter wraps
`byte dirBuf[256]`	Used when calling ***mt_readdir()*** to hold the raw contents of a directory entry
`CD_DIR_ENTRY readEntry`	Used when calling ***mt_readdir()*** to hold the processed contents of a directory entry

CD-ROM Driver Functions

A `cdfm` device driver consists of these functions, which are typically used in this order:

init()	Initializes device
readTOC()	Reads CD-ROM table of contents
read()	Reads sector specified as a logical sector number
diskchange()	Reports if a disk has been changed

The CD-ROM driver structure is defined in **mtio.h** as:

```
struct driver_cd {
    int (*init)(DEVICE *);
    int (*read)(uint32, struct cdfm_buffer_s *);
    int (*diskchange)(DEVICE *);
    int (*readTOC)(DEVICE *);
};
```

For a specific instance of a driver, these routines will be given the above-mentioned names with a unique prefix prepended to them to designate the driver (e.g., ***cdromdrv_read()***).

The exact function performed by these routines depends upon what the file manager calling them expects. The division of responsibilities between the file manager and the device driver may be altered if a new file manager is developed. The expectations of the `cdfm` file manager are described in the following function descriptions.

Driver diskchange() function

```
int diskchange(DEVICE *devp);
```

The ***diskchange()*** function returns a non-zero value if a media change has been detected since the last read or write operation to the drive. This function should return a valid error code. It is possible that a timeout may occur while the drive is becoming ready after the CD-ROM has been changed, in which case an `ETONRDY` error may be reported.

Driver init() function

```
int init(DEVICE *devp);
```

The initialize function is called once for each drive controlled by the driver. It should do any initialization required by the device such as hardware reset, initialize interrupt vectors, etc. Zero is returned if successful, and a non-zero error code if not. If more than one drive is called, ***init()*** should keep a static flag to tell it that it has already been called so it can avoid repeating operations that should be done only once. The ***cdromdrv_init()*** function installs interrupt vectors, sets up the CD-ROM drive I/O mode, and tests to see if the device is ready.

Driver read() function

```
int read(uint32 logical_sect, CDFM_BUFFER *bufp);
```

The driver ***read()*** function reads the logical sector indicated into the buffer at *`bufp->buf`*, from the drive indicated by the *`bufp`* structure. Any other information required by the driver about the device can be found through the *`bufp`* structure. Parameters in *`bufp`* may indicate that a consecutive number of sectors are to be read, in which case this action should be taken.

If *`bufp->usrbuf`* is not `NULL`, then the ***read()*** function will read *`bufp->nsects`* sectors to the user's buffer at *`bufp->userbuf`*, instead of transferring a single sector to *`bufp->buf`*.

Driver readTOC() function

```
int read(DEVICE *devp);
```

The driver ***readTOC()*** function reads the CD-ROM's table of contents. This is used to determine the location of the last session of a multisession CD-ROM. The starting sector of the last session is stored in the device parameter *`session_start`* in the `CDFM_PARM` structure (see ***cdromdrv_readTOC()*** in **cdromdrv.c** and **mtio.h**). If you will not be using multisession disks, then *`session_start`* can simply be set to 16. This function will return 0 if the table of contents is successfully read, otherwise an error value should be returned.

Function Call Hierarchy

Table E-4 shows how the stream I/O functions map to the CD-ROM file manager and then to the driver. Not all stream I/O functions are shown, because they are not all appropriate for a CD-ROM device.

Table E-4: Function Hierarchy for CD File Manager

Stream I/O	File Manager	Driver
mt_clearerr()		
mt_fclose()	***cdfm_close()***	
mt_feof()		
mt_ferror()		
mt_fflush()	***cdfm_fmioctl()***	
mt_fgetc()	***cdfm_read()***	***read()***
mt_fgetpos()		
mt_fgets()	***cdfm_readln()***	***read()***
mt_fopen()	***cdfm_open()***	***init(), diskchange(), readTOC(), read()***
mt_fread()	***cdfm_read()***	***read()***
mt_readdir()	***cdfm_fmioct()***	
mt_fseek()	***cdfm_seek()***	
mt_fsetpos()	***cdfm_seek()***	
mt_ftell()		
mt_rewind()	***cdfm_seek()***	

Recommended Reading

For a detailed description of the ISO 9660 file system we recommend the specification document:

ISO 9660 : 1988
Information processing – Volume and file structure of CD-ROM for information interchange.

F. FAT32 File System

Overview

The FAT32 file system is heavily dependent on the DOS FAT12/16 file system. You should read Chapter 2, *File System Description*, before reading this appendix if you are not already familiar with the FAT file system.

This appendix will describe how to install and configure USFiles-32 and describe the differences between the following FAT32 and FAT12/16 structures:

- BPB
- Partition Table
- FAT
- Directory entries
- Root directory

There are also additional items contained in a FAT32 partition, and these will also be mentioned. Only one of these new items is of interest to USFiles.

Installation and Configuration

USFiles-32 is provided on a single disk. Make the drive containing the installation disk the current drive and type **install**. You will be provided with installation instructions.

To include support for USF-32 you will have to include `usf32` in the `PRODLIST` in **config.mak.**

Test Programs

USFiles for FAT32 is provided with the test program **f32test.c**. We recommend that you run **usftest** (provided as standard with USFiles) initially.

The **usftest** routines will exercise most of the functionality on a FAT32 partition, but there are some additional features of FAT32 that **usftest** does not test. These features are tested with **f32test.c**.

The **usftest** tests will likely take some time, since the last thing that it does is fill the disk. We recommend only running **usftest** on a partition that is set aside for USFiles testing.

After running **usftest**, you will have to make room on the drive to run **f32test**. FAT32 devices treat the root directory differently from FAT12/16 devices. Therefore, you can either reformat the drive, or remove the file **bigfile.tmp** from the drive's root directory. This should clear up the space necessary for the remaining tests.

The size of the root directory is not defined in advance, and clusters can be allocated to the root directory. The **f32test** ensures that additional clusters are allocated to the root directory when necessary for:

- adding a volume label, and
- adding a file or directory

There is one additional test in **f32test** that is by default disabled. This test creates a small file and moves it to the last available cluster on the drive to ensure that USFiles will access the full extent of the drive. To enable this test:

1. Set the macro `DO_LASTCLUSTTEST` in **f32test.c** to 1.
2. Remove (or comment out) the static label on the ***set_fat()*** function in **pcfmclus.c**.
3. Compile and run.

Modified Structures

This section describes the BIOS parameter block (BPB), the partition table, and the file allocation table (FAT).

BIOS Parameter Block (BPB)

The FAT32 BPB is used the same way as the FAT12/16 BPB, but there are additional fields included. Table F-1 outlines the fields in the BPB. Bold items indicate entries that are new to FAT32.

Table F-1: The FAT32 BPB

Byte in Sector (hex [decimal])	**Field Description**	**Comments**
`0B-0C [11-12]`	Bytes per sector	USFiles only supports disks with 512 bytes per sector.
`0D [13]`	Sectors per cluster	
`0E-0F [14-15]`	Reserved sectors	
`10 [16]`	Number of FATs	
`11-12 [17-18]`	Number of root directory entries	**Not used by FAT32.**
`13-14 [19-20]`	Total sectors in logical volume	Not used if volume size is greater than 32 MB.
`15 [21]`	Media descriptor byte	Stored, but not used.
`16-17 [22-23]`	Number of sectors per FAT	**Always 0 for FAT32.**
`18-19 [24-25]`	Number of sectors per track	

Table F-1 (continued): The FAT32 BPB

Byte in Sector (hex [decimal])	**Field Description**	**Comments**
1A-1B [26-27]	Number of heads	
1C-1F [28-31]	Number of hidden sectors	We have found that some, but not all, disk format utilities include prior disk partitions in this value.
20-23 [32-35]	Total sectors in logical volume	Used only if volume size is greater than 32 MB.
24-27 [36-39]	**Number of sectors per FAT**	
28-29 [40-41]	**Partition flags**	**Ignored**
2A-2B [42-43]	**File system version**	**Ignored**
2C-2F [44-47]	**Root directory starting cluster**	
30-31 [48-49]	**File system information sector**	
32-33 [50-51]	**Backup boot sector number**	
34-3F [52-63]	**Reserved**	
40 [64]	Physical drive number	Ignored (**new location**)
41 [65]	Reserved	Ignored (**new location**)
42 [66]	Extended boot signature record	Ignored (**new location**)
43-46 [67-70]	Drive serial number	(**new location**)
47-51 [71-81]	Volume label	**(new location**)

Partition Table

The FAT 32 Partition Table remains unchanged. With the addition of FAT32 support to USFiles, the two partition types `0Bh` and `0Ch` (see Table 2-3) are now supported.

File Allocation Table (FAT)

As the name indicates, the FAT entries for FAT32 consist of 32 bits. The upper four bits of each FAT entry are unused, though. Possible FAT entries are:

`00000000h`	Cluster free for use
`00000001h – 0FFFFFEFh`	Indicates next cluster for file
`0FFFFFF8h – 0FFFFFFFh`	Last cluster of file
`0FFFFFF0h – 0FFFFFF7h`	Cluster not usable

USFiles will neither read nor modify the upper four bits of a FAT32 entry.

Directory Entries

The FAT32 directory entries must indicate a 32-bit starting cluster. The additional two bytes are taken from previously reserved bytes. The FAT32 directory entry is described in Table F-2.

Table F-2: FAT32 Directory Entry

Relative Byte Position (hex [decimal])	**Field Description**	**Comments**
`00-07 [0-7]`	File name	Base of file name
`08-0A [8-10]`	File extension	
`0B [11]`	File attribute	See Table 2-5

`0C [12]`	Reserved	
`0D [12]`	Creation time in 4-millisecond units	VFAT only
`0E-11 [14-17]`	Time and date created	VFAT only
`12-13 [18-19]`	Date of last access	VFAT only
`14-15 [20-21]`	**High bytes of first cluster for file**	**FAT32 only**
`16-19 [22-25]`	Time and date created	
`1A-1B [26-27]`	Low bytes of first cluster for file	
`1C-1F [28-31]`	File size	

With FAT32, the root directory is now allocated like any other directory, as is discussed in the next section, so it has a non-zero cluster number associated with it. Any subdirectory that resides in the root directory has a directory entry (‘..’) that refers back to the root directory. The cluster number indicated in this entry is still zero, even though the root directory has a non-zero cluster number with FAT32.

The Root Directory

The root directory on a FAT32 partition is allocated like any other file or directory. It has a starting cluster (generally 2), and it has no limits on the number of sectors that it can occupy. The root directory starting cluster is provided by the FAT 32 BPB.

New Structures

Several more sectors are used in FAT32 partitions for file system maintenance. As is indicated in Table F-1, there is a sector that has a copy of the boot sector. This is ignored by USFiles. The bootstrap code for FAT32 partitions now spans more than one sector, because the BPB has more entries in it now. USFiles does nothing with bootstrap code, so this is ignored.

File System Information Sector

The one new structure that USFiles maintains is the File System Information Sector, which keeps track of the number of free clusters and the last sector allocated on the disk. The information sector number is stored in the FAT32 BPB and is recorded as the number of sectors past the BPB sector. The significant elements in the sector are shown in Table F-3.

Table F-3: FAT32 File System Information Sector

Byte in Sector (hex [decimal])	**Field Description**	**Comments**
`1E4-1E7 [484-487]`	File system information sector signature	`61417272h` stored Little-Endian
`1E8-1EB [488-491]`	Number of free clusters	
`1EC-1EF [492-495]`	Last cluster allocated	

There are additional codes in the sector, but USFiles only checks the bytes from `1E4h` to `1E7h` for identification. USFiles updates the number of free clusters and the last cluster allocated. The number of free clusters is returned by ***free_clust_cnt()***.

Limitations on USFiles-32

Directory entries in the FAT32 file system only allow 32 bits to specify the file size. Even though the disk geometry may allow it, we have to limit the file size to 4 GB. If the user attempts to write a file larger than this, then the error code `EBIGFILE` will be returned.

The FAT32 file system BPB specifies the following items that USFiles ignores:

- Drive flags to signal whether FAT mirroring is enabled (USFiles will always mirror the FAT)
- File system version number
- Backup boot sector

Using free_byte_cnt

The function ***free_byte_cnt()*** returns the number of free bytes on a disk as an unsigned 32-bit integer. For a FAT32 partition, a 32-bit integer may not be large enough to store the number of free bytes. Use this function with caution.

The functions ***free_kb_cnt()*** and ***free_clust_cnt()*** might provide the most reliable means of determining free space on a FAT32 volume. FAT32 partitions that are smaller than 8 GB have 4 KB per cluster. Be aware that FAT32 partitions may be as large as 2 TB, in which case each cluster has 32 KB.

G. Error Codes

USFiles Error Codes

This is a summary of the error codes that USFiles functions may signal. The error will usually be reported through the variable *`errno`*.

Table G-1: USFiles Error Codes (from **mtio.h**

Label	**Decimal Value**	**Meaning**
EWRGFMT	1	Wrong disk format
ECAPERR	2	Device capabilities error
ENOMEM	3	No memory available
ENMFILE	4	`NUMSTREAMS` limit has been reached
ENOENT	5	No file entry found in directory
EDSKCHG	6	Disk change error has occurred
ENOPATH	7	Part of the path was not found
EATEOF	8	File pointer is at EOF
EBADCLUST	9	Bad cluster found
ENOBUF	10	No file buffer is available
EBADNAM	11	File name too long or contains bad characters
ENOTDIR	12	Name specified is not for a directory
EACCESS	13	Trying to open directory as file or vice versa
ERDONLY	14	Trying to open read-only file for write

Table G-1 (continued): USFiles Error Codes (from **mtio.h**)

Label	**Decimal Value**	**Meaning**
EDSKFUL	15	Disk is full, no more clusters to allocate
ERDFULL	16	Root directory is full
EBADFP	17	Bad file pointer or device not initialized
EUNSUP	18	Device does not support requested operation
EBADARG	19	Bad function argument supplied
EBADPOS	20	Seeking past allowed file boundaries
EEXIST	21	Trying to create a directory that already exists
EBADPART	22	Bad partition signature encountered
EPARTID	23	Unsupported ID byte in partition entry
EISOPEN	24	Path already open
EUNINIT	25	Trying to access uninitialized RAM drive
EWRGDEV	26	Attempted rename to different device
ENOTMT	27	Subdirectory is not empty
EISATT	28	Keyboard is already attached
ENOTATT	29	Keyboard is not attached
EBADASS	30	Keyboard cannot be assigned at this location
EWRTPRT	31	Attempted to write to write-protected disk
ENORESP	32	No response from drive (door may be open)

Table G-1 (continued): USFiles Error Codes (from **mtio.h**)

Label	**Decimal Value**	**Meaning**
`ENOTFND`	33	Address mark or sector not found
`EBADSECT`	34	Bad sector encountered
`EDMABND`	35	DMA memory-boundary crossing error
`EIOERR`	36	Miscellaneous I/O error
`EBADSIZE`	37	Pipe size of zero requested
`EMEMERR`	38	Memory release error
`EBADFAT`	39	FAT sectors not readable
`EBADBPB`	40	Bad BPB sector
`ELOCKED`	41	Timeout waiting for access to file system
`ECTLFAIL`	42	Controller failure
`EBIGPATH`	43	Path name too long
`ENODESC`	44	CD-ROM Volume Descriptor not found
`ENOTJOLIET`	45	CD-ROM has Supplementary Volume Descriptor, but it is not for Joliet Extensions
`ENOTPT`	46	Sector does not contain path table
`ENODISK`	47	No disk in CD-ROM drive (door may be open)
`ETONRDY`	48	Timeout occurred while waiting for device to become ready
`EDEVRST`	49	Device reset occurred
`EBIGFILE`	50	File size cannot exceed 2^{32} bytes

Several of these error codes do not apply specifically to USFiles, and error codes 44 - 49 only apply to USFiles for CD-ROM. The error code `EBIGFILE` is only used by USF-32. With expansion of USFiles support, more error codes might be added. Please examine the file **ussio.h** for the most recent list of error codes.

H. Index