# ERTFS 4.4zb Pro/Basic

## Table of Contents

*Revised March, 2007*

## Table of Contents

## SECTION 1: GETTING STARTED WITH ERTFS

### Start by installing ERTFS on your system.

The 'C' source code and header files reside in the install directory. A subdirectory named **targets** contains sample porting files that may be used to build ERFTS for several platforms.

Build ERTFS by following these steps.

- Read through the porting and configuration guide carefully.
- Edit **PORTCONF.H** to set the byte order. (See Section 9)
- Edit **PORTCONF.H** making sure all device drivers are disabled except for **INCLUDE_ROMDISK** and **INCLUDE_RAMDISK**.
- Compile all files. The only requirement of the compiler is that it pack structures. We provide sample Microsoft and GNU make files if you wish to use them. A sample MS Visual 'C' project file to build the Windows prototype project is provided in subdirectory *rtfsdemo*. If you choose not to use these make files, your requirements are simple; compile all of the files and put all of the objects except **portmain.o** into a library. If you want to build the ERTFS test shell as a standalone program, link **portmain.o** against the library you just created. If you wish to just use the API, add the library you just created to your project's library list. You may, if you wish, call the test/command shell as a subroutine from your task. Be sure to call **pc_ertfs_init()** before you call the test shell or any other API function.
- If at all possible, hook a console device to the routines named **rtfs_port_tm_gets()** and **rtfs_port_puts()**. This will allow you to run the ERTFS test environment interactively. This makes the process of bringing up a new port much more pleasant.
- Edit the rest of **portkern.c** to support your RTOS environment. (See Section 9 for more information.)
- One you have a basic build you may now customize ERTFS for your application and target environment. Tune your configuration, balancing resource usage with performance. See Section 3 for more information on configuring ERTFS.
- Enable support for selected device drivers in **PORTCONF.H**, and edit the sections of **PORTKERN.C** and **PORTIO.C** that must be modified to support that device in your environment. See Section 17 for a complete explanation.
- Implement your own device drivers if necessary. If the ERTFS package does not include a driver for your device type, you may be forced to implement your own driver. See Section 25 for a complete explanation of how to implement your own device drivers.
- Create populate your own rom-disk images for your target application.
- Write your applications code.

*Note: Before you use ERTFS, you must call the initialization routine **pc_ertfs_init()**.*

## SECTION 2: THE SOURCE CODE STRUCTURE

The ERTFS source code package is comprised of demonstration code, core file system code, language localization files, file system feature set specific files, configuration files, device drivers and target specific porting files. For any files, the filename's prefix identifies what type of file it is. Demonstration code starts with **app**, code that runs only in a windows demo environment start with **win**, files that contain exported APIs start with **api**, core system files start with **rt**, character set and localization files start with **cs**, drivers start with **dr**, RTFSPRO components start with **pr**, and files that need porting start with **port**.

This section contains a description of each of these files.

**PUBLIC HEADER FILES**

**rtfsapi.h** - The ERTFS exported application programmers interface (API). This header file must be included in all applications code that uses the published ERTFS API.

**PRIVATE HEADER FILES**

**rtfs.h** - The ERTFS header file included by all ERTFS source files.

**rtfsconf.h** - Compile time configuration parameters for ERTFS. Rtfsconf.h controls inclusion of packages such as **FAT32**, **VFAT**, **UNICODE** and **JIS** character sets ***(FAT32, VFAT, Unicode, and JIS are included with ERTFS Pro only)***.

**portconf.h** - Compile time configuration parameter for ERTFS. Controls device driver inclusion and byte order.

**rtfspro.h** - Defines and structures for RTFSPRO Enhancements.

**DEMONSTRATION SAMPLE CODE AND UTILITY PROGRAMS**

**appdemo.c** - Small entry point that initializes ERTFS and then calls the command shell.

**appcmdsh.c** - Interactive command shell like DOS's **command.com**. Provides many examples of using the ERTFS API. This application will run on any target that can get and put lines of text to a console. The test shell has a lot of features that are useful for debugging and file system maintenance. Your release contains a Windows version of the shell. Please experiment to help you decide if you want to use it. For more information about the command shell subroutine, see the documentation Section 26 and in the Command Reference in Appendix H.

**winhdisk.c** - Host disk driver. Emulates a disk drive in a host file. Supported only in the win32 prototyping environment.

**winmkrom.c** - Tool that populates a host disk volume by copying files from a Windows subdirectory tree and creating a virtual disk from the data. The tool can then export this virtual file into a header file that may then be used as a rom-disk image by the rom-disk driver. These utility functions are provided pre-built in the **rtfsdemo.exe** utility for windows in the **tools** subdirectory. This file runs only in the win32 prototyping environment.

**PORTING FILES**

These files require modification when moving ERTFS to a new environment. Several implementations are provided in the **targets** subdirectory. To select a target, copy its porting files to the main tree. To create a new port, copy one of the existing ports to the main tree and edit the files for your environment. If you prefer to start from "scratch," the subdirectories named **template.io** and **template.krn** contain completely generic implementations of these files that you may copy and modify.

**portio.c** - Register access functions for external devices. This file must be modified if you are porting any of the ERTFS hardware device drivers to a new platform.

**portkern.c** - Kernel and clock services. This file must be modified if you are porting ERTFS to a new operating system, CPU or target platform. See Section 10: Porting Overview.

**portmain.c** - Run time entry point. This may need to be modified for your RTOS and development environment. It is needed only if you are linking the RTFS demo stand-alone. If you will link ERTFS to an already running system then you won't need **portmain.c**. Instead you need to make sure that you call **pc_ertfs_init()** first and then begin calling the API. If you want to run the test shell from an already running system please call **rtfs_app_entry()**, this routine will initialize ERTFS and then start the test shell interpreter.

**SOURCE CODE FOR USER MODIFIABLE PUBLIC FUNCTIONS**

**apicnfig.c** - User modifiable code that controls resource usage of ERTFS at run time. See Section 3 for a thorough discussion of this topic.

**Apiinit.c** - User modifiable code that controls what device drivers ERTFS will control and what drive letters will be assigned to them. See Section 6 for a thorough discussion of this topic.

**SOURCE CODE FOR USER API CALLS; SEE THE API SECTION**

apickdsk.c apideltr.c apienum.c apifilio.c apifilmv.c
apifrmat.c apigetwd.c apigfrst.c apiinfo.c apimkdir.c
apirealt.c apiregrs.c apisetwd.c apistat.c apiwrite.c

**SOURCE CODE FOR ERTFS PRO USER API CALLS**

prapipro.c

**SOURCE CODE FOR THE ERTFS CORE FUNCTIONS**

**prblock.c** - Block buffer pool logic
**rtdevio.c** - Device access, auto-mount and media check logic
**rtdrobj.c** - directory object support
**rtlowl.c** - essential functions
**rtutbyte.c** - parsing code
**rtutil.c** - parsing and other utility code
**rtregion.c** - Fat region management code
**rtfat16.c** - Fat management code
**rtfat32.c** - Fat management code ***(included with ERTFS Pro only)***
**rtfatxx.c** - Fat management code
**rtnvfat.c** - Long file name support
**rtvfat.c** - Short file name support ***(included with ERTFS Pro only)***
**rtkernfn.c** - portable OS resource helper functions
**rttermin.c** - portable terminal helper functions

**SOURCE CODE FOR LOCALIZATION AND MULTIPLE CHARACTER SET SUPPORT**

**csascii.c** - ASCII character set specific code.
**csjis.c** - Japanese character set specific code ***(included with ERTFS Pro only)***
**csjistab.c** - Japanese character set specific code ***(included with ERTFS Pro only)***
**csunicod.c** - Code to support 16 bit Unicode character sets ***(included with ERTFS Pro only)***
**csstrtab.c** - Modifiable string table for localization
**csstrtab.h** - String table identifiers for localization

**SOURCE CODE FOR ERTFS FAILSAFE FEATURE**

**prfsapi.c** - User API level source code for ERTFS-Pro FailSafe functions
**prfscore.c** - ERTFS Pro FailSafe internal routines
**prfsnvio.c** - ERTFS Pro FailSafe journal file access routines
**prfstest.c** - ERTFS PRO FailSafe test suite

**SOURCE CODE FOR ERTFS DEVICE DRIVERS**

All of the provided device drivers are portable and should compile on any system. To make the device drivers work in a target environment you must edit the file **portio.c** and perhaps **portkern.c**. One exception to this is the floppy disk driver. Even though it is actually quite portable we chose to only support it for the x86 environment. If you wish to move it to another environment you must modify the driver source code itself.

You may disable any of these devices by editing the line in **portconf.h** that enables the device.

**drramdsk.c** - Portable ram disk driver
**drromdsk.c** - Portable rom disk driver
**drromdsk.h** - rom disk data generated by the MKROM function of the ERTFS Command Shell
**drfloppy.c** - Floppy disk driver
**drideata.c** - IDE/ATA, Compact flash driver
**drflash.h** - Flash Translation Layer Driver
**drflsftl.c** - Flash Translation Layer Driver
**drflsmtd.c** - Intel Flash Memory Technology Driver and RAM based Flash Emulator
**drmmccrd.c** - Multi-Media card driver
**drpcmram.c** - PCMCIA SRAM card driver
**drpcmcia.c** - PCMCIA support functions
**drpcmctl.c** - 82365 PCMCIA controller driver

## SECTION 3: CONFIGURING RUNTIME FEATURES

### CONFIGURING THE MEMORY USAGE AND CAPACITIES OF ERTFS

ERTFS is configured and its working memory is provided by the user through a user initialized parameter block. A helper function named **pc_ertfs_config()** is provided that the user may modify to change from the default configuration. The source code resides in apicnfig.c. See Appendix C to view example source code. It initializes the configuration block and provides ERTFS with the addresses of memory that it needs. **pc_ertfs_config()** is called immediately by ERTFS when it enters the initialization routine, **pc_ertfs_init()**.

This routine was designed to be modifiable by the user to change the default configuration. To simplify it, we define some configuration constants in this file that may be modified to change the configuration. These constants are only used locally to this file.

The default configuration is:

**ALLOC_FROM_HEAP**

Use malloc() to allocate, set this to 0 to use declared memory arrays instead. ***The default is 1.***

**NDRIVES**

The default maximum number of drives is set to 10 (J:) the highest possible value is 25 (Z:). ***The default is 10.***

*Note: A FAT buffer pool must be allocated for each drive. If NDRIVES is changed, you must add code to allocate or assign more FAT buffer pools. The default configuration allocates or assigns 10 fat buffer pools, If you increase NDRIVES, add code that duplicates the current code that initializes the {to be changed:: prtfs_cfg->fat_buffers[] array to support the added drives. If you decrease NDRIVES, eliminate the current code that initializes the prtfs_cfg->fat_buffers[] array entries for the drives that you are not using.} If you are not using ALLOC_FROM_HEAP, you should then also add or remove the static memory array declarations we use to reserve the memory at compile time.*

**NUM_USERS**

The maximum number of USER contexts. Set this to the number of tasks that will simultaneously use ERTFS. For polled mode set it to one. ***The default is 1.***

**NBLKBUFFS**

Number of blocks in the block buffer pool. ERTFS uses 536 bytes per block buffer pool entry. This setting impacts performance during directory traversals. It must be at least 4. More than 20 is rarely required. ***The default is 16.***

**BLKHASHSIZE**

Size of the block buffer hash table. This value must be a power of two, for example 2,4,8,16,32,64,128,256,512. If it is not a power of two the behavior is undefined. Increasing the hash table size increases the speed of block buffer pool accesses. Four bytes are required per hash table entry. While the block cache size can impact performance, it only has a significant impact on systems performing operations on volumes populated with large numbers of files and/or subdirectories. The hash table size and the number of buffers *do not* have to be the same. A theoretical increase in performance will be achieved by increasing the hash table size even if it is greater than the number of buffers. ***The default is 16***.

**Notes:**

1. *ERTFS-PRO provides an API call named* **pc_assign_buffer_pool()** *to assign a private buffer pool and hash table to an individual drive. If you decide to use this feature for all disk buffering, you may then wish to reduce* ***BLKHASHSIZE*** *and* ***NBLKBUFFS****. In this case some shared buffers are still needed because they are used as scratch buffers to perform certain operations, thus* ***NBLKBUFFS*** *should never be reduced below two.*
2. If the user wishes to use the ERTFS-PRO FailSafe API on a particular drive he must assign a private buffer pool to that drive.
3. Proper use of the ERTFS-PRO FailSafe API provides a good degree of time determinism when creating directory entries and extending or truncating files. When the API is used properly and enough ram resources are available the user can guarantee that the disk will not be accessed while these operations are performed. Only file data reads and writes will access the disk drive.

### CONFIGURING FAT BUFFER POOLS AND FAT HASH TABLES

The user determines, for each drive, the size of a hash table and the size of a buffer pool to be used for buffering FAT blocks. The choice of buffer pool size and hash table size significantly impacts performance. As a rule, increasing the size of these resources will increase theoretical maximum performance. Limitations to this rule are provided in the following paragraph.

There is no benefit to increasing either the fat buffer pool's size or the hash table's size beyond the size of the volume's on-

disk FAT. Thus, for 16 bit volumes there is no benefit to increasing the hash table size or the buffer pool size beyond 256. For 12 bit volumes there is no benefit to increasing the hash table size or the buffer pool size beyond 16.

If RAM memory conservation is a high priority in your design, note that on all but the slowest media a certain amount of FAT cache swapping is tolerable to human perception, and it is possible to assign relatively small FAT buffer pools to drives. As little as 1 or 2 blocks of FAT buffering will provide tolerable performance on a fast device.

FAT block access request patterns are governed mainly by how applications use the file system. Applications that access several files at one time and multitasking systems that access the file system from multiple threads will definitely see improved performance with added FAT buffering. Applications that sequentially read or write blocks to a single file will not see large overall performance increases, but with the ERTFS-PRO package, by increasing FAT buffer space, the user makes it possible to perform file system operations with deterministic behavior.

FAT buffers consume approximately 520 bytes each. Hash table entries consume 12 bytes per entry. The hash table size and the number of buffers *do not* have to be the same. A theoretical increase in performance could be achieved by increasing the hash table size (up to the size of the actual FAT) even if it is greater than the number of buffers.

*Note: If the user wishes to successfully use ERTFS-PRO's FailSafe mode on a particular drive, he must assign enough FAT buffer space to the drive to hold cached changes to the file allocation table. See the ERTFS-PRO's FailSafe API section for more information* ***(included with ERTFS Pro only)****.*

The hash table size must be a power of two, for example 2,4,8,16,32,64,128,256,512. If it is not a power of two, the behavior is undefined.

For convenience we define two sets of values that we use to configure the individual drives. **SMALL_FAT_SIZE** and **SMALL_FAT_HASHSIZE** determine the default size of the buffer pool and hash table for most drives. **LARGE_FAT_SIZE** and **LARGE_FAT_HASHSIZE** determine the default size of the buffer pool and hash table that we arbitrarily assign to drive C: (which, in the default configuration, we arbitrarily assign to a fixed IDE disk drive). These constants are used only in the file **apicnfig.c** to simplify the configuration process. The user is free to replace the use of these constants and individually configure each drive**.** *To do this you must review and modify the source code for* ***apicnfg.c*****. Apicnfig.c** is designed such that the user may edit the source code to precisely tune ERTFS.

**LARGE_FAT_SIZE**

The number of 520 byte blocks committed for buffering fat blocks on drive C: (see above for an explanation). ***The default is 32.***

**LARGE_FAT_HASHSIZE**

The number of 12 byte hash table entries committed for use on drive C: (see above for an explanation). ***The default is 32.***

**SMALL_FAT_SIZE**

The number of 520 byte blocks committed for buffering fat blocks on drives other than C: (see above for an explanation). ***The default is 32.***

**SMALL_FAT_HASHSIZE**

The number of 12 byte hash table entries committed for use on drives other than C: (see above for an explanation). ***The default is 32.***

**NUSERFILES**

The maximum number of files that may be open at one time. ***The default is 10.***

## SECTION 4: CONFIGURING COMPILE TIME FEATURE SET OPTIONS

There are a few compile time options that must be configured before building the ERTFS library. These options are included in the files **PORTCONF.H** and **RTFSCONF.H**. Constants in **RTFSCONF.H** control the inclusion of file system features. Constants in **PORTCONF.H** are needed to define your CPU configuration. A second section in **PORTCONF.H** controls your selection of active device drivers. ERTFS fully supports installable device drivers, these constants are here only as a convenience to control the inclusion and exclusion of EBS provided drivers. They also allow us to exclude unneeded functions from the porting layer when a service is not needed. See Appendix B for the **RTFSCONF.H** source code.

**CONFIGURATION CONSTANTS FROM RTFSCONF.H**

**Character set support**

ERTFS supports Unicode, JIS and standard 8 bit ASCII character sets. Select one character set from the following constants:

**INCLUDE_CS_JIS** - Set to 1 to API support JIS, Japanese Language ***(included with ERTFS Pro only)***

**INCLUDE_CS_ASCII** - Set to 1 to support 8 bit ASCII only ***(included with ERTFS Pro only)***

**INCLUDE_CS_UNICODE** - Set to 1 to support unicode characters. *Note: Unicode support requires VFAT*. ***(included with ERTFS Pro only)***

**File system support**

By default ERTFS support 12 and 16 bit File Allocation Table and short file names. Modify the following constants to enable long filename support and FAT32:

**VFAT** - Set to 1 to support long filenames ***(included with ERTFS Pro only)***

**FAT32** - Set to 1 to support 32 bit FAT ***(included with ERTFS Pro only)***

**Failsafe Operating Mode Support**

Failsafe mode provides a means to eliminate the risk of file system corruption that results from unexpected power interruptions and media removal events.

**INCLUDE_FAILSAFE_CODE** - Set to 1 to include failsafe support. ***(included with ERTFS Pro only)***

**CD-ROM Support.**

If the CD-ROM support package was purchased, the CDFS feature set may be enabled by editing **portconf.h**, setting **INCLUDE_CDROM** to 1. If **INCLUDE_CDROM** is enabled, **INCLUDE_IDE** must also be enabled.

**Extended DOS Partitions**

ERTFS contains code to interpret extended DOS partitions but since this feature is rarely used, it is provided as a compile time option. Modify the following constant to support extended DOS partitions:

**SUPPORT_EXTENDED_PARTITIONS** - Set to 1 to include support for extended DOS partitions. ERTFS contains code to support extended DOS partitions but they are rarely used and their use is not recommended.

**Setting file size limitations and file full policies**

You may set a limit on the maximum file size and determine the behavior when that limit is reached.

**RTFS_MAX_FILE_SIZE** - This is the maximum file size in bytes that ERTFS will extend a file to when it is executing po_write(), po_chize() or po_extendfile(). The default value is 0xffffffff. You may change this value to limit file sizes. For example to limit the maximum file size to 2 Gigabytes set RTFS_MAX_FILE_SIZE to 0x80000000. If a file reaches this length the operation is either truncated or an error is returned, depending on which function is being exectutes and the values of RTFS_ TRUNCATE_WRITE_TO_MAX.

**RTFS_TRUNCATE_WRITE_TO_MAX** - If this is set to 1 po_write() calls are truncated write up to RTFS_MAX_FILE_ SIZE bytes. If that value is exceeded the write count is truncated and the short write count is returned.

If this is set to 0, if po_write() attempts to write beyond RTFS_MAX_FILE_SIZE, errno is set to PETOOLARGE and po_write() returns -1.

Regardless of this setting po_chsize() and po_extend_file() will always fail and set errno to PETOOLARGE if an attempt is made to extend a file beyond RTFS_MAX_FILE_SIZE. .

## SECTION 5: CONFIGURING AT COMPILE TIME THE PROCESSOR CONFIGURATION

### CONFIGURATION CONSTANTS FROM PORTCONF.H

Several CPU configuration constants are provided in **PORTCONF.H**. These values must be checked and modified for your configuration. Please refer to the Section 9 for a thorough explanation of how to modify the preprocessor constants for your CPU.

## SECTION 6: CONFIGURING INCLUSION OF DEVICE DRIVER SUPPORT

### DEVICE SELECTION CONSTANTS FROM PORTCONF.H

Compile time constants are used to control whether we compile a device driver and whether we install the driver at startup and, in some cases, whether we require porting layer code that is required for that device driver to operate. PORTCONF.H contains a set of defines that control this for ERTFS provided device drivers.

*Note: ERTFS fully supports run time binding of device drivers. These constants are used to make the process more convenient to the user. If you create your own driver, you are not required to use this same technique, as long as you attach the driver inside the **pc_ertfs_init()** routine.*

The following constants are defined. Set the constant to 1 to include the device driver, set it to 0 to exclude it. After you have selected a device, consult the porting section of the reference guide for any special instruction on supporting the device.

| | | |
|---|---|---|
| **INCLUDE_IDE** | - | Include the IDE driver |
| **INCLUDE_PCMCIA** | - | Include the PCMCIA driver |
| **INCLUDE_PCMCIA_SRAM** | - | Include the PCMCIA static ram card driver |
| **INCLUDE_COMPACT_FLASH** | - | Support compact flash (requires IDE and PCMCIA) |
| **INCLUDE_FLASH_FTL** | - | Include the linear flash driver |
| **INCLUDE_ROMDISK** | - | Include the rom disk driver |
| **INCLUDE_RAMDISK** | - | Include the RAM disk driver |
| **INCLUDE_FLOPPY** | - | Include the floppy disk driver |
| **INCLUDE_HOSTDISK** | - | Include the Windows disk simulator |
| **INCLUDE_UDMA** | - | Include ultra dma support for the ide driver |
| **INCLUDE_82365_PCMCTRL** | - | Include the 82365 PCMCIA controller driver |

*Note: The default configuration is to enable a ram-disk and a rom-disk driver only.*

These devices require no additional support from the porting layer.

## SECTION 7: INTRODUCTION TO PORTING ERTFS

The ERTFS porting layer has been greatly simplified from previous versions. There are three files that must be modified to support a new target: **portconf.h**, **portkern.c** and **portio.c**. A fourth file, **portmain.c**, must be ported if the RTFS test programs will be run in a standalone environment and a main program entry point must be provided.

## SECTION 8: PORTING BY BORROWING FROM EXISTING TARGETS

The ERTFS install directory contains a subdirectory named ‘targets.’ This directory contains several subdirectories containing sample solutions of the porting layer for several target real time operating systems and microprocessor targets. To build ERTFS for a selected target you must use the copies of **portkern.c**, **portconf.h** and **portio.c** from the appropriate **(.krn**) and **(.io)** subdirectories. The simplest way to do this is to copy these files over top of the generic files of the same name in the install directory. If you are creating your own port of ERTFS, we suggest that you can start with either a reference port that is similar to your own or that you start with the completely generic files provided in the template.krn and template.io subdirectories.

The notes in the following sections describe the porting issues for each file.

Please read them before porting to your environment.

## SECTION 9: PORTING ISSUES FOR THE FILE PORTCONF.H.

(See Appendix A for portconf.h source code.)

**PORTCONF.H** contains a few simple definitions that are related to porting. They must be modified to match your processor architecture.

**KS_LITTLE_ENDIAN** - Set this to 1 for little endian byte architectures such as Intel. Set it to 0 for big endian architectures such as Motorola.

**KS_RTFS_LITTLE_ODD_PTR_OK** - This value is not relevant if **KS_LITTLE_ENDIAN** is 0. If **KS_LITTLE_ENDIAN** is 1 then this value should be set to 1 if the CPU is forgiving about accessing words on odd boundaries. A good rule of thumb is to set this to 1 on x86 based little endian systems and to zero on non x86 little endian systems.

**KS_CONSTANT** - Set this to your compiler's const declaration. A typical setup is:

```
#define KS_CONSTANT const /* If this presents problems use the NULL alternative
#define KS_CONSTANT
```

**KS_FAR** - This value is only relevant in segmented architectures such as real mode Intel 8086. The **KS_FAR** modifier is used when declaring data arrays in a few places so the data is put in FAR memory rather than near memory. For these architectures use the following definition:

```
#define KS_FAR far
```

Otherwise leave the definition as it is:

```
#define KS_FAR
```

**INCLUDE_IDE** - Set this to zero if you will not be using the IDE device driver. Set it to 1 if you will use the IDE device driver. When this constant is set to zero, the IDE device driver compiles to nothing and routines required by the IDE device driver in **portkern.c** and **portio.c** are excluded.

**INCLUDE_PCMCIA** - Set this to zero if you will not be using the PCMCIA compact flash interface or the PCMCIA SRAM device driver. Set it to 1 if you will use one of these devices.

**INCLUDE_COMPACT_FLASH** - Set this to one if you will be using the PCMCIA compact flash interface to the IDE device driver. Set it to 0 if not. *Note: If you enable INCLUDE_COMPACT_FLASH you must also manually enable INCLUDE_PCMCIA and INCLUDE_IDE.*

**INCLUDE_FLOPPY** - Set this to zero if you will not be using the floppy disk device driver. Set it to 1 if you will use the floppy device driver. When this constant is set to zero, the floppy device driver compiles to nothing and routines required by the floppy disk device driver in **portkern.c** and **portio.c** are excluded.

**INCLUDE_82365_PCMCTRL** - Set this to zero if you will not be using the 82365 PCMCIA controller chip driver in drpcmctl.c. Set it to 1 if you will use the pcmctrl device driver. When this constant is set to zero, the floppy device driver compiles to nothing and routines required by the device driver in **portkern.c** and **portio.c** are excluded.

**INCLUDE_UDMA** - Set this to one if you will be using the ultra-dma mode of the IDE device driver. If set to one, you must implement the following functions. **rtfs_port_ide_bus_master_address**, **ide_rd_udma_status**, **ide_wr_udma_status**, **ide_rd_udma_command**, **Ide_wr_udma_command**, and **ide_wr_udma_address**. The requirements are explained in Section 24.

**INCLUDE_PCMCIA_SRAM** - Set to 1 to include the pcmcia static RAM card driver. *Note **INCLUDE_PCMCIA** must also be enabled.*

**INCLUDE_FLASH_FTL** - Set to 1 to include the linear flash driver (The FTL device driver is being revised)

**INCLUDE_ROMDISK** - Set to 1 to include the ROM disk driver

**INCLUDE_RAMDISK** - Set to 1 to include the RAM disk driver

**INCLUDE_HOSTDISK** - Set to 1 to include the windows host disk, disk simulator

## SECTION 10: PORTING: OVERVIEW

**portkern.c** contains ERTFS's interface to the RTOS and to some elements of the underlying hardware. See Appendix E for example source code for portkern.c. The RTOS resources required by ERTFS include mutex semaphores, interrupt event signals for selected device drivers and some simple timer management routines. This section describes ERTFS requirements from the RTOS and discusses porting requirements. Device access functions and interrupt management requirements are discussed in another section.

## SECTION 11: PORTING: PERIODIC CLOCK SUPPORT

ERTFS requires a periodic clock for measuring device watchdog timeouts. Also if ERTFS is running in POLLED mode, it needs a clock source timed signal tests. The user must implement the clock tick function if using any of the non-ram or rom media based device drivers.

This routine takes no arguments and returns an unsigned long. The routine must return a tick count from the system clock. The macro named MILLISECONDS_PER_TICK (also local to **portkern.c**) must be defined in such a way that it returns the rate at which the tick increases in milliseconds per tick.

This routine is declared as static to emphasize that its use is local to the **portkern.c** file only.

**static dword rtfs_port_get_ticks(void)**

## SECTION 12: PORTING: MUTEX SEMAPHORE SUPPORT

*Note: ERTFS does not require mutex semaphore in non-multitasking environments. If you are porting to a non-multitasking environment, or if you can guarantee that only one task at a time will use ERTFS, then please skip this section.*

ERTFS requires several mutex (binary) semaphores. One mutex semaphore is required per drive to be used and one system wide critical section mutex semaphore is required. There are a maximum of 26 possible drives but the actual number to support is user configurable as described in Section 3.

In addition some device drivers may require their own mutex semaphores to function properly. The only stock driver that requires a mutex is the ATA/IDE driver. If you are using ATA/IDE please plan for one extra mutex.

The Mutex code is abstracted into three functions that must be modified by the user to support the target RTOS. The required functions are:

1. rtfs_port_alloc_mutex()
2. rtfs_port_claim_mutex()
3. rtfs_port_release_mutex()

The requirements for each of these functions is provided here.

**unsigned long rtfs_port_alloc_mutex(void)** -

This routine takes no arguments and returns an unsigned long. The routine must allocate and initialize a Mutex, setting it to the “not owned” state. It must return an unsigned long value that will be used as a handle. ERTFS will not interpret the value of the return value. The handle will only be used as an argument to the **rtfs_port_claim_mutex()** and **rtfs_port_release_mutex()** calls. The handle may be used as an index into a table or it may be cast internally to an RTOS specific pointer. If the mutex allocation function fails, this routine must return 0 and the ERTFS calling function will return failure.

**void rtfs_port_claim_mutex(unsigned long handle)** -

This routine takes as an argument a mutex handle that was returned by **rtfs_port_alloc_mutex()**. If the mutex is already claimed, it must wait for it to be released and then claim the mutex and return.

**void rtfs_port_release_mutex(unsigned long handle)** -

This routine takes as an argument a mutex handle that was returned by **rtfs_port_alloc_mutex()** that was previously claimed by a call to **rtfs_port_claim_mutex()**. It must release the handle and cause a caller blocked in **rtfs_port_claim_mutex()** for that same handle to unblock.

## SECTION 13: PORTING: EVENT SIGNALING SUPPORT

A set of signaling functions is required to support interrupt driven device drivers. The porting layer provides a set of signal management functions that the user must populate if interrupt driven device drivers are to be used. The signals are always allocated, tested and cleared from the within a task context. They are always signaled from the interrupt context.

*Note: Currently ERTFS only requires these functions if the floppy disk is used or if the IDE driver is used in interrupt mode rather than polled mode. Additional user supplied device drivers may opt to use these calls or they may implement their own native signaling method at the programmer's discretion.*

If you are not using the floppy disk driver or the IDE driver in interrupt mode you may skip this section.

The signaling code is abstracted into four functions that must be modified by the user to support the target RTOS. The required functions are: **rtfs_port_alloc_signal()**, **rtfs_port_clear_signal()**, **rtfs_port_test_signal()**, and **rtfs_port_set_signal()**. The requirements for each of these functions are provided here.

*Note: In a NON-RTOS environment the implementation of these functions do not need user modification as long as the routine **rtfs_port_get_ticks()** has been implemented. In an RTOS environment these routines MUST be implemented.*

**unsigned long rtfs_port_alloc_signal(void)**

This routine takes no arguments and returns an unsigned long. The routine must allocate and initialize a signaling device (typically a counting semaphore) and set it to the "not signaled" state. It must return an unsigned long value that will be used as a handle. ERTFS will not interpret the value of the return value. The handle will only used as an argument to the **rtfs_port_clear_signal()**, **rtfs_port_test_signal()**, and **rtfs_port_set_signal()** calls.

**void rtfs_port_clear_signal(unsigned long handle)**

This routine takes as an argument a handle that was returned by **rtfs_port_alloc_signal()**. It must place the signal in an unsignalled state such that a subsequent call to **rtfs_port_test_signal()** will not return success until **rtfs_port_set_signal()** has been called. This clear function is necessary since it is possible, although unlikely, that an interrupt service routine could call **rtfs_port_set_signal()** after the intended call to **rtfs_port_test_signal()** timed out. A typical implementation of this function for a counting semaphore is to set the count value to zero or to poll it until it returns failure.

**int rtfs_port_test_signal(unsigned long handle, int timeout)**

This routine takes as an argument a handle that was returned by **rtfs_port_alloc_signal()** and a timeout value in milliseconds. It must block until timeout milliseconds have elapsed or **rtfs_port_set_signal()** has been called. If the test succeeds it must return 0, if it times out it must return a non-zero value.

**void rtfs_port_set_signal(unsigned long handle)**

This routine takes as an argument a handle that was returned by **rtfs_port_alloc_signal()**. It must set the signal such that a subsequent call to **rtfs_port_test_signal()** or a call currently blocked in **rtfs_port_test_signal()** will return success.

*Note: **rtfs_port_set_signal()** is always called from the device driver interrupt service routine while the processor is executing in the interrupt context.*

## SECTION 14: PORTING: TIMER SUPPORT FUNCTIONS

Three abstract timer support functions are included. These routines are only used by the floppy disk driver, the IDE device driver and the PCMCIA support subsystem. If none of these devices is being used then please skip this section.

*Note: In a NON-RTOS environment the implementation of these functions do not need user modification as long as the routine rtfs_port_get_ticks() has been implemented. In an RTOS environment these routines MUST be implemented.*

The requirements for each of these functions is defined below.

**void rtfs_port_sleep(int sleeptime)**

This routine takes as an argument the time to sleep milliseconds. It must not return to the caller until at least sleeptime milliseconds have elapsed. In a multitasking environment this call should yield the cpu.

**unsigned long rtfs_port_elapsed_zero()**

This routine takes no arguments and returns an unsigned long. The routine must return an unsigned long value that will later be passed to **rtfs_port_elapsed_check()** to test if a given number of milliseconds or more have elapsed. A typical implementation of this routine will read the system tick counter and return it as an unsigned long. ERTFS makes no assumptions about the value that is returned. *Note: In either a POLLED or RTOS environment this does not need modification as long the routine **rtfs_port_get_ticks()** has been implemented.*

**int rtfs_port_elapsed_check(unsigned long zero_val, int timeout)**

This routine takes as arguments an unsigned long value that was returned by a previous call to **rtfs_port_elapsed_zero()** and a timeout value in milliseconds. If “timeout” milliseconds have not elapsed it should return 0. If “timeout” milliseconds have elapsed it should return 1. A typical implementation of this routine would read the system tick counter, subtract the zero value, scale the difference to milliseconds and compare that to timeout. If the scaled difference is greater or equal to timeout it should return 1, if less than timeout it should return 0. *Note: In either a POLLED or RTOS environment this does need modification as long the routine **rtfs_port_get_ticks()** has been implemented.*

## SECTION 15: PORTING: IDENTIFY CURRENT TASK ID

ERTFS supports per task contexts for storing the task's current working directory context and the last errno value produced by ERTFS on behalf of that task. In order to do this each executing task running ERTFS must have a unique task identifier. The user is required to populate the function **rtfs_port_get_taskid()** to provide this service. The function is described below.

*Note: Before a user task exits, if it has used the ERTFS API, it must call **pc_free_user()**. Failure to do this will cause ERTFS to run out of user context blocks.*

*Note: This function requires no modification to run in polled mode. If you do not implement this function in an RTOS environment you will lose the ability for each task to have it's own current working directory environment. All tasks will share the same working directory. In this case DO NOT call pc_free_user() before tasks exit.*

**unsigned long rtfs_port_get_taskid()**

This function must return an unsigned long number that is unique to the currently executing task such that each time this function is called from the same task it returns this same unique number. A typical implementation of this function would get address of the current task control block, cast it, and return it.

*Note: **NUM_USERS** must be large enough to support the number of tasks. See Section 3.*

## SECTION 16: PORTING: CONSOLE INPUT AND OUTPUT

Two console-I/O functions are used by ERTFS to display messages and retrieve user input. These routines are not essential for the correct operation of ERTFS but they are useful for printing diagnostics. They are essential if you wish to use the interactive command shell.

*Note: If you don't wish to display diagnostic messages or use the command shell, skip this section.*

**void rtfs_port_tm_gets(byte *buffer)**

This routine requires porting if you wish to use the interactive test shell. It must provide a line of input from the terminal driver. The line must be NULL terminated and it must not contain ending newline or carriage return characters.

**void rtfs_port_puts(byte *buffer)**

This routine requires porting if you wish to display messages to your console. This routine must print a line of text from the supplied buffer to the console. The output routine must not issue a carriage or linefeed command unless the text is terminated with the appropriate control character (\n or \r).

## SECTION 17: PORTING: INTERRUPT ENABLE/DISABLE

In a few places ERTFS must disable interrupts, execute some lines of code and then re-enable interrupts. Two functions in the porting file must be populated to add this support. The functions are **rtfs_port_disable()** and **rtfs_port_enable()**. The functions are described in the following section.

*Note: Currently ERTFS only requires these functions if either the floppy disk, the 82365 PCMCIA controller or the flash chip memory technology driver is being used. If you are not using any of these devices please skip this section.*

**void rtfs_port_disable(void)**

This function must disable interrupts and return. An example implementation of this function for Intel X86 is:

```
__asm cli
```

**void rtfs_port_enable(void)**

This function must re-enable interrupts that were disabled via a call to **rtfs_port_disable()**. An example implementation of this function for Intel X86 is.

```
__asm sti
```

## SECTION 18: PORTING: MISCELLANEOUS FUNCTIONS

**void rtfs_port_exit(void)**

*Note: Most embedded systems never exit. If your application never exits, skip this section.*

This function must exit the RTOS session and return to the user prompt. It is only necessary when an RTOS is running inside a shell environment like Windows. It is not necessary to implement this function if you are using the ERTFS command library in an environment that will ever exit, i.e., if ERTFS is running with an RTOS in a Dos/Windows or Unix environment you should put a call to your RTOS exit code.

## SECTION 19: PORTING: CLOCK CALENDAR

**DATESTR *pc_getsysdate(DATESTR * pd)**

When ERTFS needs to date stamp a file, it calls this routine to get the current time and date. The sample routine returns a fixed data and time. If your target has a clock calendar function, you may modify this routine to provide real date and time values.

The source for this routine is in file **portkern.c** (see Appendix F) and is self-explanatory. If you wish to integrate a calendar clock module please see the comments in the source code.

**Device driver specific hardware and system access functions**

**Portio.c** contains ERTFS's interface underlying hardware. Routines in this file must be modified to adapt the portable device drivers to new hardware. Most of the required functions are simple I/O port access routines. Please read the following sections to examine the details. See Appendix F for the **portio.c** source code.

## SECTION 20: PORTING: FLOPPY DISK

*Note: These routines are provided to support the floppy disk device driver in drfloppy.c. If your application will not be using a floppy disk device please ignore this section.*

The floppy driver is for NEC 756 class controllers and is primarily for PC architectures but it can be made to work in non-PC environments. Six routines are listed here as routines that may need modification. In a standard PC environment the only routine that will need changing is the hook interrupt routine, this routine will need modification to support your RTOS interrupt hook and dispatch method.

**void hook_floppy_interrupt(int irq)**

This routine is called by the floppy disk device driver. It must establish an interrupt handler such that the plain 'C' function **void floppy_isr(void)** is called when the floppy disk interrupt occurs. The value in "irq" is always 6. This is the PC's standard mapping of the floppy interrupt. If this is not correct for your system, just ignore the irq argument.

**get_floppy_type(int driveno)**

The source for this routine is in file drfloppy.c. It is hardwired to return **DT_144** (1.44MB 3.5 inch floppy disk), by far the most common floppy disk drive type in use. The source code is self-explanatory and describes what values to return for other drive types.

**BOOLEAN fl_dma_init()**

This routine must set up a DMA transfer for the floppy device driver. The source for this routine is in file drfloppy.c. It is hardwired to start the appropriate dma transfer for DMA channel 2 on a standard PC AT architecture. It must be analyzed and modified if being used on some other system.

**Floppy disk register-access routines**. Six functions, all contained in **drfloppy.c**, are provided to access the registers of the NEC765 class floppy disk controller. They are all hard wired to use Intel I/O out and I/O in instructions in the address range 0x3f0 to 0x3ff. If the floppy driver is being moved to a non PC environment, these routines must be modified.

- **fl_read_data** - Reads a byte from the 765 data register
- **fl_read_drr** - Reads a byte from the 765 data rate register
- **fl_read_msr** - Reads a byte from the 765 master status register
- **fl_write_data** - Writes a byte to the 765 data register
- **fl_write_dor** - Writes a byte to the 765 digital output register
- **fl_write_drr** - Writes a byte to the 765 data rate register

## SECTION 21: PORTING: PCMCIA

ERTFS is supports Compact Flash and PCMCIA Ram cards. If these devices are to be used then either the PCMCIA controller must be managed externally or the ERTFS PCMCIA subsystem must be used. The subsystem consists of the files **drpcmcia.c**, which is portable and **drpcmctl.c,** a device driver for INTEL 82365 compatible pcmcia controller chips.

Several functions are required for the portable pcmcia code in **drpcmcia.c** to operate. They are provided by ERTFS in the file **drpcmctl.c** for the 82365 controller.

### SECTION 21.1: PORTING PCMCIA TO A NON-82365 PCMCIA CONTROLLER

*Note: If you have another type of pcmcia controller, you must provide these five functions and possibly the management interrupt handler we will discuss in this section. If you do have an 82365 based controller or you are not using PCMCIA, you can skip this section.*

**void pcmctrl_put_cis_byte(int socket, dword offset, byte c)**

This function must store the byte c to the location that is offset bytes into the CIS region of the card at slot number 0 or 1.

**byte pcmctrl_get_cis_byte(int socket, dword offset)**

This function must read and return the byte stored offset bytes into the CIS region of the card at slot number 0 or 1.

**void pcmctrl_map_ata_regs(int socket, dword ioaddr, int interrupt_number)**

This function must configure the pcmcia controller such that the IDE driver can access the ATA register in the Compact Flash card. It must also map in the PMCIA interrupt line so that it will generate the I/O completion interrupt and it must apply VCC power to the card.

Notes:

1. The value **ioaddr** is the address that was passed into the IDE device driver. This value is assigned by the user inside **pc_ertfs_init()** through the variable pdr->register_file_address. This is the range of addresses that the ide register access functions will address. (see below).
2. The value **interrupt_number** is the interrupt number that will be used by hook_ide_interrupt()to field I/O completion interrupts. This value is assigned by the User inside pc_ertfs_init() through the variable pdr->interrupt_number. If this value is set to -1, the user is requesting polled interaction with the device and no system level interrupt handling is required.

**BOOLEAN pcmctrl_card_installed(int pcmcia_slot_number)**

This routine must return TRUE if a card is installed in the slot, FALSE if it is not.

**BOOLEAN pcmctrl_card_changed(int pcmcia_socket_number)**

This routine must return TRUE if a media change event has occurred on the card.

*Note: If the pcmcia controller supports card removal interrupts then this routine is not needed and may simply return FALSE always. If card removal interrupts are not supported then this routine must be implemented if hot swapping is needed.*

If this routine is required, it must check the pcmcia interface at the logical socket for the presence of a latched media change condition. If a change has occurred then it must clear the latched condition and return TRUE. Otherwise, it must return FALSE.

If a change is detected the routine also unmaps the card in pcmcia space and shuts down power to the card. This is done to assure that the card powers up appropriately when it is reopened.

**Inputs:** pcmcia slot number (0,1)

**Returns:**

TRUE - a card has been inserted or removed since last called

FALSE - no card has been inserted or removed since last called

**byte *pd67xx_map_sram(int socket_no , dword offset)**

This routine is required only if the pcmcia SRAM card driver is being used. It must return a pointer to a block in the pcmcia sram's memory space that is offset bytes from the beginning of the SRAM memory area and make sure that 512 bytes are readable and writable at that location.

*Note: The SRAM driver actually passes in the byte offset of the block divided by two plus 256. This is because pcmcia SRAM cards are mapped into CIS space which has mod two addressability and starts at location 512 in CIS space. You may modify this algorithm in pcmsram_block() if needed.*

### SECTION 21.2: IMPLEMENTING CARD EVENT HANDLERS FOR EXTERNALLY SUPPLIED PCMCIA CONTROLLER

**DRIVERS**

If the user wishes to support hot swapping of compact flash cards, a management interrupt service or some other mechanism to announce card removal events to the IDE device driver must be provided. The 82365 driver (drpcmctl.c) contains a routine called mgmt_isr() that should be used as a model for how to approach this. The source code is provided below.

Please note the following about how the routine works and how it must behave. The routine detects a card change, if the change was a removal event(no card in slot) then it must turn off VCC power to the card and then look up and call the IDE device driver's device I/O control function with the appropriate arguments to report a card removal event.

Here is the source code for the model event handler.

```
void mgmt_isr(void)
{
int i;
byte card_status;

    for (i = 0; i < NSOCKET; i++)
    {
        card_status = pd67xx_read(i, 0x04);
        /* Write the status register to clear */
        pd67xx_write(i, 0x04, 0xff);
        if (card_status)
        {
            if (!pcmctrl_card_is_installed(i, FALSE))
                    {
                pcmctrl_card_down(i);
                {
                int j;
                DDRIVE *pdr;

                                                for (j = 0; j < prtfs_cfg->cfg_NDRIVES; j++)
                                            {

                                                                pdr = pc_drno_to_drive_struct(j);
                        if (pdr &&
                            pdr->drive_flags & DRIVE_FLAGS_PCMCIA &&
                            pdr->drive_flags & DRIVE_FLAGS_VALID  &&
                            pdr->pcmcia_slot_number == i &&
                            pdr->pcmcia_controller_number == 0)
                               pdr->dev_table_perform_device_ioctl(
                                   pdr->driveno,
                                   DEVCTL_REPORT_REMOVE, (PFVOID) 0);
                    }
                }
            }
        }
    }
}
```

## SECTION 22: PORTING: 82365 PCMCIA CONTROLLER

*Note: Several routines are provided in the porting layer to support the 82365 pcmcia controller device driver in the file drpcmctl.c. If you are not using PCMCIA or are not using and 82365 class controller, you can skip this section.*

**void hook_82365_pcmcia_interrupt(int irq)**

This routine must establish an interrupt handler that will call the plain 'C' routine **void mgmt_isr(void)** when the chip's management interrupt event occurs. The value of the argument 'irq' is the interrupt number that was put into the 82365's management-interrupt selection register and is between 0 and 15. This is controlled by the constant "MGMT_INTERRUPT.

**phys82365_to_virtual(PFBYTE * virt, unsigned long phys)**

This routine must take a physical linear 32 bit bus address passed in the "phys" argument and convert it to an address that is addressable in the logical space of the CPU, returning that value in "virt." Two sample methods are provided; a flat version where the virtual address is simply the "phys" address cast to a pointer; and a second segmented version for X86 segmented applications.

*Note: This routine must be changed if you are using and 82365 class PCMCIA controller chip in an environment that is not either 16 bit segmented Intel or a flat 32 bit environment where the virtual address is the same as the physical address.*

**void write_82365_index_register(byte value)**

**void write_82365_data_register(byte value)**

**byte read_82365_data_register()**

*Note: These routines must be modified if you are using an 82365 class PCMCIA controller in a non PC environment.*

These routines write and read the 82365 index and data registers, which, in a standard PC environment, are located in I/O space at address 0x3E0 and 0x3E1. Non PC architectures typically map these as memory mapped locations somewhere high in memory such as 0xB10003E0 and 0xB10003E1.

## SECTION 23: PORTING: IDE AND COMPACT FLASH

This section describes actions that must be taken by the user to support the ERTFS ATA/ATAPI device driver that is implemented in the file drideata.c.

*Note: If you are not using ATA, ATAPI or compact flash based devices in your application, skip this section.*

To support this device driver, the user must populate several register access functions and provide an interrupt service layer if he wishes to run the device in interrupt driven mode versus polled mode.

**void hook_ide_interrupt(int irq, int controller_number)**

The user must provide this function if he desires to use the IDE driver in interrupt mode. If you don't wish to use the ATA driver in interrupt mode, make sure that the value of pdr->interrupt_number is set to -1 inside **pc_ertfs_init()** for the ide device, and skip this section since **ide_hook_interrupt()** will then not be called.

**hook_ide_interrupt()** is called with the interrupt number in the argument **irq** taken from the user's setting of pdr->interrupt_number in **pc_ertfs_init()**. Controller number is taken from the pdr->controller_number field as set in **pc_ertfs_init()** by the user. **hook_ide_interrupt()** must establish an interrupt handler such that the plain 'C' function "**void ide_isr(int controller_number)**" is called when the IDE interrupt occurs. The argument to **ide_isr()** must be the controller number that was passed to **hook_ide_interrupt()**, this value is typically zero for single controller system.

*Note: For all ide register access functions, **register_file_address** is the value that was provided by the user in **pc_ertfs_init()** in the variable pdr->register_file_address.*

**REGISTER ACCESS FUNCTIONS REQUIRED BY THE IDE DRIVER:**

**byte ide_rd_status(dword register_file_address)**

This function must return the byte in location 7 (IDE_OFF_STATUS) of the ide register file.

*Note: For all ide register access functions, register_file_address is the value that was provided by the user in pc_ertfs_init() in the variable pdr->register_file_address.*

**word ide_rd_data(dword register_file_address)**

This function must return the word in location 0 (IDE_OFF_DATA) of the ide register file at register_file_address.

**byte ide_rd_sector_count(dword register_file_address)**

This function must return the byte in location 2 (IDE_OFF_SECTOR_COUNT) of the ide register file.

**byte ide_rd_alt_status(dword register_file_address, int contiguous_io_mode)**

This function must return the byte in location 0x206 (IDE_OFF_ALT_STATUS) of the ide register file at register_file_address. If the value of the argument contiguous_io_mode is 1, then the register must be 14 rather than 0x206.

**byte ide_rd_error(dword register_file_address)**

This function must return the byte in location 1 (IDE_OFF_ERROR)of the ide register file.

**byte ide_rd_sector_number(dword register_file_address)**

This function must return the byte in location 3 (IDE_OFF_SECTOR_NUMBER) of the ide register file.

**byte ide_rd_cyl_low(dword register_file_address)**

This function must return the byte in location 4 (IDE_OFF_CYL_LOW) of the ide register file.

**byte ide_rd_cyl_high(dword register_file_address)**

This function must return the byte in location 5 (IDE_OFF_CYL_HIGH) of the ide register file.

**byte ide_rd_drive_head(dword register_file_address)**

This function must return the byte in location 6 (IDE_OFF_DRIVE_HEAD) of the ide register file.

**byte ide_rd_drive_address(dword register_file_address,int contiguous_io_mode)**

This function must return the byte in location 0x207 (IDE_OFF_DRIVE_ADDRESS) of the ide register file at register_file_address. If the value of the argument contiguous_io_mode is 1 then the register must be 15 rather than 0x207.

**void ide_wr_dig_out(dword register_file_address, int contiguous_io_mode, byte value)**

This function must place the byte in value at location 0x206 (IDE_OFF_ALT_STATUS) of the ide register file at register_file_address. If the value of the argument contiguous_io_mode is 1, then the register must be 14 rather than 0x206.

**void ide_wr_data(dword register_file_address, word value)**

This function must place the word in location 0 (IDE_OFF_DATA) of the ide register file at register_file_address */

**void ide_wr_sector_count(dword register_file_address, byte value)**

This function must place the byte in value at location 2 (IDE_OFF_SECTOR_COUNT) of the ide register file.

**void ide_wr_sector_number(dword register_file_address, byte value)**

This function must place the byte in value at location 3 (IDE_OFF_SECTOR_NUMBER) of the ide register file.

**void ide_wr_cyl_low(dword register_file_address, byte value)**

This function must place the byte in value at location 4 (IDE_OFF_CYL_LOW) of the ide register file.

**void ide_wr_cyl_high(dword register_file_address, byte value)**

This function must place the byte in value at location 5 (IDE_OFF_CYL_HIGH) of the ide register file. register_file_address is the value that was provided by the user in pc_ertfs_init() in the variable pdr->register_file_address.

**void ide_wr_drive_head(dword register_file_address, byte value)**

This function must place the byte in value at location 6 (IDE_OFF_DRIVE_HEAD) of the ide register file.

**void ide_wr_command(dword register_file_address, byte value)**

This function must place the byte in value at location 0 (IDE_OFF_DATA) of the ide register file.

**void ide_wr_feature(dword register_file_address, byte value)**

This function must place the byte in value at location 1 (IDE_OFF_FEATURE) of the ide register file.

**void ide_insw(register_file_address, unsigned short *p, int nwords)**

This function must read nwords 16 bit values from the data register at offset 0 of the ide register file and place them in successive memory locations starting at p. Since large blocks of data are transferred from the drive in this way, this routine should be optimized. On x86 based systems the rep insw instruction should be used, on non x86 platforms the loop should be as tight as possible.

**void ide_outsw(register_file_address, unsigned short *p, int nwords)**

This function must write nwords 16 bit values to the data register at offset 0 of the ide register file. The data is taken from successive memory locations starting at p. Since large blocks of data are transferred from the drive in this way this routine should be optimized. On x86 based systems the rep outsw instruction should be used, on non x86 platforms the loop should be as tight as possible.

## SECTION 24: PORTING: IDE ULTRA DMA MODE

This section describes actions that must be taken by the user to support the ERTFS ATA/ATAPI device driver in ultra dma mode as implemented in the file **drideata.c**. These support functions reside in portio.c and must be modified for your target if you wish to use Ultra-dma.

**dword rtfs_port_ide_bus_master_address(int controller_number)**

This function must determine if the specified controller is a PCI bus-mastering IDE controller and if so return the location of the control and status region for that controller. If it is not a bus-mastering controller or ultra dma mode isn't supported it must return zero. This will tell the IDE device driver to use PIO mode.

**byte ide_rd_udma_status(dword bus_master_address)**

This function must read the status byte value at location 2 of the bus master control region.

**void ide_wr_udma_status(dword bus_master_address, byte value)**

This function must write the byte value to location 2 of the bus master control region.

**byte ide_rd_udma_command(dword bus_master_address)**

This function must read the command byte value at location 0 of the bus master control region.

**void ide_wr_udma_command(dword bus_master_address, byte value)**

This function must write the byte value to location 0 of the bus master control region.

**void ide_wr_udma_address(dword bus_master_address, dword bus_address)**

This function must write the dword to location 4 of the bus master control region.

**unsigned long rtfs_port_bus_address(void * p)**

This function must take a logical pointer and convert it to an unsigned long representation of its address on the system bus.

## SECTION 25: SUPPORTING REMOVABLE TRUE-IDE DEVICES

This section describes what is required to support hot swapping of removable TRUE IDE devices.

Hot swapping may be supported if either a card removal interrupt can be generated by the controller or if the controller provides a latched mediachange event. Three routines related to TRUE IDE hot-swapping are provided: **trueide_card_changed**, **trueide_card_installed** and **trueide_report_card_removed**. The requirements for each of these routines is provided in this section.

**BOOLEAN trueide_card_changed(DDRIVE *pdr)**

Modify this routine to support removable TRUEIDE devices.

This routine may be used to provide support for hot swapping of removable TRUEIDE devices in cases where a removal interrupt source is not available.

The TRUEIDE circuit must provide a latch that detects a card removal. This routine must report the value of the latch, TRUE if the media has changed, FALSE if it has not. It must clear the latch before it returns.

By default **trueide_card_changed()** returns FALSE, emulating a fixed disk or a removable disk with no media changed latch.

Note: The DRIVE_FLAGS_REMOVABLE flag must be set in apiinit.c for removable media. To do this, OR in DRIVE_FLAGS_REMOVABLE to the drive flags field.

pdr->drive_flags |= DRIVE_FLAGS_REMOVABLE;

**Example:**

```
BOOLEAN trueide_card_changed(DDRIVE *pdr)
{
if (read_ide_change_latch() == 1)
{
clear_ide_change_latch();
return(TRUE);
}
else
return(FALSE);
}
```

**BOOLEAN trueide_card_installed(DDRIVE *pdr)**

Modify this routine to support removable trueide devices.

**trueide_card_installed()** must return TRUE if IDE compatible media is installed, FALSE if it is not.

By default **trueide_card_installed()** returns TRUE, emulating a fixed disk.

Note: The DRIVE_FLAGS_REMOVABLE flag must be set in **apiinit.c** for removable media. To do this, OR in DRIVE_FLAGS_REMOVABLE to the drive flags field.

pdr->drive_flags |= DRIVE_FLAGS_REMOVABLE;

To support removable trueide media you must modify this function to interface with your trueide media detect circuit. If media is installed it must return TRUE, if not it must return FALSE.

**Example:**

```
BOOLEAN trueide_card_installed(DDRIVE *pdr)
{
if (read_ide_installed_latch() == 1)
return(TRUE);
else
return(FALSE);
}
```

**void trueide_report_card_removed(int driveno)**

To support removable trueide media you must modify your media change interrupt service routine to call this function when a card has been removed.

Drive number of the card that was removed must be passed in.

The drive number must be the same as the value assigned to the pdr->driveno in **apiinit.c**.

Note: The DRIVE_FLAGS_REMOVABLE flag must be set in **apiinit.c** for removable media. To do this, OR in DRIVE_FLAGS_REMOVABLE to the drive flags field.

pdr->drive_flags |= DRIVE_FLAGS_REMOVABLE;

**Example:**

```
#define TRUEIDE_DRIVEID 2 – C:
void trueide_removal_interrupt(void)
{
trueide_report_card_removed(TRUEIDE_DRIVEID);
}
```

## SECTION 26: ERTFS LINEAR FLASH SUPPORT

ERTFS linear flash device support is provided through a portable Flash Translation layer (FTL) implemented in **drfldftl.c**.

The flash subsystem is included if the following line is true in portconf.h:

#define INCLUDE_FLASH_FTL 1 /* - Include the linear flash driver */

If INCLUDE_FLASH_FTL is zero, the flash subsystem is not included.

The FTL layer supports maps logical block addresses to physical block addresses and manages block replacement, spare block management and block wear leveling. A simple interface to underlying device specific Memory Technology Drivers (MTS's) is provided.

MTD's are implemented in **drflsmtd.c**. The requirements of MTD's are provided in the next section.

### FLASH MEMORY TECHNOLOGY DRIVERS

#### Introduction

The file **drflsmtd.c** contains two Memory Technology drivers. One is a driver that implements flash emulation in RAM, the other is a driver for Intel 28Fxxx flash parts. Other drivers may be implemented by editing three or four routines if **drflsmtd.c**. This section describes the required routines and the provided sample implementation for RAM emulation of flash and for Intel flash chips.

### ADDING YOUR OWN FLASH MEMORY TECHNOLOGY DRIVERS

To implement a new mtd driver you must implement custom versions of these four functions in this file.

- flash_probe() - must report if flash is present and the total size, the erase block size and the address and memory window width of the flash.
- flash_eraseblock() - must initialize one erase block of the flash.
- mtd_window() - must assure that a region of the flash is addressable.
- flash_write_bytes() - must program a region of the flash.

### FUNCTIONS THAT MUST BE PROVIDED TO SUPPORT A FLASH DEVICE

#### int flash_probe(void)

**flash_probe()** must determine if a flash chip is present and if so, determine the address of the flash, the total size of the flash, the size of an erase block, and the window of the flash that is addressable at any one time.

These values that must be set by the flash_probe routine:

- flashchip_TotalSize - Set this to the total size of the flash in bytes.
- flashchip_BlockSiz - Set this to the size of one erase block in bytes.
- flashchip_WindowSize - Set this to the addressable window. If the part is fully addressable, set it to the size of the part in bytes.
- flashchip_start - Set this byte pointer to the start of the flash.

**flash_probe()** must return 1 if a device is found, zero otherwise.

**void * mtd_MapWindow ()**

RTFDrvFlashData *DriveData

dword BlockIndex

dword WindowIndex)

mtd_MapWindow must map in a flashchip_WindowSize'ed region of the flash memory for reading and writing. The location of the flash region to map in is calculated by multiplying the BlockIndex times the erase block size (flashchip_BlockSize) and adding in the WindowIndex times the window map size (flashchip_WindowSize).

The included version of mtd_MapWindow assumes that the flash part is fully addressable. It simply multiplies these values together, adds them to the start of flash address and returns the result. This version will not need to be changed in most flat 32 bit target environments. In some environments it may be necessary to add software to perform some sort of bank register selection in this routine.

**flash_erase_block - Erase one flash erase-block.**

int flash_erase_block(dword BlockIndex)
flash_erase_block() must set the erase block of size flashchip_BlockSize at BlockIndex to the erased (all 1's) state.
flash_erase_block() must return 0 on success -1 on failure.

**int flash_write_bytes(byte volatile * dest, byte * src, int nbytes);**

flash_write_bytes() must take nbytes of data from the buffer at src and write it to the flash memory at address dest.

Dest is an address pointer for a location in a region of the flash that was returned by **mtd_MapWindow()**. The region between dest and dest plus nbytes is guaranteed to reside within flashchip_WindowSize bytes of the pointer returned by mtd_MapWindow().

flash_write_bytes must return 0 on success -1 on failure.

**SAMPLE MTD DRIVERS**

**Introduction**

Two sample MTD drivers are provided. One is a simple FLASH emulator implemented in RAM, the other is a driver for the Intel 28FXX flash series.

**Flash emulation in RAM**

If #define USE_EMULATED_FLASH is set to 1 the ram flash emulator is enabled. The size of the emulated Flash can be changed by changing the constant, FLASHEMUTOTALSIZE. The default is 64K. The flash memory is emulated in the array FlashEmuBuffer[]. The total size of FlashEmuBuffer[] is FLASHEMUTOTALSIZE bytes. The Ram emulator is very simple and may be used as a starting point for other flash device drivers.

If #define USE_DISK_EMULATOR is also set to 1, the ram will be mirrored to disk. Any time a write occurs, the disk file will be updated. This is only intended for testing purposes in Windows, and the disk emulation code is not portable.

**Intel Flash Chip Driver.**

If #define USE_INTEL_FLASH is set to 1, the INTEL flash chip driver is enabled.

This driver is for Intel several 28FXXX components between 2 and 8 megabytes in size. Other components from the series may be added by modifying the routine **flash_probe()** to recognize the device and to correctly report its total size, erase block size, and it's address.

Two compile time constants must be changed when porting the Intel MTD driver to a new target.

Two compile time constants tell the device driver the address of the Intel flash part and the width of the address range window through which the part may be read and written.

The defaults are arbitrarily set to ten million and 64 K respectively.

```
#define FLASH_STARTING_ADDRESS 0x10000000
#define FLASHWINDOWSIZE   64*1024L
```

These must be changed, set FLASH_STARTING_ADDRESS to the base of your flash memory and set FLASHWINDOWSIZE to one of the following:

If your flash part is fully linearly addressable from FLASH_STARTING_ADDRESS, set FLASHWINDOWSIZE to the size of the flash (in bytes). If it is addressable only through a memory bank that is smaller then the whole part, set FLASHWINDOWSIZE to the width of the window. The routine mtd_MapWindow() will be called to "seek" to the appropriate window each time a region of the flash is accessed.

## SECTION 27: ADDING YOUR OWN DEVICE DRIVERS

ERTFS device drivers are attached and initialized at run time. To implement one, you must provide two function entry points and attach them to the appropriate drive structure at run time. Section 6 explains this process in detail. This section explains in detail what is required of your device driver. In addition to studying this section you should also study existing device driver source code. All driver file names begin with the prefix, dr.

### SECTION 27.1: THE DEVICE BLOCK DATA TRANSFER FUNCTION

The device driver must provide a device block data transfer subroutine. The routine takes four arguments: the logical drive number, the sector number to read to or from, a count of 512 byte blocks to transfer, and a boolean flag reading. If reading is **TRUE**, this is a read request, if **FALSE**, it is a write request.

BOOLEAN xxx_io(driveno, sector, buffer, count, reading)

| | | |
|---|---|---|
| int driveno | - | Drive number |
| dword sector | - | Starting sector number to read or write |
| void *buffer | - | Buffer to read to write from |
| word count | - | Number of sectors to transfer |
| BOOLEAN reading | - | True for a read, False for a write request. Must return TRUE on success, FALSE on failure |

This is the device driver's read/write function. It is passed the drive number, a read or write flag, the first sector number on the device to read/write, the count of blocks, and the address of a buffer to move the data to or from. If the transfer succeeds, it must return TRUE, if it fails, it must return FALSE.

### SECTION 27.2: THE DEVICE I/O CONTROL FUNCTION

The device I/O control subroutine takes three arguments: the logical drive number, an op-code and a void pointer to a parameter passing memory area. The return values of return codes and the structure of the parameter passing array depend on which op-code is being processed. The requirements for each op-code are described in this section.

int xxx_perform_device_ioctl(driveno, opcode, pargs)

| | | |
|---|---|---|
| int driveno | - | driveno is 0,1,2,,25 for drives A:, B:, C:..Z: |
| int opcode | - | One of op-codes described below |
| void *pargs | - | A void pointer to an argument block. Currently the only defined use of pargs is to pass device geometry information from the driver to the ERTFS format tools. |

The device driver is required to implement an I/O control function that handles the following op-codes:

Device I/O Control op-codes that must be implemented

DEVCTL_WARMSTART

DEVCTL_CHECKSTATUS

DEVCTL_GET_GEOMETRY

DEVCTL_FORMAT

DEVCTL_REPORT_REMOVE

Device I/O Control op-codes that are optional.

DEVCTL_POWER_RESTORE

DEVCTL_POWER_LOSS

### SECTION 27.3: I/O CONTROL OP-CODE DEVCTL_WARMSTART

int xxx_perform_device_ioctl(driveno, DEVCTL_WARMSTART, 0)

Must return:

- If the device is a non-removable drive that is now accessible then return 0
- If the device is a non-removable drive that fails diagnostics, then return -1
- Return 0 if the device is a controller for removable media and the controller is accessible
- If the controller does not function return –1
- Do not return –1 if the controller is functional but no media is installed, return 0 in this case

This op-code is always called first before any other calls to the device driver. The device driver must use this call to perform initialization of the controller.

The simplest implementation of WARMSTART is one that that can't fail and requires no initialization, i.e., a ROMDISK driver). It will simply set the DRIVE_FLAGS_VALID flag in the pdr->drive_flags field and return zero. For other devices the WARMSTART routine is more complex. The drive structure is used for parameter passing during the WARMSTART call. You may want to use some of these parameters. *Note: Even if you don't use any of the fields in the drive structure you must still set the DRIVE_FLAGS_VALID bit in pdr->drive_flags.*

To access the drive structure, perform the following:

```
DDRIVE *pdr;

pdr = pc_drno_to_drive_struct(driveno);
```

**SECTION 27.3.1: PASSING PARAMETERS TO THE DEVICE DRIVER**

The ERTFS drive structure:

**pdr->pcmcia_slot_number** - For PCMCIA device the user may pass the PCMCIA slot number to the driver by setting this value in the drive structure before the WARMSTART call is made.

**pdr->pcmcia_cfg_opt_value** - This option is only valid for COMPACT FLASH cards. It is the user assigned value to be written to the cards configuration option register.

**pdr->controller_number** - This is set by the user before calling WARMSTART. In most cases this value is unused and should be 0. It may be used to tell the device driver to use another controller.

For example, most PC's today have two ATA controllers with separate I/O spaces and interrupts. To use the second controller the user must set pdr->controller_number to 1 before calling WARMSTART.

**pdr->logical_unit_number** - This is index of the device on the controller. In most cases this value should be 0 but it may be used to tell the device driver which device to use on the controller. For example, to mount an ATA slave device you should set **pdr->logical_unit_number** to 1.

**pdr->register_file_address** - This is an unsigned long value that the init function may use to pass an I/O address to the driver. This is shared only between the init function and the device driver. The device driver does not have to use it.

**pdr->interrupt_number** - This is an integer value that the init function may use to pass an interrupt number to the driver. This is shared only between the init function and the device driver. The device driver does not have to use it.

**pdr->partition_number** - If **DRIVE_FLAGS_PARTITIONED** is set by the user in the **pdr->drive_flags** field then this call to **WARMSTART** is for a partitioned device. The partition number assigned to the drive must be passed to the driver through this field. Other low level code in ERTFS reads this field too, so it is important set the field appropriately.

If the device driver supports removable media that contains several partitions, it must return **DEVTEST_CHANGED** the next time **DEVCTL_CHECKSTATUS** is requested for each of the partitions.

**SECTION 29.3.2: STATE FLAGS USED BY THE DEVICE DRIVER**

Bit flags are used for parameter passing during the **WARMSTART** call and later for driver state maintenance.

**pdr->drive_flags** - This field contains bit flags. Some of these flags must be set by the user before WARMSTART is called, others must be filled in by the device driver.

These must be set by the user before calling the **WARMSTART** routine.

**DRIVE_FLAGS_PARTITIONED** - The user must set this bit if this is a partitioned device. If so then the pdr->partition_ number field must contain the partition number (usually zero). ERTFS looks at this bit later on when it mounts the device to determine if it should interpret the first block on the device as a partition table.

*Note: When prd->partition_number is 0 and DRIVE_FLAGS_PARTITION is true and there is no partition table on the disk, but a valid BPB is found on the first block of the device, the mount will still succeed. This behavior supports certain media such as compact flash that may or may not contain a partition table.*

**DRIVE_FLAGS_PCMCIA** - The user must set this bit before calling WARMSTART to tell the device driver that this is a PCMCIA device. This is used by the IDE/Compact flash to distinguish between a native IDE device and a PCMCIA ATA device. ERTFS does not look at this bit. It is used only by the device drivers.

These bits may be must be set by device driver, not the user.

**DRIVE_FLAGS_VALID** - The driver must set this in the **WARMSTART** section if the controller is present and accessible. For Non-Removable media this must imply that the media is now accessible, for removable media this implies only that the controller is available, not that the media is accessible. If the **WARMSTART** routine returns 0 it must also set **DRIVE_FLAGS_VALID**.

**DRIVE_FLAGS_FORMAT** - The driver must set this if the media is not already formatted at power up. A simple example of a driver that sets this flag every time **WARMSTART** is called is the RAM disk driver. The linear flash driver sets this flag if it detects that the flash has not been formatted. If this bit is set when the **WARMSTART** call returns, the initialization code auto-formats the device before it returns.

**DRIVE_FLAGS_REMOVABLE** - The driver must set this in the **WARMSTART** call if the media is removable. If this bit is set, then ERTFS will call the **driver's DEVCTL_CHECKSTATUS I/O** control function each time it accesses the device to check for status changes.

**DRIVE_FLAGS_INSERTED** - The driver may use this flag to help track the state of removable media. It is never looked at by ERTFS but it is a useful bit for the device driver to have when fielding **DEVCTL_CHECKSTATUS** requests and **DEVCTL_REPORT_REMOVE** events.

**SECTION 27.4: I/O CONTROL OP-CODE DEVCTL_CHECKSTATUS**

xxx_perform_device_ioctl(driveno, DEVCTL_CHECKSTATUS, 0)

Must return one of the following.

**DEVTEST_NOCHANGE** - Non removable media should always return this. Removable media must return this if correct media is installed and has not changed since the last **DEVCTL_CHECKSTATUS** call.

**DEVTEST_NOMEDIA** - Removable media must return this if no media is installed.

**DEVTEST_UNKMEDIA** - Removable media must return this if media is installed but is not useable**.**

**DEVTEST_CHANGED** - Removable media must return this the media has been replaced since the last call for DEVCTL_CHECKSTATUS. The driver must then clear this status so it won't be reported again until another change event occurs.

**SECTION 27.5: I/O CONTROL OP-CODE DEVCTL_REPORT_REMOVE**

The **DEVCTL_REPORT_REMOVE IO** control call should be used as follows. When an external interrupt or monitoring task detects that the media associated with this drive number has been removed, it must call the driver's I/O control function with **DEVCTL_REPORT_REMOVE** as an op-code. The device driver must then put itself in a state such that the next time **DEVCTL_CHECKSTATUS** is called it should re-check if the media is installed and return **DEVTEST_NOMEDIA** if it is not. If media is installed, the **DEVCTL_CHECKSTATUS** routine must reinitialize the card. If that is successful, it may return either **DEVTEST_NOCHANGE** or **DEVTEST_CHANGED**. Under these circumstances it should return **DEVTEST_NOCHANGE** only if it is certain that the card that was re-inserted is the same card that was removed.

**SECTION 29.5.1: EXAMPLE CARD REMOVAL EVENT HANDLER**

The following code fragment reports a media removal event to the device driver. This simple system has one removable device mapped to the A: drive and it is designed such that a routine named **card_removed_interrupt()** is called whenever the card is removed.

```
#define REMOVABLE_DRIVE 0 /* System uses A: for this drive */
card_removed_interrupt()
{
    pdr = pc_drno_to_drive_struct(REMOVABLE_DRIVE);
    if (pdr)
        pdr->dev_table_perform_device_ioctl(pdr->driveno,
            DEVCTL_REPORT_REMOVE, (PFVOID) 0);
}
```

**SECTION 27.6: I/O CONTROL PROCESSING FOR DEVCTL_POWER_LOSS**

**xxx_perform_device_ioctl(driveno, DEVCTL_POWER_LOSS, 0)** Must return – 0

The **DEVCTL_POWER_LOSS IO** control call can be used to report to the device driver that it should go into low power mode and be aware that it must exit low power the next time the driver is called. The calling sequence is the same as when calling **DEVCTL_REPORT_REMOVE**, except the op-code is **DEVCTL_POWER_LOSS**. ERTFS does not implement this OPCODE in any drivers, but it is an appropriate interface for handling low power management.

The following code fragment reports a low power request to the device driver. This simple system has one power sensitive device mapped to the B: drive and it is designed such that a routine named **report_low_power_mode()** is called whenever the system enters low power mode.

```
#define POWER_AWARE_DRIVE 1 /* System uses B: for this drive */
report_low_power_mode()
{
    pdr = pc_drno_to_drive_struct(POWER_AWARE_DRIVE);
    if (pdr)
        pdr->dev_table_perform_device_ioctl(pdr->driveno,
            DEVCTL_POWER_LOSS, (PFVOID) 0);
}
```

**SECTION 27.7: I/O CONTROL OP-CODE DEVCTL_REPORT_RESTORE**

**xxx_perform_device_ioctl(driveno, DEVCTL_POWER_RESTORE, 0)** returns - nothing

The **DEVCTL_POWER_RESTORE IO** control call can be used to report to the device driver that power is restored. It can either exit low power mode now or ignore this signal and exit low power mode next time the driver is called. ERTFS does not implement this op-code in any drivers, but it may be an appropriate interface for handling power restore.

**SECTION 27.8: I/O CONTROL OP-CODE DEVCTL_GET_GEOMETRY**

**xxx_perform_device_ioctl(driveno, DEVCTL_GET_GEOMETRY, pargs)** returns - 0 if it successfully reported the correct geometry, otherwise -1.

pargs points to a structure of type **DEV_GEOMETRY**. This structure must be first zeroed out by the driver and then filled with appropriate values in either HCS format (head cylinder, sector) or in LBA (linear block address) format. This information is asked for only when ERTFS is asked to format or partition the device.

Here is an example implementation.

```
case DEVCTL_GET_GEOMETRY
{
DEV_GEOMETRY gc;
    rtfs_memset(&gc, 0, sizeof(gc));
    gc.dev_geometry_heads       = fill in these
    gc.dev_geometry_cylinders   = 3 fields with
        gc.dev_geometry_secptrack   = the HCN values, or leave them blank.
    gc.dev_geometry_lbas        = and put the lba value here
copybuff(pargs, &gc, sizeof(gc));
        return (0);
}
```

An alternate method is available that allows the device driver to return a specific format control structure. This method is used by the floppy disk driver so ERTFS formatted floppies have the identical structure as DOS floppies. Please study the source code for the floppy driver if you wish to use this alternate method.

**SECTION 27.9: I/O CONTROL OP-CODE DEVCTL_FORMAT**

xxx_perform_device_ioctl(driveno, DEVCTL_FORMAT, pargs)

returns - 0 if it the low level media is formatted, -1 otherwise.

**pargs** - points to a structure of type **DEV_GEOMETRY**. This structure contains the values reported by **DEVCTL_GET_GEOMETRY**.

If the device in question never needs low level formatting, this routine should just return 0 for success. If the device can be formatted (such as formatting a floppy disk), then the routine should format the device and return 0 on success or -1 on failure.

## SECTION 28: ERTFS APPLICATION PROGRAMMERS INTERFACE

Summary of ERTFS API Calls

**pc_ertfs_init**

This must be the first call to the ERTFS API in order to initialize ERTFS.

**Summary:**

BOOLEAN pc_ertfs_init(void)

**Description:**

This function works in conjunction with **pc_ertfs_config()** to configure and initialize ERTFS memory, target support services and device drivers. It also assigns drive letters to devices.

*Note: The section of the source code that selects device drivers and drive designators is designed to be user modifiable. Please see the Porting Section for detailed instructions on this process.*

**Returns:**

**TRUE** if all memory and system resource initialization succeed and ERTFS is usable, otherwise **FALSE**.

This function does not set errno.

**See Also:**

pc_ertfs_config

**Example:**

```
If (pc_ertfs_init()) /* Initialize ERTFS and start the test shell */
    tst_shell();
```

**pc_ertfs_config**

Do not call this function, but modify the source code.

**Summary:**

**Description:**

This function is called internally by **pc_ertfs_init**. Do not call it directly. **pc_ertfs_init** tells ERTFS what configuration values to use such as buffer size, the maximum number of drives, maximum number of files, and maximum number of users to be supported. It is also responsible for passing in the necessary memory and buffer space for the configuration being requested. To reconfigure ERTFS, please edit this routine in source file apicnfig.c. Section 3 goes into this in some detail.

**Returns:**

This function does not set errno.

**See Also:**

pc_ertfs_init

Example:

**pc_free_user**

Release this task's ERTFS user context block

**Summary:**

**#include <ertfs.h>**

void **pc_free_user()**

**Description:**

**This routine must be called by all tasks that have used ERTFS before they exit.**

When a task uses the ERTFS API, a user context block is automatically created specifically for that task. Before the task exits it must release it's context block, otherwise ERTFS will run out of context blocks and all new tasks will have to share the same context block.

**Returns:**

Nothing

**Example:**

```
void my_ftp_server_task()
{
    do_server_session();  /* Call the ftp server function here */
    pc_free_user();       /* Free ERTFS resources for this thread */
    exit(0);              /* Terminate the thread */
}
```

**pc_set_cwd**

Set current working directory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN **pc_set_cwd**(byte *path)

**Description:**

Make path the current working directory for this task. If path contains a drive component, the current working directory is changed for that drive, otherwise the current working directory is changed for the default drive.

**Returns:**

Returns **TRUE** if the current working directory was changed, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDPATH | - | Path specified badly formed. |
| PENOENT | - | Path not found |
| PEACCESS | - | Not a directory |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
if(!pc_set_cwd("D:\\USR\\DATA\\FINANCE"))
    printf("Can't change working directory\n");
```

**pc_set_default_drive**

Set the default drive.

**Summary**

**#include <ertfs.h>**

BOOLEAN pc_set_default_drive(byte *drive)

**Description:**

Use this function to set the current default drive that will be used when a path specifier does not contain a drive specifier.

**Returns:**

Returns **FALSE** if the drive is invalid.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |

**Example:**

```
#include <ertfs.h>
if(!pc_set_default_drive("C:"))
      printf("Can't change working drive\n");
```

**pc_pwd**

Return the current working directory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_pwd(byte *drive, byte *return_buffer)

**Description:**

Fill **return_buffer** with the full path name of the current working directory for the current task for the drive specified in **drive**. If **drive** points to an empty string **(“”)** or is an invalid drive specifier, the current working directory for the default drive is returned. In a multitasking system ERTFS maintains a current working directory for each task.

*Note: ERTFS must be configured correctly in order for each task have its own current working directory. Please see the documentation of the routine **pc_ertfs_config** for a complete explanation of this requirement.*

*Note: return_buffer must point to enough space to hold the full path without overriding user data. The worst case possible is 260 bites.*

**Returns:**

Returns YES if a valid path was returned in **return_buffer**, otherwise NO if the current working directory could not be found.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| An ERTFS system error | - | See Appendix J for a description of system errors |

***The failure mode would be due to either the fact that the drive is not mounted or an I/O error occurred.***

**Example:**

```
#include <ertfs.h>
byte  pwd[EMAXPATH];
if (pc_pwd(“A:”, pwd))
      printf (“Working dir is %s\n”,  pwd);
else
      printf (“Can’t find working dir for A:\n”);
```

**pc_gfirst**

Return the first entry in a directory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_gfirst(DSTAT *statobj, byte *pattern)

**Description:**

Given a pattern which contains both a path specifier and a search pattern, fill in the structure at statobj with information about the file and set up internal parts of statobj to supply appropriate information for calls to **pc_gnext**.

**Examples of patterns are:**

“D:\USR\RELEASE\NETWORK*.C”

“BIN\UU*.*”

“MEMO_?.*”

“*.*”

*Note: You must call pc_gdone to free internal resources if pc_gfirst succeeds.*

**Returns:**

Returns **TRUE** if a match was found, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPATH | - | Path specified badly formed. |
| PENOENT | - | Not found, no match |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**See Also:**

**pc_gnext**, **pc_gdone**, and dols() in apptstch.c

***Example:***

See PC_GNEXT

**pc_gnext**

Return next entry in a directory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_gnext(DSTAT *statobj)

**Description:**

Continue with the directory scan started by a call to **pc_gfirst**.

**Returns:**

Returns **TRUE** if a match was found, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDPARMS | - | statobj argument is not valid |
| PENOENT | - | Not found, no match (normal termination of scan) |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**See Also**

**pc_gnext**, **pc_gdone**, and **pcls.c** in the samples directory. This function does not set errno.

**Example:**

```
#include <ertfs.h>
if (pc_gfirst(&statobj,"A:\\dev\\*.c"))
{
    do
        {
            /* print file name, extension and size */
            printf("%-8s.%-3s %7ld \n",statobj.fname,
                    statobj.fext,statobj.fsize);
        }
    while (pc_gnext(&statobj));
    /* Call gdone to free up internal resources */
    pc_gdone(&statobj);
}
```

**pc_gdone**

Free directory scan resources originally allocated by **pc_first**.

**Summary:**

**#include <ertfs.h>**

void pc_gdone(DSTAT *statobj)

**Description:**

Given a pointer to a DSTAT structure that was set up by a call to **pc_gfirst** free internal elements used by the statobj.

*Note: You MUST call this function after you have finished calling pc_gfirst and pc_gnext or a memory leak will occur.*

**Returns:**

Nothing. pc_gdone does not set errno.

**Example:**

See pc_gnext

**pc_enumerate**

Recursively process all directory entries that match a pattern.

**Summary:**

```
int pc_enumerate(
byte * from_pattern_buffer    -  pointer to a scratch buffer of size EMAXPATH
byte * spath_buffer           -  pointer to a scratch buffer of size EMAXPATH
byte * dpath_buffer           -  pointer to a scratch buffer of size EMAXPATH
byte * root_search            -  Root of the search IE C:\ or C:\USR etc.
word match_flags              -  Selection flags (see above)
byte match_pattern            -  Match pattern (see above)
int  maxdepth                 -  Maximum depth of the traversal.
PENUMCALLBACK pcallback       -  User callback function (see below).
)
```

**Description:**

This routine traverses a subdirectory tree and tests each directory entry to see if it matches user supplied selection criteria. If it does match the criteria, a user supplied callback function is called with the full path name of the directory entry and a pointer to a DSTAT structure that contains detailed information about the directory entry (see the manual page for a detailed description of the DSTAT structure).

**Selection criteria:**

Two arguments are used to determine the selection criteria. One is a flags word that specifies attributes, the other is a pattern that specifies a wild card pattern.

The flags argument contains a bitwise or one or more of the following:

**MATCH_DIR** - Select directory entries

**MATCH_VOL** - Select volume labels

**MATCH_FILES** - Select files

**MATCH_DOT** - Select '.' entry MATH_DIR must be true too

**MATCH_DOTDOT** - Select '..' entry MATCH_DIR must be true too

The selection pattern is a standard wildcard pattern such as '*.*' or *.txt.

*Note: Patterns don't work the same for VFAT and DOS 8.3. If VFAT is enabled, the pattern *. will return any file name that has a '.' in it; in 8.3 systems it returns all files.*

*Note: pc_enumerate requires a fair amount of buffer space to function. Instead of allocating the space internally, we require that the application pass two buffers of size EMAXPATH in to the function. See below.*

*Note: to scan only one level set maxdepth to 1. For all levels set it to 99.*

**Return value:**

Returns 0 unless the callback function returns a non-zero value at any point. If the callback returns a non-zero value, the scan terminates immediately and returns the returned value to the application.

This function does not set errno.

**About the callback:**

The callback function returns an integer and is passed the fully qualified path to the current directory entry and a DSTAT structure. The callback function must return 0 if it wishes the scan to continue or any other integer value to stop the scan and return the callback's return value to the application layer.

**Example:**

**Example 1: Multilevel directory scan and print**

```
int rdir_callback(byte *path, DSTAT *d) {
      printf("%s\n", path);return(0);
}
rdir(byte *path, byte *pattern) {
      pc_enumerate(from_path,from_pattern,spath,
      dpath, path, (MATCH_DIR|MATCH_VOL|MATCH_FILES),
      pattern, 99, rdir_callback);
}
```

**Example 2: Poor mans deltree package**

```
int delfile_callback(byte *path, DSTAT *d) {
      pc_unlink(path);   return(0);
}
int deldir_callback(byte *path, DSTAT *d) {
      pc_rmdir(path); return(0);
}
deltree(byte *path)
{
int i;
=>  First delete all of the files
      pc_enumerate(from_path,from_pattern,spath,
      dpath,path,(MATCH_FILES),
      "*",99, delfile_callback);
      i = 0;
=>  Now delete all of the dirs. Deleting path won't
      work until the tree is empty
      while(!pc_rmdir(path) && i++ < 50)
            pc_enumerate(from_path,from_pattern,spath,
            dpath,path, (MATCH_DIR), "*", 99,
            deldir_callback);
```

**get_errno**

Get the last ERTFS assigned errno value for the calling task

**Summary:**

int get_errno(void)

**Description:**

This function retrieves the last ERRNO value set by ERTFS for this task.

**See Also:**

**Example:**

```
If (!pc_mkdir("Test"))
        printf("mk_dir failed: ERRNO == %d\n", get_errno());
```

**rtfs_set_driver_errno**

Provided so device driver writers can set device driver specific 32 bit error codes.

**Summary:**

void rtfs_set_driver_errno(dword error)

**Description:**

The device driver error code is similar to errno. Each task has its own driver error code and the application can retrieve it by calling **rtfs_set_driver_errno**.

At this time ERTFS does not generate driver error codes or interpret them but in the future we may formalize some values.

For some application environments it is convenient if device drivers signal to the application such conditions as write protect and card removal. Then the application can try to correct the problem and restart the operation.

*Note: Do not call rtfs_set_driver_errno from an interrupt service routine.*

**Example:**

A write is attempted to a read only device.

```
#define CARD_IS_WRITE_ONLY 0x12345678
BOOLEAN romdisk_io(driveno, block, buffer, count, reading)
{
    if (!reading)
    {
    rtfs_set_driver_errno(CARD_IS_WRITE_PROTECTED) ;
    return(FALSE);
    }
}
```

**rtfs_get_driver_errno**

This API call is provided so applications developers can retrieve device driver specific 32 bit error codes.

**Summary:**

dword rtfs_get_driver_errno(void)

**Description:**

The device driver error code is similar to errno. Each task has its own driver error code and the application can retrieve it by calling **rtfs_set_driver_errno**.

For some application environments it is convenient if device drivers signal to the application such conditions as write protect and card removal. Then the application can try to correct the problem and restart the operation.

**Example:**

An applicatoion attempts to create a directory on a read only device.

```
#define CARD_IS_WRITE_ONLY 0x12345678
if(!pc_mkdir("DATADIRECTORY"))
{
      if (rtfs_get_driver_errno() == CARD_IS_WRITE_ONLY)
            printf("Can't create directory, card is write protected\n");
}
```

**pc_free**

Return disk free space statistics

**Summary:**

**#include <ertfs.h>**

long pc_free(byte *drive, dword *total blocks, dword *free blocks);

**Description:**

Given a drive ID, return the number of free bytes on the drive, the total number of blocks on the drive and the number of blocks free.

**Returns:**

The number of bytes free on the disk.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
    free_bytes = pc_free (“A:”, & total_blocks, & free_blocks);
    printf (“%ld free bytes \n:”, free_bytes);
    printf (“%ld blocks free out of %ld \n:”, free_blocks,
        total_blocks);
```

**pc_get_attributes**

Get File Attributes of the named file

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_get_attributes(byte *path, byte *p_return);

**Description:**

Given a file or directory name, return the directory entry attributes associated with the entry.

One or more of the following values will be or'ed together:

| BIT | Nemonic |
|---|---|
| 0 | ARDONLY |

**Returns:**

Returns **TRUE** if successful, otherwise it returns **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PENOENT | - | Path not found |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
byte attribs;
    if (pc_get_attributes("A:\\COMMAND.COM", &attribs)
    {
        if (attribs & ARDONLY)
            printf("File is %s\n", "ARDONLY");
        if (attribs & AHIDDEN)
            printf("File is %s\n", "AHIDDEN");
        if (attribs & ASYSTEM)
            printf("File is %s\n", "ASYSTEM");
        if (attribs & AVOLUME)
            printf("File is %s\n", "AVOLUME");
        if (attribs & ADIRENT)
            printf("File is %s\n", "ADIRENT");
        if (attribs & ARCHIVE)
            printf("File is %s\n", "ARCHIVE");
    }
```

**pc_isdir**

Test if a path is a directory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_isdir(byte *path)

**Description:**

This is a simple routine that opens a path and checks if it is a directory, then closes the path. The program cp2pc in the samples directory uses it to test if a destination is a directory. The same functionality can be gotten by calling **pc_gfirst** and testing the DSTAT structure.

**Returns:**

Returns **TRUE** if path points to a valid existing directory, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PENOENT | - | Path not found |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**pc_isvol**

Test if a path name is a volume label.

**Summary:**

BOOLEAN pc_isvol(byte *path)

**Description:**

Tests to see if a path specification is a volume label specifier.

**Returns:**

Returns **TRUE** if it is a volume, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PENOENT | - | Path not found |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**pc_stat**

Return properties of a named file or directory.

**Summary:**

**#include <ertfs.h>**

int pc_stat(byte *name,ERTFS_STAT *pstat)

**Description:**

This routine searches for the file or directory provided in the first argument.

If found, it fills in the stat structure as described here:

**st_dev** - the entry's drive number
**st_mode;**
- **S_IFMT** - type of file mask
- **S_IFCHR** - character special (unused)
- **S_IFDIR** - directory
- **S_IFBLK** - block special (unused)
- **S_IFREG** - regular (a "file")
- **S_IWRITE** - Write permitted
- **S_IREAD** - Read permitted.

**st_rdev** - the entry's drive number
**st_size** - file size
**st_atime** - creation date in DATESTR format
**st_mtime** - creation date in DATESTR format
**st_ctime** - creation date in DATESTR format
**t_blksize** - optimal blocksize for I/O (cluster size)
**t_blocks** - blocks allocated for file
**fattributes** - the DOS attributes. This is non-standard but supplied if you want to look at them.

*NOTE: ERTFS_STAT structure is equivalent to the STAT structure available with most posix like run time environments. Unfortunately certain run time environments like uITRON also use a structure named STAT so in order to avoid namespace collisions ERTFS uses the proprietary name ERTFS_STAT. If you are porting an application that uses STAT you may put the following preprocessor macro in ERTFS.H just below the declaration of ERTFS_STAT:*

*#define STAT ERTFS_STAT*

**Returns:**

Returns 0 if successful, otherwise -1.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PENOENT | - | File or directory not found |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
struct ERTFS_stat st;
if (pc_stat("A:\\MYFILE.TXT", &st)==0)
{
      printf("DRIVENO: %02d SIZE: %7ld DATE:%02d-%02d-%02d
                  TIME:%02d:%02d\n",
      st.st_dev,
      st.st_size,                         /* Size in bytes */
      (st.st_atime.date >> 5 ) & 0xf,     /* Month */
      (st.st_atime.date & 0x1f),          /* Day */
      80 +(st.st_atime.date >> 9) & 0xff,           /* Year */
      (st.st_atime.time >> 11) & 0x1f,    /* Hour */
      (st.st_atime.time >> 5) & 0x3f);    /* Minute */
      printf("OPTIMAL BLOCK SIZE: %7ld FILE size (BLOCKS): %7ld\n",
      st.st_blksize,st.st_blocks);
      printf("MODE BITS :");
      if (st.st_mode&S_IFDIR)
            printf("S_IFDIR|");
      if (st.st_mode&S_IFREG)
            printf("S_IFREG|");
      if (st.st_mode&S_IWRITE)
            printf("S_IWRITE|");
      if (st.st_mode&S_IREAD)
            printf("S_IREAD\n");
      printf("\n");
      }
```

**pc_fstat**

Return properties of a file associated with a file descriptor.

**Summary:**

**#include <ertfs.h>**

int pc_fstat(PCFD file_descriptor, ERTFS_STAT *pstat)

**Description:**

This routine uses the file descriptor in the first argument and fills in the stat structure as described here.

**st_dev** - the entry's drive number
**st_mode;**
- **S_IFMT** - type of file mask
- **S_IFCHR** - character special (unused)
- **S_IFDIR** - directory
- **S_IFBLK** - block special (unused)
- **S_IFREG** - regular (a "file")
- **S_IWRITE** - Write permitted
- **S_IREAD** - Read permitted.

**st_rdev** - the entry's drive number
**st_size** - file size
**st_atime** - creation date in DATESTR format
**st_mtime** - creation date in DATESTR format
**st_ctime** - creation date in DATESTR format
**t_blksize** - optimal blocksize for I/O (cluster size)
**t_blocks** - blocks allocated for file
**fattributes** - the DOS attributes. This is non-standard but supplied if you want to look at them.

***NOTE: ERTFS_STAT structure is equivalent to the STAT structure available with most posix like run time environments. Unfortunately certain run time environments like uITRON also use a structure named STAT so in order to avoid namespace collisions ERTFS uses the proprietary name ERTFS_STAT. If you are porting an application that uses STAT you may put the following preprocessor macro in ERTFS.H just below the declaration of ERTFS_STAT.***

***#define STAT ERTFS_STAT***

**Returns:**

Returns 0 if all went well, otherwise it returns -1.

errno is set to one of the following:

0 - No error
PEBADF - Invalid file descriptor

PECLOSED - Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition.

**Example:**

```
#include <ertfs.h>
struct ERTFS_stat st;
PCFD fd;
fd = po_open("A:\\MYFILE.TXT",(PO_BINARY|PO_RDONLY),0);
if (pc_fstat(fd, &st)==0)
{
printf("DRIVENO: %02d SIZE: %7ld DATE:%02d-%02d-%02d TIME:%02d:%02d\n",
st.st_dev,
st.st_size,                             /* Size in bytes */
(st.st_atime.date >> 5 ) & 0xf,                 /* Month */
(st.st_atime.date & 0x1f),              /* Day */
80 +(st.st_atime.date >> 9) & 0xff,             /* Year */
(st.st_atime.time >> 11) & 0x1f,        /* Hour */
(st.st_atime.time >> 5) & 0x3f);        /* Minute */
        printf("OPTIMAL BLOCK SIZE: %7ld FILE size (BLOCKS): %7ld\n",
            st.st_blksize,st.st_blocks);
        printf("MODE BITS :");
        if (st.st_mode&S_IFDIR)
            printf("S_IFDIR|");
        if (st.st_mode&S_IFREG)
            printf("S_IFREG|");
        if (st.st_mode&S_IWRITE)
            printf("S_IWRITE|");
        if (st.st_mode&S_IREAD)
            printf("S_IREAD\n");
        printf("\n");
    }
```

**pc_check_disk**

Check a volume's integrity

**Summary:**

BOOLEAN pc_check_disk(byte *drive_id,
CHKDISK_STATS *pstat, int verbose, int fix_problems, int write_chains)

**Description:**

This routine scans the disk searching for lost chains and crossed files and returns information about the scan in the structure at pstat. If **fix_problems** is non-zero it corrects file sizes if necessary. If **fix_problems** is non-zero and if **write_chains** is zero, it frees lost cluster chains; if **write_chains** is non-zero, it writes lost chains to files names FILE???.CHK in the root directory. If **fix_problems** is zero the **write_chains** argument is ignored.

**pstat** - a pointer to a structure of type CHKDISK_STATS. **pc_check_disk** returns information about the disk in this structure.

**verbose -** If this parameter is 1 pc_check_disk() prints status information as it runs, if it is 0 pc_check_disk() runs silently.

**fix_problems** - If this parameter is 1 pc_check_disk() will make repairs to the volume, if it is zero, problems are reported but not fixed.

**write_chains** - If this parameter is 1 pc_check_disk() creates files from lost chains. If write_chains is 0 lost chains are automatically discarded and freed for re-use. If fix_problems is 0 then write_chains has no affect.

```
typedef struct chkdisk_stats {
    dword  n_user_files;            /* Total #user files found */
    dword  n_hidden_files;          /* Total #hidden files found */
    dword  n_user_directories;      /* Total #directories found */
    dword  n_free_clusters;         /* # free available clusters */
    dword  n_bad_clusters;          /* # clusters marked bad */
    dword  n_file_clusters;         /* Clusters in non hidden files */
    dword  n_hidden_clusters;       /* Clusters in hidden files */
    dword  n_dir_clusters;          /* Clusters in directories */
    dword  n_crossed_points;        /* Number of crossed chains. */
    dword  n_lost_chains;           /* # lost chains */
    dword  n_lost_clusters;         /* # lost clusters */
    dword  n_bad_lfns;              /* # corrupt/disjoint win95 lfn chains */
} CHKDISK_STATS;
```

**Returns:**

**TRUE** if there was no ERROR, otherwise **FALSE**.

errno is set to one of the following:

pc_check_disk() does not set errno.

**Example:**

```
CHKDISK_STATS chkstat;
pc_check_disk("A:", &chkstat, 1, 1, 0); /* Check disk, be verbose, fix problems, free lost chains */
pc_check_disk("A:", &chkstat, 1, 1, 1); /* Check disk, run quietly, fix problems, convert lost chains to files */
    return(0);
}
```

**pc_mkdir**

Create a subdirectory.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_mkdir(byte *path)

**Description:**

Create a subdirectory in the path specified by path. Fails if a file or directory of the same name already exists or if the directory component (if there is one) of path is not found.

**Returns:**

Returns **TRUE** if the subdirectory was created, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPATH | - | Path specified badly formed. |
| PENOENT | - | Path to new directory not found |
| PEEXIST | - | File or directory of this name already exists |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>

pc_mkdir("\\USR\\LIB\\HEADER\\SYS");
```

**pc_mv**

Rename files and directories

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_mv(char *oldpath, char *newpath)

**Description:**

Moves the file or subdirectory named oldpath to the new name specified in newpath. Oldpath and Newpath must be on the same drive but they may be in different directories. Both names must be fully qualified (see examples). Fails if newpath is invalid, already exists or oldpath is not found.

**Returns:**

Returns **TRUE** if the file was renamed, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid or they are not the same |
| PEINVALIDPATH | - | Path specified by old_name or new_name is badly formed. |
| PEACCESS | - | File or directory in use, or old_name is read only |
| PEEXIST | - | new_name already exists |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
/* Move the a file named \USR\LETTER.TXT to LETTER.OLD in the current working directory */
if (!pc_mv("\\USR\\TXT\\LETTER.TXT", "LETTER.OLD"))
printf("Can't move LETTER.TXT\n");
/* Move the 'a' directory named \users\summerhelp\joe to users\oldemployees\joe */
if (!pc_mv(\\users\\summerhelp\\joe, "\\users\\oldemployees\joe")
    printf("Can't move (\\users\\summerhelp\\joe\n");
```

**pc_rmdir**

Delete a directory

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_rmdir(byte *name)

**Description:**

Delete the directory specified in path. Fails if path is not a directory, is read only or is not empty.

**Returns:**

**TRUE** if the directory was successfully removed, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPATH | - | Path specified badly formed. |
| PENOENT | - | Directory not found |
| PEACCESS | - | Directory is in use or is read only |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
if (!pc_rmdir("D:\\USR\\TEMP")
    printf("Can't delete directory\n");
```

**pc_deltree**

Delete a directory tree

**Summary:**

**#include<ertfs.h>**

BOOLEAN pc_deltree(byte *directory)

**Description:**

Delete the directory specified in name, all subdirectories of that directory, and all files contained therein. Fail if name is not a directory, is read only or is currently in use.

*Note: If a portion of the tree being deleted is in use, either with an open file or directory traversal, then the deltree algorithm will abort leaving the tree partially removed.*

**Returns:**

Returns **TRUE** if the directory was successfully removed.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive name is invalid |
| PEINVALIDPATH | - | Path specified by name is badly formed. |
| PENOENT | - | Can't find path specified by name. |
| PEACCES | - | Directory or one of its subdirectories is read only or in use. |
| An ERTFS system error | - | See Appendix J for a description of System Errors |

**pc_set_attributes**

Set File Attributes

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_set_attributes(byte *path, byte attributes)

**Description:**

Given a file or directory name set the directory entry attributes associated with the entry. One or more of the following values may be or'ed together.

| BIT | Nemonic |
|---|---|
| 0 | ARDONLY |

**Returns:**

Returns **TRUE** if successful, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDPARMS | - | Attribute argument is invalid |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPATH | - | Path specified badly formed |
| PENOENT | - | Path not found |
| PEACCESS | - | Object is read only |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
byte attribs;
if (pc_get_attributes("A:\\COMMAND.COM", &attribs)
{
    attribs |= ARDONLY|AHIDDEN
    pc_set_attributes("A:\\COMMAND.COM", attribs);
}
```

**pc_unlink**

Delete a file.

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_unlink(byte *path)

**Description:**

Delete the file in name. Fail if not a simple file, if it is open, does not exist, or is read only.

**Returns:**

Returns **TRUE** if it successfully deleted the file, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPATH | - | Path specified badly formed. |
| PENOENT | - | Can't find file to delete |
| PEACCESS | - | File in use, is read only or is not a simple file. |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
if (!pc_unlink("B:\\USR\\TEMP\\TMP001.PRN") )
    printf("Cant delete file \n")
```

**pc_diskflush()**

Flush the FAT and all files to a disk

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_diskflush(byte *path)

**Description:**

Given a path containing a valid drive specifier, flush the file allocation table and all changed files to the disk. After this call returns, the disk image is synchronized with an ERTFS internal view of the volume.

**Returns:**

Returns **TRUE** if the disk flush succeeded, otherwise **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>

        if (!pc_diskflush("A:"))
            printf("Flush operation failed \n");
```

**po_open**

Open a file.

**Summary:**

**#include <ertfs.h>**

PCFD po_open(byte *path, word flag, word mode)

**Description:**

Open the file for access as specified in flag. If creating use mode to set the access permissions.

**Flag values are**:

| | | |
|---|---|---|
| **PO_APPEND** | - | All writes will be appended to the file |
| **PO_BINARY** | - | Ignored |
| **PO_TEXT** | - | Ignored |
| **PO_RDONLY** | - | Open for read only |
| **PO_RDWR** | - | Read/write access allowed |
| **PO_WRONLY** | - | Open for write only |
| **PO_CREAT** | - | Create the file if it does not exist |
| **PO_EXCL** | - | If flag has (PO_CREAT\|PO_EXCL) and the file already exists, fail and set **fs_user->p_errno** to EEXIST |
| **PO_TRUNC** | - | Truncate the file if it already exists |
| **PO_BUFFERED** | - | If this is set, reads and writes of less than 512 bytes and operations that do not start or end on block boundaries are buffered. The buffer is flushed when **po_close** is called, when **po_flush** is called or if a buffered IO request is made to a different block number. Using the PO_BUFFERED flag increases performance of applications performing reads and writes of small or un alligned data buffers. |
| **PO_AFLUSH** | - | Enable auto flush mode. The file is flushed automatically by **po_write** whenever the file length changes. |
| **PO_NOSHAREANY** | - | Fail if already open, fail if another open is tried |
| **PO_NOSHAREWRITE** | - | Fail if already open for write and fail if another open for write is tried |

**Mode values are:**

| | | |
|---|---|---|
| **PS_IWRITE** | - | Write permitted |
| **PS_IREAD** | - | Read permitted (Always true anyway) |

**Returns:**

Returns a non-negative integer to be used as a file descriptor for calling **po_read**, **po_write**, **po_lseek**, **po_flush**, **po_truncate**, and **po_close**, otherwise it returns -1.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PENOENT | - | Not creating a file and file not found |
| PEMFILE | - | Out of file descriptors |
| PEINVALIDPATH | - | Invalid pathname |
| PENOSPC | - | No space left on disk to create the file |
| PEACCES | - | Is a directory or opening a read only file for write |
| PESHARE | - | Sharing violation on file opened in exclusive mode |
| PEEXIST | - | Opening for exclusive create but file already exists |
| PEEXIST | - | Opening for exclusive create but file already exists |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

**#include <ertfs.h>**

```
PCFD fd;
    if(pcfd=po_open("\\USR\\MYFILE",(PO_CREAT|PO_EXCL|PO_WRONLY)
            ,P S_IWRITE)<0))
        printf("Cant create file error:%i\n" ,fs_user->p_errno)
```

**po_close**

Close a file that was opened with po_open.

**Summary:**

**#include <ertfs.h>**

int po_close(PCFD fd)

**Description:**

Close the file and update the disk by flushing the directory entry and file allocation table, then free all core associated with FD.

**Returns:**

Returns 0 if all went well, otherwise -1.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**See Also:**

po_flush

**Example:**

```
#include <ertfs.h>
if (po_close(fd) < 0)
    printf("Error closing file:%i\n",p_errno);
```

**po_flush**

Flush a file to disk.

**Summary:**

#include <ertfs.h>

**BOOLEAN po_flush(PCFD fd)**

**Description:**

Write the file's directory entry to disk and flush the FAT. After this call completes, the on disk view of the file is completely consistent with the in memory view. It is a good idea to call this function periodically if a file is being extended. If a file is not flushed or closed and a power down occurs, the file size will be wrong on disk and the FAT chains will be lost.

**Returns:**

Returns **TRUE** if flush succeeded, otherwise **FALSE** is returned.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEACCES | - | File is read only |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
if (po_flush(fd) < 0)
    printf("Error flushing file:%i\n",p_errno);
```

**See Also:**

pc_dskflush()

**po_read**

Read from a file.

**Summary:**

**#include <ertfs.h>**

int po_read(PCFD fd, byte *buf, int count)

**Description:**

Attempt to read **count** bytes from the current file pointer of file at **fd** and place the data in **buf**. The file pointer is updated.

**Returns:**

Returns the actual number of bytes read or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEIOERRORREAD | - | Read error |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
PCFD fd;
PCFD fd2;
fd = po_open(“FROM.FIL”,PO_RDONLY,0);
fd2 =po_open(“TO.FIL”,PO_CREAT|PO_WRONLY,PS_IWRITE)
if (fd >= 0 && fd2 >= 0)
      while (po_read(fd, buff, 512) ==512)
            po_write(fd2, buff, 512);
```

**po_write**

Write to a file.

**Summary:**

**#include <ertfs.h>**

int po_write(PCFD fd, UNITY *buf, int count)

**Description:**

Attempt to write count bytes from **buf** to the current file pointer of file at **fd**. The file pointer is updated.

**Returns:**

Returns the number of bytes written or -1 on error.

*Note: If the requested file length exceeds RTFS_MAX_FILE_SIZE, po_write may be configured to either truncate the write size or to return an error, depending on configurations values set in rtfsconf.h. If RTFS_TRUNCATE_WRITE_TO_MAX is set to one the write tries to complete up to that many bytes and returns the number of bytes written. If it is 0, no writing occurs, errno is set to PETOOLARGE and -1 is returned.*

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEACCES | - | File is read only |
| PEIOERRORWRITE | - | Error performing write |
| PEIOERRORREAD | - | Error reading block for merge and write |
| PENOSPC | - | Disk full |
| PETOOLARGE | - | Trying to extend a file beyond RTFS_MAX_FILE_SIZE |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
PCFD fd;
PCFD fd2;
fd = po_open(“FROM.FIL”,PO_RDONLY,0);
fd2 =po_open(“TO.FIL”,PO_CREAT|PO_WRONLY,PS_IWRITE)
if (fd >= 0 && fd2 >= 0)
      while (po_read(fd, buff, 512) ==512)
            po_write(fd2, buff, 512);
```

**po_lseek**

Move file pointer

**Summary:**

**#include <ertfs.h>**

long po_lseek(PCFD fd, long offset, int origin)

**Description:**

Move the file pointer **offset** bytes from the origin described by **origin**. **Origin** may have the following values:

**PSEEK_SET** - Seek from beginning of file

**PSEEK_CUR** - Seek from the current file pointer

**PSEEK_END** - Seek from end of file

Attempting to seek beyond end of file puts the file pointer one byte past end of file. Seeking zero bytes from origin **PSEEK_END** returns the file length.

**Returns:**

The new offset or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEINVALIDPARMS | - | Attempt to seek past EOF or to a negative offset |
| PEINVALIDCLUSTER | - | Files contains a bad cluster chain |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
#include <ertfs.h>
record = rec_number * rec_size;
if (po_lseek(fd, record , PSEEK_SET) != record)
      printf("Cant find record %ld\n",record);
```

**po_ulseek**

Move file pointer (unsigned)

**Summary:**

**#include <ertfs.h>**

BOOLEAN po_useek(PCFD fd, unsigned long offset, unsigned long *pnew_offset, int origin)

**Description:**

Move the file pointer **offset** bytes from the origin described by **origin**. **Origin** may have the following values:

| | | |
|---|---|---|
| **PSEEK_SET** | - | Seek from beginning of file |
| **PSEEK_CUR** | - | Seek from the current file pointer |
| **PSEEK_CUR_NEG** | - | Seek from the current file pointer minus unsigned 32 bit offset |
| **PSEEK_END** | - | Seek from end of file |

The new file pointer is returned in *pnew_offset

**Returns:**

TRUE - The seek was succesful and the new offset is in *pnew_offset.
FALSE - An error occured.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEINVALIDPARMS | - | Attempt to seek past EOF or to a negative offset |
| PEINVALIDCLUSTER | - | Files contains a bad cluster chain |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**po_chsize**

Truncate or extend an open file.

**Summary:**

**#include <ertfs.h>**

int po_chsize(PCFD fd, long offset)

**Description:**

Given a file handle and a new file size, either extend the file or truncate it. If the current file pointer is still within the range of the file, it is unmoved, otherwise it is moved to the end of file. This function uses other API calls and does not set errno itself.

*Note: This is not an ATOMIC file system operation. It uses other API calls po_lseek, po_truncate and po_write to size, truncate and extend the file.*

**Returns:**

Returns 0 on success, -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEACCES | - | File is read only |
| PEINVALIDPARMS | - | Invalid or inconsistent arguments |
| PETOOLARGE | - | Trying to extend a file beyond RTFS_MAX_FILE_SIZE |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**po_truncate**

Truncate an open file.

**Summary:**

**#include <ertfs.h>**

BOOLEAN po_truncate(PCFD fd, long offset)

**Description:**

Truncates the open file at **fd** to newsize. Any file blocks beyond newsize are freed and the file size is adjusted. The file pointer is left at the new end of file.

**Returns:**

Returns **TRUE** if po_truncate succeeded, otherwise, **FALSE** is returned.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEACCES | - | File is read only or opened more than once |
| PEINVALIDPARMS | - | Invalid or inconsistent arguments |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

```
fd = po_open("DATA.FIL",PO_RDWR,0);
if (fd > 0)
    po_truncate(fd, 1024L);
```

**pc_get_media_parms()**

Get device geometry for a named device.

**Summary:**

BOOLEAN pc_get_media_parms(byte *path, PDEV_GEOMETRY pgeometry)

**Description:**

Query the drive's associated device driver for a description of the installed media. This information is used by the **pc_format_media**, **pc_partition_media** and **pc_format_volume** routines. The application may use the results of this call to calculate how it wishes the media to be partitioned.

*Note: The floppy device driver uses a "back door" to communicate with the format routine through the geometry structure. This allows us to not have floppy specific code in the format routine but still use the exact format parameters that DOS uses when it formats a floppy.*

See the following definition of the pgeometry structure:

```
typedef struct dev_geometry {
int dev_geometry_heads;          — - Must be < 256
int dev_geometry_cylinders;      — - Must be < 1024
int dev_geometry_secptrack;      — - Must be < 64
dword dev_geometry_lbas;         — - For oversized media that supports logical block addressing.  If this is non-zero dev_ge-
                                        ometry_cylinders is ignored but dev_geometry_heads and dev_geometry_
                                        secptrack must still be valid.
BOOLEAN fmt_parms_valid;         — - If the device I/O control call sets this TRUE, then it tells the applications
                                       — layer that these format parameters should be used. This is a way to format
                                      — floppy disks exactly as they are formatted by dos.
    FMTPARMS fmt;
} DEV_GEOMETRY;
typedef struct dev_geometry KS_FAR *PDEV_GEOMETRY;
```

**Returns:**

Returns **TRUE** if it was able to get the parameters, otherwise it returns **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEDEVICEFAILURE | - | Device driver get device geometry request failed |
| PEINVALIDPARMS | - | Device driver returned bad values |

**See Also:**

**pc_format_media**, **pc_partition_media**, **pc_format_volume**

**Example:**

*Note: This routine is designed to work in a specific context. See the documentation for routine pc_format_volume() for example usage.*

**pc_partition_media**

Partition a disk

**Summary:**

BOOLEAN pc_partition_media(byte *path, PDEV_GEOMETRY pgeometry, PFWORD partition_list)

**Description:**

Writes a partition table onto the disk at path. **pgeometry** contains the values returned by **pc_get_media_parms**. **partition_list** must be provided by the user. It is a null terminated list of partition sizes. If **pgeometry->dev_geometry_lbas** is non-zero, the partition sizes must be entered in units of logical blocks. If **pgeometry->dev_geometry_lbas** is zero the partition sizes must be entered in units of cylinders.

**pc_get_media_parms**, **pc_format_media**, **pc_format_volume**

**Returns:**

Returns **TRUE** if it was able to perform the operation, otherwise it returns **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEINVALIDPARMS | - | Inconsistent or missing parameters |
| PEIOERRORWRITE | - | Error writing partition table |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

*Note: This routine is designed to work in a specific context. See the documentation for routine pc_format_volume() for example usage.*

**pc_format_media**

Perform a device level format

**Summary:**

BOOLEAN pc_format_media(byte *path, PDEV_GEOMETRY pgeometry)

path is the device's drive id (A:, B: etc).

**pgeometry** is the value returned **pc_get_media_parms**

**Description:**

This routine performs a device level format on the specified device.

**Returns:**

Returns **TRUE** if it was able to perform the operation, otherwise it returns **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEDEVICEFAILURE | - | Device driver format request failed |

**See Also:**

pc_get_media_parms, pc_partition_media, pc_format_volume

**Example:**

*Note: This routine is designed to work in a specific context. See the documentation for routine pc_format_volume() for example usage.*

**pc_format_volume**

Perform a volume format

**Summary:**

BOOLEAN pc_format_volume(byte *path, PDEV_GEOMETRY pgeometry)

**Description:**

This routine formats the volume referred to by drive letter. Drive structure is queried to determine if the device is partitioned or not. If the device is partitioned, the partition table is read and the volume within the partition is formatted. If it is a non-partitioned device, the device is formatted according to the supplied pgeometry parameters. The pgeometry parameter contains the media size in HCN format.

**See Also:**

pc_get_media_parms, pc_partition_media, pc_format_media

**Returns:**

Returns **TRUE** if it was able to perform the operation, otherwise it returns **FALSE**.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive component is invalid |
| PEIOERRORREADMBR | - | Partitioned device. I/O error reading |
| PEINVALIDMBR | - | Partitioned device has no master boot record |
| PEINVALIDMBROFFSET | - | Requested partition has no entry in master boot record |
| PEINVALIDPARMS | - | Inconsistent or missing parameters |
| PEIOERRORWRITE | - | Error writing during format |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**Example:**

The following subroutine is included in the test shell program apptstsh.c. It may be modified for your application:

```
        *****
int doformat(int agc, byte **agv)
{
byte buf[10];
byte working_buffer[100];
DDRIVE *pdr;
int driveno;
dword partition_list[3];
byte path[10];
DEV_GEOMETRY geometry;
        /* "Enter the drive to format as A:, B: etc " */
        rtfs_print_prompt_user(UPROMPT_TSTSH3, path);
        pdr = 0;
        driveno = pc_parse_raw_drive(path);
        if (driveno != -1)
        pdr = pc_drno_to_drive_struct(driveno);
        if (!pdr)
        {
inval:
        /* "Invalid drive selection to format, press return" */
        rtfs_print_prompt_user(UPROMPT_TSTSH4, working_buffer);
        return(-1);
}

if ( !(pdr->drive_flags&DRIVE_FLAGS_VALID) )
{
        goto inval;
}

OS_CLAIM_LOGDRIVE(driveno)
/* check media and clear change conditions */
if (!check_drive_number_present(driveno))
{
        /* "Format - check media failed. Press return" */
```

```
        rtfs_print_prompt_user(UPROMPT_TSTSH5, working_buffer);
        goto return_error;
}
/* This must be called before calling the later routines */
if (!pc_get_media_parms(path, &geometry))
{
        /* "Format: get media geometry failed. Press return" */
        rtfs_print_prompt_user(UPROMPT_TSTSH6, working_buffer);
        goto return_error;
}

/* Call the low level media format. Do not do this if formatting a
volume that is the second partition on the drive */
/* "Format: Press Y to format media " */
rtfs_print_prompt_user(UPROMPT_TSTSH7, buf);
if (tstsh_is_yes(buf))
{
        RTFS_PRINT_STRING_1(USTRING_TSTSH_67, PRFLG_NL); /* "Calling media format" */
        if (!pc_format_media(path, &geometry))
        {
                DISPLAY_ERRNO("pc_format_media")
                /* "Format: Media format failed. Press return" */
                rtfs_print_prompt_user(UPROMPT_TSTSH8, working_buffer);
                goto return_error;
        }
}
/* Partition the drive if it needs it */
if ( pdr->drive_flags&DRIVE_FLAGS_PARTITIONED )
{
        /* "Format: Press Y to partition media " */
        rtfs_print_prompt_user(UPROMPT_TSTSH9, buf);
        if (tstsh_is_yes(buf))
        {
                if (geometry.dev_geometry_lbas)
                {
                        do
                        {
                                /* "Format: Press Y to USE LBA formatting, N to use CHS " */
                                rtfs_print_prompt_user(UPROMPT_TSTSH10, buf);
                        }
                        while (!tstsh_is_yes(buf) && !tstsh_is_no(buf));
                        if (tstsh_is_no(buf))
                        {
                                geometry.dev_geometry_lbas = 0;
                        }
                }

                if (geometry.dev_geometry_lbas)
                {
                        RTFS_PRINT_STRING_1(USTRING_TSTSH_68, 0); /* "The drive is contains this many logical blocks " */
                        RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_lbas, PRFLG_NL);
                        /* "Format: Select the number of lbas for the first partition :" */
                        rtfs_print_prompt_user(UPROMPT_TSTSH11, working_buffer);
                        partition_list[0] = (dword)rtfs_atol(working_buffer);
                        partition_list[1] = 0;

                        if (partition_list[0] != geometry.dev_geometry_lbas)
                        {
                                RTFS_PRINT_STRING_1(USTRING_TSTSH_69, 0); /* "This many logical blocks remain" */
                                RTFS_PRINT_LONG_1((dword) (geometry.dev_geometry_lbas - partition_list[0]), PRFLG_NL);
                                /* "Format: Select the number of lbas for the second partition :" */
                                rtfs_print_prompt_user(UPROMPT_TSTSH12, working_buffer);
                                partition_list[1] = (dword)rtfs_atol(working_buffer);
                                partition_list[2] = 0;
                        }
                        if ((partition_list[0] == 0) || ((dword)(partition_list[0]+partition_list[1]) > geometry.dev_geometry_lbas))
```

```
                {
                    /* "Format: Bad input for partition values. Press return" */
                    rtfs_print_prompt_user(UPROMPT_TSTSH13, working_buffer);
                    goto return_error;
                }
            }
            else
            {
                RTFS_PRINT_STRING_1(USTRING_TSTSH_70, 0); /* "The drive contains this many cylinders" */
                RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_cylinders, PRFLG_NL);
                /* "Format: Select the number of cyls for the first partition :" */
                rtfs_print_prompt_user(UPROMPT_TSTSH14, working_buffer);
                partition_list[0] = (word)rtfs_atoi(working_buffer);
                partition_list[1] = 0;

                if (partition_list[0] != geometry.dev_geometry_cylinders)
                {
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_71, 0); /* "There are this many cylinders remainng" */
                    RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_cylinders - partition_list[0], PRFLG_NL);
                    /* "Format: Select the number of cyls for the second partition :" */
                    rtfs_print_prompt_user(UPROMPT_TSTSH15, working_buffer);
                    partition_list[1] = (dword)rtfs_atol(working_buffer);
                    partition_list[2] = 0;
                }
                if ((partition_list[0] == 0) || ((dword)(partition_list[0]+partition_list[1]) > geometry.dev_geometry_cylinders))
                {
                    /* "Format: Bad input for partition values. Press return" */
                    rtfs_print_prompt_user(UPROMPT_TSTSH13, working_buffer);
                    goto return_error;
                }
            }
            if (!pc_partition_media(path, &geometry, &partition_list[0]))
            {
                /* "Format: Media partition failed. Press return" */
                rtfs_print_prompt_user(UPROMPT_TSTSH16, working_buffer);
                goto return_error;
            }
        }
    }

    /* Put the DOS format */
    /* "Format: Press Y to format the volume " */
    rtfs_print_prompt_user(UPROMPT_TSTSH17, buf);
    if (tstsh_is_yes(buf))
    {
        if (!pc_format_volume(path, &geometry))
        {
            /* "Format: Format volume failed. Press return" */
            rtfs_print_prompt_user(UPROMPT_TSTSH18, working_buffer);
            goto return_error;
        }
    }
    OS_RELEASE_LOGDRIVE(driveno)
    return (0);
    return_error:
    OS_RELEASE_LOGDRIVE(driveno)
    return(-1);
    }
```

```
int doformat(int agc, byte **agv)
{
byte buf[10], working_buffer[100], byte path[10];
```

**pc_cluster_size**

Return a drive's cluster size

**Summary:**

int pc_cluster_size(byte *drive)

**Description:**

This function will return the cluster size of the mounted device named in the argument.

**Returns:**

The cluster size, otherwise 0 if the device is not mounted.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Drive name is invalid |

**See Also:**

po_extend_file

**Example:**

Given a byte count, calculate by rounding up how many clusters to extend a file by and then extend the file.

**int cluster_size;**

int n_clusters;

cluster_size = pc_cluster_size("C:");

n_clusters = (n_to_write + cluster_size-1)/cluster_size;

po_extend_file(fd, n_clusters, PC_FIRST_FIT);

**pc_get_file_extents**

Get the list of block segments that make up a file

**Summary:**

**#include <ertfs.h>**

int pc_get_file_extents(PCFD fd, int infolistsize, FILESEGINFO *plist, BOOLEAN raw)

Where FILESEGINFO is a structure defined as:

```
typedef struct fileseginfo {
    long block; Block number of the current extent
    long nblocks; Number of blocks in the extent
} FILESEGINFO;
```

And **infolistsiz**e is the number of elements in the storage pointed to by plist.

If raw is **TRUE**, the blocks are reported as block offsets from the physical base of the drive, otherwise the block offset origin is the beginning of the partition. Set raw to **TRUE** if you will be using the resultant list to set up DMA transfers to or from the disk.

**Description:**

This routine traverses the cluster chain of the open file **fd**. It logs into the list at plist and gets the block location and length in blocks of each segment of the file.

The block numbers and block length information can then be used to read and write the file directly using **pc_raw_read** and **pc_raw_write** or the information may be used to set up DMA transfers to or from the raw block locations. If the file contains more extents than will fit in plist as indicated by infolistsize, then the list is not updated beyond infolistsize elements but the count is updated and returned so the list size may be adjusted and the routine may be called again.

**Returns:**

Returns the number of extents in the file or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |
| PEINVALIDPARM | - | Invalid or inconsistent arguments |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**pc_get_free_list**

Get a list free cluster segments on the drive

**Summary:**

**#include <ertfs.h>**

int pc_get_free_list(byte *drivename, int listsize, FREELISTINFO *plist, long threshhold)

Where drivename is a valid drive specifier for example “C:” An empty string “” denotes the current working drive.

FREELISTINFO is a structure defined as:

```
typedef struct freelistinfo {
    CLUSTERTYPE cluster;    Cluster where the free region starts
    long nclusters;         Number of free clusters the free segment
} FREELISTINFO;
```

Listsize is the number of elements in the storage pointed to by plist. Threshhold is the smallest contiguous free region to report. This is provided to allow the caller to exclude free chains that are too small to be interesting. Setting this to a higher value also reduces the number of entries in plist that will be used up. The value of threshold must be at least 1. If it is one, then every free cluster segment is reported.

**Description:**

This routine traverses the file allocation table of the drive. It places in the results structure the starting point and size of each free segment. The free list information may then be used by **po_extend_file** to allocate specific clusters for specific files.

If the FAT contains more free extents than will fit in plist as indicated by listsize, the list is not updated beyond listsize elements but the count is updated and returned so that the list size may be adjusted and the routine may be called again.

**Returns:**

Returns the number of free segments or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |
| PEINVALIDPARMS | - | Invalid or inconsistent arguments |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**pc_raw_read**

Read raw blocks directly from a disk

**Summary:**

**#include <ertfs.h>**

int pc_raw_read(int driveno,  byte *buf, long blockno, int nblocks, BOOLEAN raw_io)

**Description:**

Attempt to read nblocks blocks starting at blockno. If **raw_io** is **TRUE**, then blockno is the offset from the beginning of the disk itself. If **raw_io** is **FALSE**, then blockno is the offset from the beginning of the partition.

This routine may be used in conjunction with **pc_get_file_extents** to find and read blocks without the additional overhead incurred when calling **po_read**.

The maximum allowable value for **nblocks** is 128.

*Note: It is possible to read any range of blocks in the* disk.

**Returns:**

Returns 0 if the read succeeded or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |
| PEINVALIDPARMS | - | Invalid or inconsistent arguments |
| PEIOERRORREAD | - | The read operation failed |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**pc_raw_write()**

Write blocks directly to a disk

**Summary:**

**#include <ertfs.h>**

int pc_raw_write(int driveno,  byte *buf, long blockno, int nblocks, BOOLEAN raw_io)

**Description:**

Attempt to write nblocks blocks starting at blockno. If **raw_io** is **TRUE**, then blockno is the offset from the beginning of the disk itself. If **raw_io** is **FALSE**, then blockno is the offset from the beginning of the partition.

This routine may be used in conjunction with **pc_get_file_extents** to find and read blocks without the additional overhead incurred when calling **po_write**.

The maximum allowable value for nblocks is 128.

*Note: It is possible to write any range of blocks in the disk.*

**Returns:**

Returns 0 if the read succeeded or -1 on error.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEINVALIDDRIVEID | - | Driveno is incorrect |
| PEINVALIDPARMS | - | Invalid or inconsistent arguments |
| PEIOERRORWRITE | - | The read operation failed |
| An ERTFS system error | - | See Appendix J for a description of system errors |

**po_extend_file()**

Extend a file by contiguous clusters.

**Summary:**

BOOLEAN po_extend_file(PCFD fd, unsigned long n_bytes, unsigned long *new_bytes, long start_cluster, int method)

**Description:**

Given a file descriptor, **n_bytes** bytes and method, extend the file and update the file size. If the file can be extended by **n_bytes** contiguous bytes, it will be done. Otherwise, if there is no contiguous region left on the disk that can contain **n_bytes**, the routine DOES NOT extend the file but does return the length of the next largest contiguous region that is available.

With this scheme, an application can request to extend the file by a given contiguous amount. If that is not possible, the routine will return the largest contiguous region available. The application can then decide if it wishes to use this region. Please read the notes below for an important discussion about the limitations of this routine.

The special method **PC_FIXED_FIT** may be used to extend the file beginning at a specific cluster. With this method, start_ cluster must be provided. It is used as the starting point for the allocation. With this method it is possible to precisely assign locations on the disk to file section. This may be used for example to create interleaved files where several files share a disk segment with interleaving clusters.:

**PC_FIRST_FIT** - The first chain in which the extension will fit

**PC_BEST_FIT** - The smallest chain in which the extension will fit

**PC_WORST_FIT** - The largest chain in which the extension will fit

**PC_FIXED_FIT** - Extend **n_clusters** from start cluster

***Please note the following issues and limitations.***

**PC_FIRST_FIT** is significantly faster than the others.

If the current end of file is not on a cluster boundary, the region to be tested will start at the cluster immediately following the last cluster in the file and the routine will allocate from the segment that starts with that cluster or it will return the number of contiguous bytes available starting at that cluster.

If possible you should allocate space in contiguous regions that are a multiple of the drive's cluster size.

If the **PC_FIXED_FIT** option is selected, a start cluster must be supplied, **n_bytes** must be an even multiple of cluster size and the file to extend must be either zero sized or the end of file must be on a cluster boundary.

*Note: If the requested file length exceeds RTFS_MAX_FILE_SIZE, po_extend_file does not extend the file, but rather fails and sets errno to PETOOLARGE. RTFS_MAX_FILE_SIZE defaults to 0xffffffff, but may be reduced by chagning the setting in rtfsconf.h*

**Returns:**

FALSE if an error occured.
TRUE if an error did not occur.

Returns n_bytes in *new_bytes if the file was extended. Otherwise it returns the largest contiguous chain of bytes available in *new_bytes. If it n_bytes is not returned the files was not extended.

errno is set to one of the following:

| | | |
|---|---|---|
| 0 | - | No error |
| PEBADF | - | Invalid file descriptor |
| PECLOSED | - | Invalid file descriptor because a removal or media failure asynchronously closed the volume. **po_close** must be called to clear this condition. |
| PEACCES | - | File is read only |

**pc_regression_test**

ERTFS File system port exercise and test routine

**Summary:**

**#include <ertfs.h>**

BOOLEAN pc_regression_test(byte *driveid, BOOLEAN do_clean)

**Description:**

This routine stress tests ERTFS, exercises most API calls and tests for memory leaks. It runs in a loop which creates a directory and then creates N subdirectories below that. Finally, the inner loop creates NUSERFILES files, writes to them, reads from them, seeks, truncates, closes, renames, and deletes them. Along the way it checks the set current working directory and get working directory calls. When the inner loop exits it checks to be sure no free space was lost.

There are a few functions that do not get tested, they are: **pc_gfirst**, **pc_gnext**, **pc_gdone**. Note that all modes of **po_open** and **po_lseek** are tested.

These are options that may be modified in the source code.

| | | |
|---|---|---|
| USEPRINTF | - | Set this to zero to run completely quiet. If this is done, you should set a break point in **regress_error** to catch errors. |
| test_dir[] | - | The directory where the test will occur |
| INNERLOOP | - | The number of times we run the inner loop |
| OUTERLOOP | - | The number of times we run the outer loop |
| SUBDIRDEPTH | - | The depth of the tested subdirectories. |
| NSUBDIRS | - | The number of subdirectories to create at each, must be less then 26. Each one of these directories will have SUBDIRDEPTH subdirectories below it. |

**Other inputs:**

| | | |
|---|---|---|
| do_clean | - | If do_clean is TRUE the regression test executes multiple passes and deletes all files and subdirectories created in each pass. If do_clean is FALSE the regression test stops after one pass and leaves all files and directories present. |

**Returns:**

Returns **TRUE** if the test succeeded, otherwise **FALSE**. If you have console output, informational messages will be written to the screen.

**Example:**

```
main()
{
    pc_ertfs_init(); /* Don't forget to call the initialization code */
    If (!pc_regression_test("A:", TRUE);
        Printf("Test failed\n");
```

**tst_shell**

Interactive command Shell

**Summary:**

tstsh

**Description:**

This subroutine provides an interactive command shell for controlling ERTFS. It provides a handy method for testing and exercising your port of ERTFS and it may be used to maintain the file system on your target system. All commands are summarized in Appendix H: Command Shell Reference Guide.

*Note: The source code for this routine in apptstsh.c contains many examples of calling and using the ERTFS API.*

Example:

```
main()
{
    pc_ertfs_init();    /* Don't forget to call the initialization code */
    pc_tstsh();         /* Call the test shell. It will execute until the user types QUIT */
    exit(0);
}
```

## SECTION 29: ERTFS PRO API FUNCTIONS

The API functions documented in this section are include in the ERTFS-Pro package only. They are not included in the Basic package.

**pro_buffer_status**

**Function**:

Return disk and failsafe status.

**Summary**:

BOOLEAN pro_buffer_status(byte *drive_name, struct pro_buffer_stats *pstat)

**Description**:

This routine returns status information about Failsafe, the block buffer pools and the fat buffer pools.

Note: This function reports statistics about failsafe but it does not require failsafe to be active. It is still useful even if failsafe is not enabled.

The information is returned to the user in the structure of type pro_buffer_stats that must be passed to the routine.

```
struct pro_buffer_stats {
    int     failsafe_mode;
    dword   failsafe_blocks_used;
    dword   failsafe_blocks_free;
    dword   total_block_buffers;
    dword   block_buffers_pending;
    dword   block_buffers_available;
    dword   block_buffers_low;
    dword   block_buffers_fail;
    dword   block_buffers_cache_hits;
    dword   block_buffers_cache_misses;
    dword   fat_buffer_primary_cache_hits;
    dword   fat_buffer_secondary_cache_hits;
    dword   fat_buffer_cache_loads;
    dword   fat_buffer_cache_swaps;
    dword   total_fat_buffers;
    dword   fat_buffers_pending;
    dword   fat_buffers_available;
    dword   fat_buffers_free;
    };
```

If the routine succeeds, it will return 0 and the following fields in pstat will be populated:

**failsafe_mode** - Failsafe operating mode.

0 - Failsafe not initialized

1 - Failsafe is initialized but journaling is not active

**failsafe_blocks_used** - If FailSafe journaling is enabled this value contains the number of bocks currently consumed in the FailSafe file.

**failsafe_blocks_free** - If FailSafe journaling is enabled this value contains the number of blocks still available in the FailSafe file.

**total_block_buffers** - Total number of directory buffers available. This may be the number of blocks in the system buffer pool or the number of blocks in the drive specific buffer pool if one was established with pro_assign_buffers.

**block_buffers_pending** - Total number of directory blocks scheduled to write but not yet written .This value is always zero.

**block_buffers_available** - Number of directory blocks still available to hold pending writes. This value is always equal to

total_block_buffers.

**block_buffers_cache_hits** - Number of block reads so far that were in the cache when a read request was made.

**block_buffers_cache_misses** - Number of block reads so far that were not in the cache when a read request was made.

**block_buffers_low** - The low water mark or lowest number of block buffers that have been available for allocation so far.

**block_buffers_fail** - The number of block allocation failures that have occurred due to insufficient buffer pool space.

**fat_buffer_primary_cache_hits** - Number of fat block accesses so far that were in the fat primary cache when the request was made.

**fat_buffer_secondary_cache_hits** - Number of fat block accesses so far that were in the fat secondary cache when the request was made.

**fat_buffer_cache_misses** - Number of fat block accesses so far that were not in the fat secondary cache when the request was made.

**total_fat_buffers** - Total number of FAT directory buffers for this drive. This value is determined by the value assigned to prtfs_cfg->cfg_FAT_BUFFER_SIZE[driveno] in **apiconfig.c**

**fat_buffers_pending** - Total number of FAT blocks scheduled to write but not yet written.

**fat_buffers_available** - Number of FAT blocks still available to store pending writes.

**fat_buffers_free** - Number of FAT buffer blocks that have never been used. If this value is non-zero it means no fat swapping has occurred.

**fat_buffer_cache_loads** - Number of FAT block reads so far that were not in the cache when the request was made.

**fat_buffer_cache_swaps** - Number of FAT block swaps so far. Blocks that were written because they contained changed data but the buffer was required in order to load another block.

**Returns:**

TRUE - Success

FALSE - Failure

If FALSE is return errno will be set to one of the following.

PEINVALIDDRIVEID -Drive argument invalid

**pro_assign_buffer_pool**

**Function**:

Assign a private buffer pool to a drive.

**Summary**:

```
BOOLEAN pro_assign_buffer_pool(byte *drive_name,
                int     block_hashtable_size,
                BLKBUFF **block_hash_table,
                int     block_buffer_pool_size,
                BLKBUFF *block_buffer_pool_data)
```

**Description**:

This routine provides a way to assign a private block buffer pool to a named drive. Normally block buffer space is shared among all drives in the system, but this function allows the caller to assign block buffers that are private to the specified drive. This function may improve performance by reducing the amount of buffer swapping that occurs when multiple drives compete for buffers in the common pool

**Inputs:**

**Drive_name** - Null terminated drive designator, for example “A:”

**Block_buffer_context** - User supplied memory for the block buffer context block that will be assigned to the drive to be used as a private buffer pool management structure. This must be a pointer to a structure of type BLKBUFFCNTXT that must remain valid for the whole session and may not be deallocated. If block_buffer_context is 0, the drive

**block_hashtable_size** - You must set this to the size of the block hash table. Each entry in the table takes up 4 bytes. This value must be a power of two **Drive_name** - Null terminated drive designator, for example “A:”

**block_hash_table** - User supplied memory for the block hash table. This must be a pointer to an array of data of type BLKBUF *, containing block_hashtable_size elements. It must remain valid for the whole session and must not be deallocated.

**block_buffer_pool_size** -You must set this to the size of the block buffer pool. Each entry in buffer pool requires approximately 540 bytes.

**block_buffer_pool_data** -User supplied memory for the block buffer pool.This must be a pointer to an array of data of type BLKBUF, containing block_buffer_pool_size elements. It must remain valid for the whole session and must not be deallocated.

***Note: Please see section 3 of the ERTFS User’s Manual for a more complete discussion of these elements.***

**Returns:**

TRUE - Success

FALSE - Failure

If FALSE is return errno will be set to one of the following.

PEINVALIDDRIVEID -Drive argument invalid

PEINVALIDPARMS - Invalid parameters.

PEINVALIDDRIVEID - Invalid drive

An ERTFS system error

## SECTION 30: FAILSAFE OPERATING MODE

### INTRODUCTION

Failsafe mode provides a means to eliminate the risk of file system corruption that results from unexpected power interruptions and media removal events.

Failsafe works by journalling all changes to the File Allocation Table (FAT) and directory structure until a function is called to commit those blocks to the disk. FailSafe may be configured to automatically commit the blocks before returning from the API, or it may be configured to hold off until a commit API function is called explicitly to write the directory and FAT blocks from the journal to the disk.

If the commit function runs to completion without a media removal or a power down, then the disk is in sync and correct. If the commit function is interrupted, some level of corruption, lost cluster chains, and incorrect file sizes will exist.

Corruption caused by an interrupted buffer commit may be corrected by executing a procedure to use information stored in the journal file to complete a previously interrupted commit function. After the restore operation is complete the disk state is the same as it would have been had the commit function completed.

The journal file contains state information that indicates if a restore operation is needed at power up. FailSafe may be configured either to require the application to manually check the status and restore, or to automatically detect commit failures and restore the file system when the disk is remounted.

### FAILSAFE FEATURES

**Automatic Operation** – FailSafe may be configured so that FailSafe operation is completely transparent to the application layer. In this mode ERTFS's init function, **pc_ertfs_init()**, automatically configures FailSafe mode at power on. FailSafe may be configured to automatically restore the volume from the journal file when the volume is mounted and to automatically commit blocks with each API call. In this mode the applications code is not required to make any explicit calls in order to achieve the benefits of FailSafe.

**Manual Operation** – FailSafe may be configured so that one or more of the operations described in the previous section may be invoked manually from the application layer. A mix of automatic and manual operations may be created, for example the device may be configured for FailSafe operation at boot time but the application layer may wish not to use the auto-commit feature and execute the commit operation manually from the API.

**Programmers Interface** – An API is provided to perform operations such as, initialize FailSafe, suspend FailSafe , resume FailSafe, suspend auto-commit mode, and resume auto-commit mode. Other API calls provide a means to check ERTFS and Failsafe buffer usage and to perform manual commits, restores and journal file queries.

**Reliability Features** – FailSafe restores checksums index information in the journal file to detect if the file was corrupted outside of the ERTFS. It also checks the volume free space versus information stored in the index to detect if the volume was modified on another system that is not running FailSafe. The FailSafe auto restore feature may be configured to ignore these errors when they occur and skip the restore function or it may be instructed to halt the mount process and report the problem to the application layer.

**Application Specific Journal File** – By default the journal is maintained in a special file that resides in the volume. For some applications it may be preferable to put the journal file into non-volatile system ram or flash memory. To make this possible the journal file size is user configurable and a journal file access API is defined which may be used to override the standard method.

### FAILSAFE RESOURCE REQUIREMENTS

When ERTFS is running in FailSafe mode, some additional media, i/o system, memory and CPU resources are required. Considering the benefits of using FailSafe the additional resources are minimal. The following sections describe FailSafe's impact on each of these system resources.

### JOURNAL FILE RESOURCE REQUIREMENTS

FailSafe requires a journal file to operate. By default, the journal file is placed on a disk when it is auto-mounted and is set to the size of the disk's File Allocation Table plus 64K. These default settings require a very small percentage of the disk space but they are still very conservative and provide more than enough caching even if long periods elapse between commit calls. The FailSafe file size is user configurable.

### FAILSAFE'S EFFECT ON CPU AND IO UTILIZATION

FailSafe's impact on API call performance is negligible. During API calls the same number of disk operations occur whether a volume is mounted in FailSafe mode or not. Internally the failsafe layer relies on some smart hashing functions and caching to execute quickly, imposing effectively zero cost to the API calls. The FailSafe commit function does add some additional over-

head. The Failsafe commit function always writes at least two blocks to the journal file and then a variable number of blocks to the volume. The number of additional block writes depends on the number of API calls made prior to the commit call, and on their complexity. Typically one or two additional block writes are required. In rare cases, if ERTFS's buffering space is limited or a complex set of operations is performed, a variable number of reads of the journal file may be required.

**RAM RESOURCE REQUIREMENTS**

FailSafe requires the user to provide, at run time, one FailSafe context structure for each drive using FailSafe and optionally one pool of block mapping buffers per drive using FailSafe. The default configuration consumes approximately 4 K, but it may be reduced to as low as 1200 bytes.

The default size of the FailSafe context structure is approximately 3 K. A compile time constant may be adjusted to reduce the structure to its minimum size of approximately 1200 bytes.

The block mapping pool by default consumes approximately 800 bytes, but this may be reduced to as little as zero bytes if desired.

**STRATEGIES FOR USING FAILSAFE**

Before using FailSafe you must decide how it will be used and configured. The following topics should be considered.

**Initialization policy** – Failsafe must be initialized before it is used. To initialize it you may either explicitly call pro_failsafe_init() from the API layer once ERTFS is running or you may configure the ERTFS initialization sequence to automatically call pro_failsafe_auto_init(). While the results of these two operations are the same they each have their place. Pro_failsafe_auto_init() may be used along with some other features to provide FailSafe functionality while being completely transparent to the application, requiring no explicit API calls. Pro_failfafe_init() may be used instead if the application wishes to determine specific conditions under which FailSafe will be enabled. For more information on the initialization process please see the following sections on initializing and configuring FailSafe and the manual pages for pro_failsafe_init() and pro_failsafe_auto_init()

**Restore policy** – FailSafe may be configured to transparently restore the volume from the FailSafe file when a volume is mounted or it may be configured to refuse to mount the disk if a restore is needed and require the user to call pro_failsafe_restore() from the API layer to restore and remount the volume. As a third option It may be configured to automatically restore and mount the volume if the journal file is okay but fail to restore and mount if it detects that the journal file was corrupted by an external process. In this case the applications layer is alerted that a journal file error is present and the user must call pro_failsafe_restore() from the API layer to clear the journal file and remount the volume.

**Commit policy** – After a successful commit operation the disk volume is guaranteed to be in sync with the application's view of the disk volume. FailSafe may be configured to run in auto-commit mode or manual commit mode. In auto commit mode ERTFS automatically performs the commit function before it returns from the API and no explicit actions are needed. Alternatively it may be configured to run in manual commit mode. In this mode the user must call pro_failsafe_commit() to commit the changes to the volume structure. The auto commit method is simpler and guarantees that when ERTFS API calls return the intended changes to the volume structure are committed to disk.

The manual commit method may be used to improve performance slightly by holding off the commit operation until a group of API call operations has been completed. The manual method also provides a way to group multiple API calls into one apparently atomic operation. By this method the user can guarantee that the volume structure contains either none of the changes or all of the changes, but it will never contain only some of them. If system power is interrupted before the commit function is started then all actions are lost. If it is interrupted while the commit function is executing, when the system is restarted, the restore function will complete the commit process and the volume will then reflect all changes made by all of the API calls.

**Recovery policy** – There are several error conditions that FailSafe restore checks for before restoring the volume from the FailSafe file. If FailSafe restore detects that the file's internal checksum value is incorrect or that the volume's freespace has changed since the previous commit was made it assumes that the FailSafe file and the volume structure are out of sync and it does not perform the restore. If this condition occurs while auto-restore is enabled the default policy is to abort the mount and set errno to reflect that this error has occurred. The application must then call pro_failsafe_restore() to clear the error condition and remount the volume. This process serves to alert the application later that the volume contains a corrupt or out of synch FailSafe file. If this alert is not required FailSafe may be configured in so called to auto-recover mode, which automatically clears the FailSafe file if one of these conditions is detected.

**CONFIGURING ERTFS TO INCLUDE FAILSAFE**

To use FailSafe mode the constant INCLUDE_FAILSAFE_CODE must be set to 1 in rtfsconf.h and all of ERTFS must be recompiled.

**CONFIGURING FAILSAFE AT COMPILE TIME**

Two compile time constants may be modified. CFG_NUM_JOURNAL_BLOCKS controls the size of the journal file and

CFG_NUM_INDEX_BUFFERS controls the amount of ram buffering available for Failsafe operation. The constants reside in and may be modified in the file prfs.h, the default values for these constants are very conservative and they should not require changing under most circumstances.

**CFG_NUM_JOURNAL_BLOCKS** - This compile time constant determines the number of 512 byte blocks added to failsafe file for journaling directory blocks. By default the journal file is made large enough to hold the whole file allocation table plus CFG_NUM_JOURNAL_BLOCKS additional blocks. The default value is 128.This is a very small percentage of most disks but it should be adequate for almost any imaginable usage. It may be reduced to as little as 1 and FailSafe will still work fine under most conditions. The user may override these default settings at run time by providing a non-zero value in the journal_size field of the FailSafe context block before calling pro_failsafe_init(). If the journal file is too small it fills up and API calls fail.

**CFG_NUM_INDEX_BUFFERS** - This compile time constant determines the number of 512 byte buffers to reserve for buffering FailSafe index pages. It must be at least one and it should be set to at least two unless ram resources are very precious. The default value is four, which should be large enough for almost any application. If CFG_NUM_INDEX_BUFFERS is too small it has a negative effect on performance.

**CONFIGURING FAILSAFE AT RUN TIME**

Several features of Failsafe may be configured at run time. These include the size of the block replacement cache and the desired restore, recover and commit policies. The journal file size may also be set at run time to override the compile time defaults. Please see the manual page for the function pro_failsafe_init() for specific details on these run time values.

**INITIALIZING FAILSAFE**

FailSafe mode must be initialized for all drives on which FailSafe will operate. The initialization sequence may occur in one of two ways, either transparently as part of ERTFS's initialization process or it may be performed explicitly by calling the pro_failsafe_init() API routine. The former method is simpler and preferable under most circumstances. For details on these two methods please consult the manual pages

**ADDITIONAL ERRNO HANDLING WHEN USING FAILSAFE**

When FailSafe mode is enabled for a drive all ERTFS API calls that access that drive use FailSafe services and thus additional API calls errno values are possible for API call failures. Six additional errno values are defined when FailSafe is being used: PEFSCREATE, PEFSRESTORENEEDED, PEFSRESTOREERROR, PEIOERRORWRITEJOURNAL, PEIOERRORREADJOURNAL and PEJOURNALFULL.

**PEFSCREATE** - The mount procedure failed in its attempt to create a journal file. The disk may be read only or perhaps it is full. To clear the error the application may call pro_failsafe_shutdown() to disable FailSafe and continue. If the application wishes to resume FailSafe operation but the disk is full it may clear the error by using pro_failsafe_shutdown() to disable FailSafe, free up some space on the disk with FailSafe mode disabled, and then call pro_failsafe_init() to restart failsafe. This error is possible for any ERTFS API call that accesses a volume.

**PEFSRESTORENEEDED** - Autorestore is not enabled and the mount procedure detected that the volume should be restored from the journal file. The application may clear this error by calling pro_failsafe_restore with instructions to either restore the disk or clear the journal file. This error is possible for any ERTFS API call that accesses a volume.

**PEFSRESTOREERROR** - Autorestore and Autorestore are not both enabled and the mount procedure detected that the journal file is invlid. The application may query the nature of the error by calling pro_failsafe_restore with instructions to return status only. It may clear the error by calling pro_failsafe_restore with instructions to clear the journal file. This error is possible for any ERTFS API call that accesses a volume.

**PEIOERRORWRITEJOURNAL** - An error occured writing the journal file while executing automount or writing a FAT or directory block or autocommiting the API call. This error is caused by an IO error to the journal, possibly due to it being write protected. If the application wishes to resume with this media it should either disable FailSafe or clear the cause of the IO error and retry the API call. This error is possible for any ERTFS API call that accesses a volume.

**PEIOERRORREADJOURNAL** - An error occured reading the journal file while executing automount or writing a FAT or directory block or autocommiting the API call. This error is caused by an IO error to the journal, If the application wishes to resume with this media it should clear the cause of the IO error and retry the API call. This error is possible for any ERTFS API call that accesses a volume.

**PEJOURNALFULL** - An error occured when an ERTFS API call attempted to journal a directory or FAT block but the journal file was full. If this error occurs the changes made by all ERTFS API calls since the last succesful FailSafe commit operation are lost. The application should not call pro_failsafe_commmit. It should immediately call pro_failsafe_restore with instructions to clear the journal file. If the journal file is full it means that application has modified more FAT and directory blocks then will fit in the journal file. This error is most likely the result of misuse of the manual mode of FailSafe by performing too many API calls without any intervening calls to pro_failsafe_commit(). If manual mode FailSafe is being used appropriately or autocommit mode is enabled, the only single API calls that could possibly cause this error

are pc_unlink of a very huge file, pc_rmdir of a very huge directory or pc_deltree of a very large subtree. If the call fails because the journal is full the disk will not be changed by the call and when the error is cleared and the disk is remounted it will appear that the call had not been made at all. If an application is unusual enough to cause a journal full error it should be modified so that it calls pro_failsafe_commit more often or it should request a larger journal file when FailSafe is initialized. This error is possible for any ERTFS API call that writes to a volume.

**ERTFS PRO FAILSAFE API FUNCTIONS**

**pro_failsafe_init**

**Function:**

Initiate a failsafe session

**Summary:**

BOOLEAN pro_failsafe_init(byte *drive_name, FAILSAFECONTEXT *pfscntxt, int configuration_flags)

**Description**:

This routine checks the user supplied parameters, initializes the failsafe context structure, makes sure that buffers are flushed and places the drive in failsafe mode. Once the device is in failsafe mode, directory and FAT block writes will be held in a journal file until the commit operation is executed. This routine may be used to initialize failsafe from the API layer. The routine named pro_failsafe_auto_init() should be used instead, if you wish to automatically enable FailSafe mode at boot time.

**Inputs:**

**drive_name** - Null terminated drive designator, for example “A:”.

**pfscntx**t - The address of a block of data that will be used as a context block for failsafe. This block must remain valid for the whole session and must not be de-allocated. Pfscntxt must be zero filled before **pro_failsafe_init(**) is called.

**configuration_flags** - You can optionally enable these features by or-ing these values into the configuration_flags argument.

FS_MODE_AUTORESTORE - Instruct ERTFS-Pro, when it mounts a disk, to automatically check the status of the Journal file, and, if the file indicates a restore is needed, perform the restore. If the restore fails because of an I/O error or because of a corrupted journal file, then errno is set to PEFSRESTOREERROR and the disk mount fails.

FS_MODE_AUTORECOVER - FS_MODE_AUTORECOVER may only be used in conjunction with FS_MODE_AUTORESTORE. In FS_MODE_AUTORECOVER mode, if the auto restore operation fails because of an I/O error or a corrupted journal file, the error is ignored, the restore is not executed and the mount process continues.

FS_MODE_AUTOCOMMIT - If AUTOCOMMIT is enabled the FailSafe commit operation will be performed automatically by ERTFS at the completion of each API call. With AUTOCOMMIT enabled the FailSafe operation is transparent to the user and it is never necessary to call pro_failsafe_commit().

**Additional Inputs:**

Several fields in the FailSafe context block may be initialized before pro_failsafe_init() is called. user_journal_size may be set under certain special circumstances. blockmap_freelist and blockmap_size should be initialized under most circumstances.

FailSafe will utilize an optional block map cache if you provide it with the necessary ram resources. Even though FailSafe will execute properly if no block mapping cache is provided, its performance will be significantly diminished. It is therefore highly recommended that you enable block mapping. FailSafe performance will be reduced drastically if the number of blocks written to the FailSafe file exceeds the number of block map elements. Each caching structure requires approximately 32 bytes. In the examples provided with ERTFS we provide 128 mapping structures. To enable block map buffering you must assign values to blockmap_freelist and blockmap_size.

**user_journal_size** - This field in the context block may be assigned with a value that prescribes the size of the FailSafe file in blocks. If this field is left at zero the FailSafe file will be sized according to the algorithm described in the compile time configuration section.

**blockmap_freelist** - This field in the context block may be assigned a pointer to an array of structures of type FSBLOCKMAP. for example: context.blockmap_freelist = &blockmap_array[0];

**blockmap_size** - This field must be filled in with the number of elements contained in the array assigned to blockmap_freelist.

**Returns:**

TRUE - Success

FALSE - Error

If it returns FALSE, errno will be set to one of the following:

PEINVALIDPARMS - Invalid parameters; either no context block was passed or the fields were not initialized correctly.

PEINVALIDDRIVEID - Invalid drive

An ERTFS system error perhaps caused if the FAT and buffer flush failed.

**pro_failsafe_auto_init**

**Function**

Automatically enable failsafe for a device at boot time

***You must modify this source code if you wish to auto-configure failsafe features for a given device.***

**Summary:**

BOOLEAN pro_failsafe_auto_init(DDRIVE *pdrive)

**Description**:

In pc_ertfs_init(), for a given device, if the flag value DRIVE_FLAGS_FAILSAFE is set in pdr->drive_flags field, then pro_failsafe_auto_init() is called to automatically enable Failsafe mode for that drive. When failsafe is initialized this way there is no need to call pro_failsafe_init() from the API layer.

***Note: pro_failsafe_auto_init() is implemented inside the file prapifs.c. To use this feature, ICLUDE_FAILSAFE_AUTO_INIT must be set to 1 in the source file and the source code for the routine should be modified to configure FailSafe to fit your needs. By default the INCLUDE_FAILSAFE_AUTO_INIT is set to zero.***

**Inputs:**

**Pdrive** - A pointer to the drive structure for the device.

*Please see the manual page for pro_failsafe_init() for a complete description of FailSafe initialization parameters*

**Returns:**

TRUE - Success

FALSE - Error. If pro_failsafe_auto_init() returns false the drive is marked invalid and all future volume mount attempts will fail.

**pro_failsafe_commit**

**Function:**

Commit failsafe buffers to disk

**Summary:**

BOOLEAN pro_failsafe_commit(byte *drive_name)

**Description:**

This routine commits changed FAT and block buffers to disk.

When **pro_failsafe_commit** completes successfully, all changes made to the volume structure since the previous successful call to **pro_failsafe_commit** are guaranteed to be committed to disk. If **pro_failsafe_commit** is interrupted, those FAT and directory changes made since the last call to **pro_failsafe_commit** may be partially committed to disk. This causes inconsistencies such as lost cluster chains and incorrect file lengths. In this case **pro_failsafe_restore** may be called to complete the commit process.

***Note: If FailSafe is initialized in auto-commit mode the commit function is called automatically and it is unnecessary to explicitly call pro_failsafe_commit.***

**pro_failsafe_commit** should be called whenever you wish to guarantee that the disk is up to date. Appropriate times to call **pro_failsafe_commit** include after a file is closed or flushed, after a file is removed or renamed or after directory has been created or removed. Typically **pro_failsafe_commit** needs to write only a few blocks to disk to complete successfully.

**Returns:**

TRUE- Success

FALSE - Failure

If FALSE is returned, errno will be set to one of the following.

| | |
|---|---|
| PEINVALIDDRIVEID | - Drive argument invalid |
| PENOINIT | - pro_failsafe_init must be called first |
| PEJOURNALOPENFAIL | - Journal file or vram init call failed |
| PEIOERRORREAD | - Error reading FAT or buffer area. |
| PEIOERRORWRITEJOURNAL | - Error writing journal file or NVRAM section |
| PEIOERRORWRITEFAT | - Error writing FAT area |
| PEIOERRORWRITEBLOCK | - Error writing directory area |

An ERTFS system error

**pro_failsafe_restore**

**Function:**

Clear the failsafe journal or check journal status or restore from the failsafe journal.

**Summary:**

int pro_failsafe_restore(byte *drname, FAILSAFECONTEXT *ctxt, BOOLEAN dorestore,BOOLEAN doclear)

**Description:**

**Clearing the Journal File**

In FailSafe mode, unless autorecover and autorestore modes are both enabled, and a journal file error is detected when the disk is mounted, the disk will not be accessible from the ERTFS API. All ERTFS API calls that try to act on the volume will fail and errno will be set to PEFSRESTOREERROR.

This check condition status will remain until either the error is cleared or FailSafe is shut down. Journal file errors are rare. They result from one of several causes. Either a version miss-match occurred between the Journal file and the current ERTFS version, or the Journal file was changed or corrupted by an external process, or the Journal file is valid and a restore is needed but FailSafe detects that the volume was modified by an external process not running FailSafe.

To learn more about the cause of the error you may call pro_failsafe_restore() with dorestore and doclear both set to FALSE. Pro_failsafe_restore() will then return FS_STATUS_BAD_JOURNAL, FS_STATUS_BAD_CHECKSUM, or FS_STATUS_OUT_OF_DATE.

To clear this condition call pro_failsafe_restore() with doclear set to TRUE.

**Providing a FailSafe context block for pro_failsafe_restore()**

The ctxt argument is provided so that pro_fasilsafe_restore() may be used even when FailSafe has not been initialized. If failsafe has been initialized a zero argument may be passed and pro_failsafe_restore will use the context block provided by pro_failsafe_inti(). Otherwize you must provide the address of a FailSafe context block structure in the ctxt argument. pro_failsafe_restore() will fail and return FS_STATUS_NO_INIT if no cointext block is available.

**Checking the status of the FailSafe file**

If doclear and dorestore are both FALSE then pro_failsafe_restore() will not attempt to restore or to clear the journal file. Instead it will return the status of the file, returning one of the codes listed below.

**Restoring the volume from the Journal file**

If doclear is FALSE and dorestore is TRUE and the contents of the journal file indicate that a restore is needed, pro_failsafe_restore() will restore the volume from the journal file. When the restore process completes the journal file is cleared. If the process is interrupted at any time before it completes the contents of the journal file remain valid and the restore process may be restarted when the system is restarted.

**Returns:**

| | |
|---|---|
| FS_STATUS_OK | - No restore required |
| FS_STATUS_NO_JOURNAL | - No journal file present |
| FS_STATUS_BAD_JOURNAL | - Journal present but has invalid fields |
| FS_STATUS_BAD_CHECKSUM | -Journal data doesn't match stored checksum |
| FS_STATUS_OUT_OF_DATE | - Journal is present but the FAT table has changed, perhaps on another computing device, since the journal file was last updated. |
| FS_STATUS_IO_ERROR | - I/O error during restore |
| FS_STATUS_RESTORED | - Restore was required and completed |
| FS_STATUS_MUST_RESTORE | -Disk must be restored from the journal file because flushing of FAT and disk blocks was interrupted. |
| FS_STATUS_NO_INIT | - FailSafe mode is not initialized for this device and the user did not provide an operating context block. |

**Example:**

```
FAILSAFECONTEXT ctx;
BOOLEAN clear;
Clear = FALSE;
 int s = pro_failsafe_restore("A:", &ctxt, FALSE, FALSE);
if (s==FS_STATUS_OK)                        printf("No restore required\n");
else if (s==FS_STATUS_RESTORED)             printf("No restore needed\n");
else if (s==FS_STATUS_NO_JOURNAL)           printf("No journal file present\n");

else if (s==FS_STATUS_BAD_JOURNAL)          {clear=TRUE;printf("Journal contains bad data\n");};
else if (s==FS_STATUS_BAD_CHECKSUM)         {clear=TRUE;printf("Journal contains bad data\n");};
else if (s==FS_STATUS_OUT_OF_DATE)          {clear=TRUE;printf("Journal out of sync\n");};

else if (s==FS_STATUS_IO_ERROR)             printf("IO error accessing journal\n");
else if (s==FS_STATUS_MUST_RESTORE)
{
        printf("Restoring disk from the journal file\n");
        if (pro_failsafe_restore("A:", &ctxt, TRUE, FALSE)==FS_STATUS_RESTORED)
                                printf("Restore was completed successfully\n");
}
if (clear)
    if (pro_failsafe_restore("A:", &ctxt, FALSE, TRUE)==FS_STATUS_OK)
                    printf("Journal was cleared successfully\n");
```

**pro_failsafe_shutdown**

**Function:**

Close or abort FailSafe mode

**Summary:**

BOOLEAN pro_failsafe_shutdown(byte *drive_name, BOOLEAN abort)

**Description:**

This routine allows the applications layer to shutdown a FailSafe session that was either auto-initialized at startup or was initialized by calling pro_failsafe_init(). After pro_failsafe_shutdown() has completed the FailSafe context block that was assigned to the drive may be reused.

If abort is TRUE then FailSafe mode is immediately disabled for the device and any currently uncommited operations are lost.

If abort is FALSE FailSafe will attempt to commit any currently journalled operations before disabling FailSafe. If the commit operation fails pro_failsafe_shutdown() will return FALSE, errno will be set to one of the values listed below and FailSafe will still be active.

After pro_failsafe_shutdown() has executed pro_failsafe_init() may be called again to re-initialize FailSafe.

**Returns:**

TRUE - Success

FALSE - Failure

If FALSE is returned, errno will be set to one of the following.

| | |
|---|---|
| PEINVALIDDRIVEID | - Drive argument invalid |
| PEJOURNALOPENFAIL | - Journal file or vram init call failed |
| PEIOERRORREAD | - Error reading FAT or buffer area. |
| PEIOERRORWRITEJOURNAL | - Error writing journal file or NVRAM section |
| PEIOERRORWRITEFAT | - Error writing FAT area |
| PEIOERRORWRITEBLOCK | - Error writing directory area |
| An ERTFS system error | |

**fs_test**

**Function:**

Test Failsafe mode

**Summary:**

BOOLEAN fs_test(byte *path)

**Description:**

Perform a series of tests on ERTFS-Pro to verify correct functioning of FailSafe mode.

**Returns:**

TRUE - if all test succeeded.

FALSE - A test failed. Review output or use a debugger to determine what failed.

***Note: fs_test() sends output to the console through the macro FSDEBUG()***

FSDEBUG is defined in the source file prfstest.c as:

```
#define FSDEBUG(X) printf("%s\n", X);
```

To run quietly replace this definition with the following:

```
#define FSDEBUG(X)
```

**The following tests are performed**:

**INDEX TEST** - This set of tests checks for correct functioning of the FailSafe indexing and caching functions. Over 50 tests are employed to verify correct functioning of the indexing and caching under varying normal conditions and error conditions.

**NVIO TEST** - This set of tests verifies correct creation and placement of the journal file when the disk is empty, when it is full and when it is fragmented.

**API TEST** - This series of tests verifies the correct functioning of ERTFS-Pro API calls when FailSafe is enabled.

**The following API calls are tested:** pc_mkdir, pc_rmdir, po_open, pc_unlink, pc_mv, pc_deltree, pc_set_attributes, po_extend_file, po_chsize, po_truncate, po_write, po_close

**The following tests are performed on each API call:**

- Test that the API call changes only the virtual view of the volume and not the volume itself when FailSafe is enabled but autocommit mode is not enabled.
- Test that the volume is automatically synchronized with the virtual volume view when autocommit mode is enabled.
- Test that the volume is synchronized with the virtual volume view after pro_failsafe_commit() is called when autocommit mode is not enabled,
- Test that the API call fails gracefully and sets errno to PEJOURNALFULL if the Journal File is full.

**RESTORE TEST** - This set of tests verifies the correct operation of the pro_failsafe_restore() function when it is asked to perform one of its functions, clearing the journal file, reporting the journal file status and restoring the volume from the journal file. This test also verifies correct functioning of FailSafe's auto restore feature.

**FSAPI TEST** - This set of tests verified the correct operation of all FailSafe API functions, including pro_failsafe_int(), pro_failsafe_commit(), pro_failsafe_shutdown() and pro_failsafe_restore().

**The following output is produced by a successful run of fs_test().**

```
INDEX TEST: Begin
INDEX TEST: Test mapping with cache > # journaled blocks
INDEX TEST: Filling Journal File in ascending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in descending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in random order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Testing Index Checksum Error
INDEX TEST: Testing Index Out of Date Error
INDEX TEST: Testing Journal File Signature Error
INDEX TEST: Testing Good Journal File
INDEX TEST: Test mapping with cache < # journaled blocks
INDEX TEST: Filling Journal File in ascending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in descending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in random order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Testing Index Checksum Error
INDEX TEST: Testing Index Out of Date Error
INDEX TEST: Testing Journal File Signature Error
INDEX TEST: Testing Good Journal File
INDEX TEST: Test mapping with no cache
INDEX TEST: Filling Journal File in ascending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in descending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in random order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Testing Index Checksum Error
INDEX TEST: Testing Index Out of Date Error
INDEX TEST: Testing Journal File Signature Error
INDEX TEST: Testing Good Journal File
INDEX TEST: Testing Journal File Full
INDEX TEST: Test mapping with cache and very large journal
INDEX TEST: Filling Journal File in ascending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in descending order
INDEX TEST: Testing Journal Block Search
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Filling Journal File in random order
INDEX TEST: Testing Journal Block Searh
```

```
INDEX TEST: Testing Sorted Journal Block List
INDEX TEST: Testing Index Checksum Error
INDEX TEST: Testing Index Out of Date Error
INDEX TEST: Testing Journal File Signature Error
INDEX TEST: Testing Good Journal File
INDEX TEST: Success
NVIO TEST: Begin
NVIO TEST: verify journal file placement
NVIO TEST: verify journal file placement with fragmentation
NVIO TEST: verify changing journal file size
NVIO TEST: test journal file create error with full disk
NVIO TEST: Success
API TEST: Begin
API TEST
API TEST: pc_mkdir
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: pc_rmdir
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: po_open
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: pc_unlink
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: pc_mv
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: pc_deltree
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: pc_set_attributes
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: po_extend_file
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
```

```
API TEST: po_chsize
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: po_truncate
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: po_write
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: po_close
API TEST:         test journalling
API TEST:         test autocommit
API TEST:         test manual commit
API TEST:         test with full Journal
API TEST: Success
API TEST: Success
RESTORE TEST: Begin
RESTORE TEST: test manual restore
RESTORE TEST: test manual restore with bad Journal file
RESTORE TEST: test auto restore with bad Journal file
RESTORE TEST: Test pro_failsafe_restore() clear function
RESTORE TEST: auto recover with bad Journal file
RESTORE TEST: test auto restore with good Journal file
RESTORE TEST: Success
FSAPI TEST: Begin
FSAPI TEST: pro_failsafe_init
FSAPI TEST: pro_failsafe_init test mount behavior
FSAPI TEST: pro_failsafe_init test bad inputs
FSAPI TEST: pro_failsafe_init user_journal_size > default
FSAPI TEST: pro_failsafe_init user_journal_size < default
FSAPI TEST: pro_failsafe_init test block map arguments
FSAPI TEST: pro_failsafe_init test FS_MODE_AUTOCOMMIT
FSAPI TEST: pro_failsafe_init test FS_MODE_AUTORESTORE
FSAPI TEST: pro_failsafe_init test FS_MODE_AUTORECOVER
FSAPI TEST: pro_failsafe_init Success
FSAPI TEST: pro_failsafe_commit
FSAPI TEST: pro_failsafe_commit test journal IO error handling
FSAPI TEST: pro_failsafe_commit test error handling
FSAPI TEST: pro_failsafe_commit test commit process
FSAPI TEST: pro_failsafe_commit Success
FSAPI TEST: pro_failsafe_restore
FSAPI TEST: pro_failsafe_restore test normal restore
FSAPI TEST: pro_failsafe_restore test journal data error handling
FSAPI TEST: pro_failsafe_restore test clear function
FSAPI TEST: pro_failsafe_restore Success
FSAPI TEST: Success
FAILSAFE TEST SUITE: Success
```

**CUSTOMIZING FAILSAFE**

In its default configuration failsafe uses a disk file named FAILSAFE in the root directory for its journal file. For some applications it may be better to store the failsafe journal in flash or non volatile ram. All failsafe journal accesses are segregated into four functions that are implemented in prfsnvio.c. If you wish to use flash or NV ram to store the journal file then these four functions must be modified.

| | |
|---|---|
| failsafe_create_nv_buffer | - Create the failsafe buffer. |
| failsafe_reopen_nv_buffer | - Re-open the failsafe buffer. |
| failsafe_write_nv_buffer | - Write a block to the failsafe buffer. |
| failsafe_read_nv_buffer | - Read a block from the failsafe buffer. |

Each of these functions is described below.

**failsafe_create_nv_buffer**

**Function:**

Create the failsafe buffer.

**Summary:**

BOOLEAN failsafe_create_nv_buffer (FAILSAFECONTEXT *pfscntxt)

**Description:**

This routine must create a failsafe buffer on the current disk or in system NV ram and return TRUE if successful, FALSE if it is unsuccessful. It may use the nv_buffer_handle field in the FAILSAFECONTEXT structure at pfscntxt to store a handle for later access

The size of the FailSafe file must be at least the following quantity (pfscntxt->num_remap_blocks + pfscntxt->num_index_blocks) in 512 byte blocks. These values are calculated internally by FailSafe before the context block is passed. If the user assigned a file size through the user_journal_size configuration parameter the quantity (pfscntxt->num_remap_blocks + pfscntxt->num_index_blocks) will be equal to the assigned value.

**failsafe_reopen_nv_buffer**

**Function:**

Re-open the failsafe buffer.

**Summary:**

BOOLEAN failsafe_reopen_nv_buffer (FAILSAFECONTEXT *pfscntxt)

**Description:**

This routine must check for the existence of a failsafe buffer on the current disk or in system non volatile RAM and return TRUE if one exists, or FALSE if one does not. It may use the nv_buffer_handle field in the structure pointed to by pfscntxt to store a handle for later access by failsafe_read_nv_buffer() and failsafe_write_nv_buffer().

**failsafe_write_nv_buffer**

**Function:**

Write a block to the failsafe buffer.

**Summary:**

BOOLEAN failsafe_write_nv_buffer(FAILSAFECONTEXT *pfscntxt,dword block_no,byte *pblock)

**Description:**

This routine must write one block to the block at offset block_no in the failsafe buffer on the current disk or in system NV RAM and return TRUE if successful, FALSE if it is unsuccessful.

**failsafe_read_nv_buffer**

**Function:**
Read a block from the failsafe buffer.

**Summary:**
BOOLEAN failsafe_read_nv_buffer(FAILSAFECONTEXT *pfscntxt, dword block_no, byte *pblock)

**Description:**
This routine must read one block from the block at offset block_no in the failsafe buffer on the current disk or in system NV RAM and return TRUE if successful, FALSE if it is unsuccessful.

**Examples**

**Example 1: Auto-Initialize failsafe mode.**

**First in apiinit.c set the** DRIVE_FLAGS_FAILSAFE **flag bits for the device. The following block demonstrates this procedure for the hostdisk device driver.**

```
if (++drives_used > prtfs_cfg->cfg_NDRIVES)
        goto need_more_drives;
    pdr->driveno = 0;                   /* MAPS DRIVE structure to A: */
    /* Install Using a host driver */
    pdr->dev_table_drive_io     = hostdisk_io;
    pdr->dev_table_perform_device_ioctl = hostdisk_perform_device_ioctl;
    STORE_DEVICE_NAME("HOST DISK HOSTDISK.DAT")
    pdr->register_file_address  = (dword) 0;    /* Not used  */
    pdr->interrupt_number       = 0;    /* Not used */
    pdr->drive_flags            = 0;
    /*  Turn on Failsafe if desired */
    pdr->drive_flags            |= DRIVE_FLAGS_FAILSAFE;
    pdr->partition_number       = 0;    /* Not used */
    pdr->pcmcia_slot_number     = 0;    /* Not used */
    pdr->controller_number      = 0;
    pdr->logical_unit_number    = 0;
    pdr->dev_table_drive_io     = hostdisk_io;
    pdr->dev_table_perform_device_ioctl = hostdisk_perform_device_ioctl;
    STORE_DEVICE_NAME("HOST DISK HOSTDISK.DAT")
    pdr++;
```

**Now modify pro_failsafe_auto_init() in prfsapi.c to appropriately initialize the selected device for FailSafe. The following example is the default implementation. To keep the default implementation easy to understand and to maximize flexibility we purposely kept the example simple. It may be modified to support multiple drives with different configuration parameters per drive. If you need to modify pro_failsafe_auto_init() in this way you may use pdrive->driveno to identify which device the call is being made for.**

**This example auto-initializes FailSafe mode for a single device. This example puts FailSafe in auto-restore, auto-commit and auto-recover mode and provides it with a 128 element block map. See the manual pages for pro_failsafe_ init() for a detailed discussion of FailSafe configuration values.**

```
#if (INCLUDE_FAILSAFE_AUTO_INIT)
FAILSAFECONTEXT auto_fscontext;
#define AUTO_BLOCKMAPSIZE 128
FSBLOCKMAP auto_failsafe_blockmap_array[AUTO_BLOCKMAPSIZE];
#endif

BOOLEAN pro_failsafe_auto_init(DDRIVE *pdrive)
{
```

```
#if (!INCLUDE_FAILSAFE_AUTO_INIT)
        return(FALSE);
#else
    /* In this example we set up failsafe mode for one device
       This code may be modified to enable failsafe on multiple
       devices and to modify FailSafe settings according to the
       documentation provided for pro_failsafe_init().
    */
    if (auto_fscontext.pdrive) /* Only one device supported int this */
        return (FALSE);        /* Example.  Add more contexts */

    rtfs_memset((void *) &auto_fscontext, 0, sizeof(auto_fscontext));
    /* Run fully automated */
    auto_fscontext.configuration_flags =
(FS_MODE_AUTORESTORE|FS_MODE_AUTORECOVER|FS_MODE_AUTOCOMMIT);
    auto_fscontext.blockmap_size = AUTO_BLOCKMAPSIZE;
    auto_fscontext.blockmap_freelist = &auto_failsafe_blockmap_array[0];
    /* Our parameters are correct so we know that the routine will not
       fail but to demonstrate the principle we return FALSE if the
       call does fail.  */
    if (!pro_failsafe_init_internal(pdrive, &auto_fscontext))
        return(FALSE);
    return(TRUE);
#endif /* (INCLUDE_FAILSAFE_AUTO_INIT) */
}
```

**Example 2: Manually initialize failsafe mode.**

**This example initializes from the API layer by calling pro_failsafe_init(). This example code is provided inside the file appcmdsh.c. FailSafe mode is selected for a single device. This example puts FailSafe in auto-restore, auto-commit and auto-recover mode and provides it with a 128 element block map. Comments inside the example code explain how these Failsafe options are selected and how other options may be selected. See the manual pages for pro_failsafe_init() for a detailed discussion of FailSafe configuration values.**

```
/* Reserve some storage for block mapping structure.  See the
   description of fscontext.blockmap_size below for a discussion
   on block maps */
FAILSAFECONTEXT fscontext;
#define BLOCKMAPSIZE 128
FSBLOCKMAP failsafe_blockmap_array[BLOCKMAPSIZE];

void app_failsafe_init(byte *drive_name)
{
   rtfs_memset((void *) &fscontext, 0, sizeof(fscontext));
    /* Initial setting.  autorestore, autocommit and autorecover options all disabled */
    fscontext.configuration_flags = 0;
    /* Select AUTORESTORE.  - The volume is autorestored if needed at mount time
      .  If restore is needed but the failsafe file is damaged the mount fails
        unless FS_MODE_AUTORECOVER is also enabled */
    fscontext.configuration_flags |= FS_MODE_AUTORESTORE;
    /* Select AUTORECOVER.  - If autorestore failed because the failsafe file
      is damaged skip the restore state and proceed with the mount */
    fscontext.configuration_flags |= FS_MODE_AUTORECOVER;
    /* Enable AUTOCOMMIT mode.  Force ERTFS to perform the FailSafe commit
       internally before the return from API calls that may have changed
       volume */
    fscontext.configuration_flags |= FS_MODE_AUTOCOMMIT;
    /* Optional setting - If the user_journal_size is set prior to the  call to pro_failsafe_init() the FailSafe file space is limited to this size.  Otherwise the Failsafe file size is set to the size of  the FAT plus CFG_NUM_JOURNAL_BLOCKS (prfs.h) blocks. The default setting is purposely conservative.  One block must be available for each FAT block and directory block that will be changed between calls to pro_failsafe_commit().  The following line, if enabled, will fix the failsafe journal size to 64K.  This would be appropriate for example if the journal file  was being held in 64K of nvram.  */
    /* fscontext.user_journal_size = 128; */    /* In blocks */
    /* Provide buffer space for failsafe to cache indexing information  pertaining to journaling activities.  Without adequate caching  failsafe will still perform as specified but with lowered performance.  Each blockmap element required approximately 32 bytes, for optimal performance one element should be        available for each FAT block and directory block that will be changed between calls to pro_failsafe_commit().  In this example we choose 128 entries.  This is very a conservative setting that will       allow failsafe to use caching in a session with up to 128  FAT and directory blocks changed.  Note that performance will degrade drastically if the number of  blocks modified exceeds the size of the block map.       If you have a need to minimize resources you may reduce blockmap_size  and user_journal_size.  To help determine appropriate values you may call pro_buffer_status(), the failsafe_blocks_used field in the status        structure will contain the number of these resources used.  In order to gauge the worst case usage, pro_buffer_status() should be called immediately prior to calling pro_failsafe_commit()    */
    fscontext.blockmap_size = BLOCKMAPSIZE;
    fscontext.blockmap_freelist = &failsafe_blockmap_array[0];
```

```
    if (!pro_failsafe_init(drive_name, &fscontext));
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_114, PRFLG_NL); /* “Failsafe init failed\n” */
        if (get_errno() == PEFSREINIT)
        {
     /* “Failsafe already running on a drive\n” */
     RTFS_PRINT_STRING_1(USTRING_TSTSH_115, PRFLG_NL);
     /* “Command shell only supports one failsafe session\n” */
     RTFS_PRINT_STRING_1(USTRING_TSTSH_116, PRFLG_NL);
      }
    }
}
```

**Example 3: Commit failsafe buffers to disk and clear the failsafe journal file**

If the commit succeeds reset the journal file. Once commit has succeeded the disk image is in sync with the memory image of the directory and fat buffers and the journal file is cleared. Note: Do not call this function If FailSafe is operating in auto-commit mode

```
int fs_demo_commit(byte *drivename)
{
   int err;
   if (!pro_failsafe_commit(drivename))
  {
      printf("Test1:Failsafe commit failed, error == %d\n", get_errno());
            return(-1);
  }
  else
      return(0);
}
```

**Example 4: pro_failsafe_init has already been called, use pro_failsafe_restore to determine the state of the journal file**

```
int fs_demo_journal_status(byte *drivename)
{
int rval;
  rval = pro_failsafe_restore(drivename,0,FALSE,FALSE);
  switch (rval)
  {
  case FS_STATUS_OK:
     printf("FAILSAFE: Volume up to date, no restore required\n");
     break;
 case FS_STATUS_NO_JOURNAL:
    printf("FAILSAFE: no journal present\n");
    break;
case FS_STATUS_BAD_JOURNAL:
    printf("FAILSAFE: bad journal file present\n");
    break;
case FS_STATUS_BAD_CHECKSUM:
  printf("FAILSAFE: journal file has checksum error\n");
  break;
case FS_STATUS_IO_ERROR:
  printf("FAILSAFE: IO error during restore\n");
  break;
case FS_STATUS_MUST_RESTORE:
 printf("FAILSAFE: Restore required\n");
 break;
case FS_STATUS_OUT_OF_DATE:
 printf("FAILSAFE: Volume and Failsafe File are out of synch\n");
 break;
case FS_STATUS_NO_INIT:
 printf("FAILSAFE: Failsafe must be initialized or a context must be provided to execute restore\n");
 break;
default:
 printf("FAILSAFE: Restore unexpected return code %d", rval);
 break;
 }
}
```

**Example 5: pro_failsafe_init has not been called, use pro_failsafe_restore to restore the volume from the journal file.**

```
FAILSAFECONTEXT fscontext;

int fs_demo_restore_volume(byte *drivename)
{
int rval;

  rval = pro_failsafe_restore(drivename,&fscontext,TRUE,FALSE);
  switch (rval)
  {
case FS_STATUS_RESTORED:
    printf("FAILSAFE: Volume was restored successfully from the journal file\n");
    break;
  case FS_STATUS_OK:
    printf("FAILSAFE: Volume up to date, no restore was required\n");
    break;
 case FS_STATUS_NO_JOURNAL:
   printf("FAILSAFE: restore failed, no journal present\n");
   break;
case FS_STATUS_BAD_JOURNAL:
   printf("FAILSAFE: restore failed, bad journal file present\n");
   break;
case FS_STATUS_BAD_CHECKSUM:
  printf("FAILSAFE: restore failed, journal file has checksum error\n");
  break;
case FS_STATUS_IO_ERROR:
  printf("FAILSAFE: restore failed, IO error during restore\n");
  break;
case FS_STATUS_OUT_OF_DATE:
 printf("FAILSAFE: Volume and Failsafe File are out of synch\n");
 break;
case FS_STATUS_NO_INIT:
 printf("FAILSAFE: Failsafe must be initialized to execute restore\n");
 break;
default:
 printf("FAILSAFE: Restore unexpected return code %d", rval);
 break;
 }
```

## APPENDIX A: PORTCONF.H

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS compile time configuration file */

```
#ifndef __PORTCONF__
#define __PORTCONF__ 1

/* CPU Configuration section */

#define KS_LITTLE_ENDIAN 1                  /* See porting reference guide for explanation */
#define KS_LITTLE_ODD_PTR_OK 1              /* See porting reference guide for explanation */
#define KS_CONSTANT const                   /* See porting reference guide for explanation */
#define KS_FAR                              /* See porting reference guide for explanation */

/* Compile time constants to control device inclusion and inclusion of
   porting layer subroutines */

#define INCLUDE_IDE                 0       /* - Include the IDE driver */
#define INCLUDE_PCMCIA              0       /* - Include the pcmcia driver */
#define INCLUDE_PCMCIA_SRAM         0       /* - Include pcmcia static ram card */
#define INCLUDE_COMPACT_FLASH       0       /* - Support compact flash */

#define INCLUDE_FLASH_FTL           1       /* - Include the linear flash driver */
#define INCLUDE_ROMDISK             1       /* - Include the rom disk driver */
#define INCLUDE_RAMDISK             1       /* - Include the rom disk driver */
#define INCLUDE_FLOPPY              0       /* - Include the floppy disk driver */
#define INCLUDE_HOSTDISK            1       /* - Include windows disk simulator */
#define INCLUDE_UDMA                0       /* - Include ultra dma support for ide */
#define INCLUDE_82365_PCMCTRL       0       /* - Include 82365 pcmcia controller */

#endif /* __PORTCONF__ */
```

## APPENDIX B. RTFSCONF.H

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS compile time configuration file */

```
#ifndef __RTFSCONF__
#define __RTFSCONF__ 1

/* Include CPU and peripheral configuration */
#include <portconf.h>

/* Character set support */
#define INCLUDE_CS_JIS          0       /* Set to 1 to support JIS (kanji)  */
#define INCLUDE_CS_ASCII        1       /* Set to 1 to support ASCII only  */
#define INCLUDE_CS_UNICODE      0       /* Set to 1 to support UNICODE */

/* Set to 1 to support long filenames */
#define VFAT            1
/* Set to 1 to support 32 bit FATs */
#define FAT32           1
/* Set to 0 to disable file share modes saves ~0.5 K */
#define RTFS_SHARE      0
/* Set to 0 to disable subdirs. Feature not implemented must be 1*/
#define RTFS_SUBDIRS    1
/* Set to 0 to disable write support. Feature not implemented must be 1*/
#define RTFS_WRITE      1
/* Set to 1 to include failsafe support */
#define INCLUDE_FAILSAFE_CODE 0

/* Set to 1 to include support for extended DOS partitions */
/* ERTFS contains code to interpret extended DOS partitions but since this feature is rarely used it is provided as a compile time option */
#define SUPPORT_EXTENDED_PARTITIONS 0

/* STORE_DEVICE_NAMES_IN_DRIVE_STRUCT - If this value is set to one then we save device names for future viewing by diagnostics */
#define STORE_DEVICE_NAMES_IN_DRIVE_STRUCT 1

/* Set to the maximum file size ERTFS may create. If po_chsize or po_extend_file() are called with a size request larger than this they fail and set errno to PETOOLARGE. When po_write() is asked to expend the file beyond this maximum
   the behavior is determined by the value of RTFS_TRUNCATE_WRITE_TO_MAX */
#define RTFS_MAX_FILE_SIZE      0xffffffff

/* Set to 1 to force RTFS to truncate po_write() requests to fit within
   RTFS_MAX_FILE_SIZE. If RTFS_TRUNCATE_WRITE_TO_MAX is set to 0, po_write requests that attempt to extend the file beyond RTFS_TRUNCATE_WRITE_TO_MAX Fail and set errno to PETOOLARGE. If RTFS_TRUNCATE_WRITE_TO_MAX is set to 1, po_write requests that attempt to extend the file beyond RTFS_MAX_FILE_SIZE are truncated to fill the file until its size reaches RTFS_MAX_FILE_SIZE bytes. */
#define RTFS_TRUNCATE_WRITE_TO_MAX    1

/* When scanning a directory cluster chain fail if more than this many
   clusters are in the chain. (Indicates endless loop)
*/
#define MAX_CLUSTERS_PER_DIR 4096

#endif /* __RTFSCONF__ */
```

## APPENDIX C: APICNFIG.C

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS run time configuration file */

**PC_ERTFS_CONFIG()** - Set up the ERTFS configuration block and allocate or assign memory blocks for ERTFS usage.

The user must modify this code if he wishes to reconfigure ERTFS. He must also initialize the ERTFS configuration structure with the addresses of memory blocks to be used by ERTFS.

**Tutorial:**

**pc_ertfs_config(void)** initializes the ERTFS configuration block and provides ERTFS with the addresses of memory that it needs. It is called from the ERTFS initialization routine **pc_ertfs_init()**.

This routine is designed to be modified by the user if he wishes to change the default configuration. To simplify the user's task we define some configuration constants in this file that may be modified to change the configuration. These constants are only used locally to this file. If another method of configuring ERTFS is more appropriate for your environment then devise an alternate method to initialize the configuration block.

The default configuration is:

| | | |
|---|---|---|
| ALLOC_FROM_HEAP | 1 | Use malloc() to allocate, set to 0 to use declared memory arrays. |
| NDRIVES | 10 | L: Max number of drives from 0 to 26 A: - Z: |

*Note: A FAT buffer pool must be allocated for each drive. If NDRIVES is changed you must add code to allocate or assign more FAT buffer pools. The default configuration allocates or assigns 10 fat buffer pools. If you increase NDRIVES, then add code that duplicates the current code that initializes prtfs_cfg->fat_buffers[9], to drives 10, 11, 12. If you decrease NDRIVES, then removes the current code that initializes prtfs_cfg->fat_buffers[9], 8, 7, 6 etc. If you are not using ALLOC_ FROM_HEAP, you should also add or remove the memory array declarations.*

| | | |
|---|---|---|
| NUM_USERS | 1 | Maximum number of USER contexts set to 1 for POLLED mode, otherwise set to the number of tasks that will simultaneously use ERTFS |
| NBLKBUFFS | 10 | Number of blocks in the buffer pool. Uses 532 bytes per block. Impacts performance during directory traversals. Must be at least 4 |
| BLKHASHSIZE | 16 | Size of the block buffer hash table. This value must be a power of two. |
| NUSERFILES | 10 | The maximum number of open Files at a time |
| LARGE_FAT_SIZE | 32 | The number of 520 byte blocks committed for buffering fat blocks on drive C: |
| LARGE_FAT_HASHSIZE | 32 | The number of 12 byte hash table entries committed for use on drive C: This value must be a power of two. |
| SMALL_FAT_SIZE | 2 | The number of 520 byte blocks committed for buffering fat blocks on drives other than C: |
| SMALL_FAT_HASHSIZE | 2 | The number of 12 byte hash table entries committed for use on drives other than C: This value must be a power of two. |

*Note: Each drive may have different fat buffer and cache sizes. For convenience here we only use two possible different sizes. You may tune each drive individually by setting the configuration value:*

```
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[DRIVENUMBER]; and
    prtfs_cfg->cfg_FAT_CACHE_SIZE[DRIVENUMBER];
    prtfs_cfg->fat_buffers[DRIVENUMBER];
    prtfs_cfg->fat_hash_table[DRIVENUMBER];
*/
#include <rtfs.h>
#define ALLOC_FROM_HEAP     0  /* Set this to 1 to use malloc() to allocate ERTFS memory at startup, set to 0 to use
                                  declare memory arrays and provide ertfs with the addresses of those arrays */
#define NDRIVES             10 /* Number of drives */
#define NUM_USERS           1  /* Must be one for POLLOS */
#define NBLKBUFFS           4  /* Number of blocks in the buffer pool. Uses 532 bytes per block. Impacts performance
                                  during directory traversals must be at least 4 */
#define BLKHASHSIZE         16 /* This value must be a power of two. Size of the block buffer hash table.  */
#define LARGE_FAT_SIZE      32 /* Number of 520 byte blocks committed for buffering fat blocks on drive C: */
#define LARGE_FAT_HASHSIZE  32 /* This value must be a power of two.  Number of 12 byte hash table entries committed
                                  for use on drive C: */
#define SMALL_FAT_SIZE      2  /* Number of 520 byte blocks committed for buffering fat blocks all other drives */
#define SMALL_FAT_HASHSIZE  2  /* This value must be a power of two.  Number of 12 byte hash table entries committed
```

```
                                        for use on all other drives */
#define NUSERFILES                  10 /* Maximum Number of open Files at a time */
/* Directory Object Needs. Conservative guess is One CWD per user per drive + One per file + one per User for directory
traversal */
/* This is calculated... do not change */
#define NFINODES   (32 + NUM_USERS*NDRIVES + NUM_USERS + NUSERFILES)
#define NDROBJS    (32 + NUM_USERS*NDRIVES + NUM_USERS + NUSERFILES)
RTFS_CFG *prtfs_cfg;                /* The user must initialize this value to point to a valid configuration block */
static RTFS_CFG rtfs_cfg_core;      /* The user must initialize this value to point */
#if (ALLOC_FROM_HEAP)
/* Using malloc to allocate memory at startup. If malloc is not available but some other heap manager is, then change the calls
to malloc() to call your heap manager instead */
#include <malloc.h>
#else
/* Not using malloc to allocate memory so declare arrays that we can send assign to the configuration block at startup. */
DDRIVE              __mem_drives_structures[NDRIVES];
BLKBUFF     __mem_block_pool[NBLKBUFFS];
BLKBUFF *   __mem_block_hash_table[BLKHASHSIZE];
PC_FILE     __mem_file_pool[NUSERFILES];
DROBJ       __mem_drobj_pool[NDROBJS];
FINODE __mem_finode_pool[NFINODES];
RTFS_SYSTEM_USER __rtfs_user_table[NUM_USERS];

FATBUFF KS_FAR __fat_buffer_0[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_1[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_2[LARGE_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_3[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_4[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_5[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_6[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_7[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_8[SMALL_FAT_SIZE];
FATBUFF KS_FAR __fat_buffer_9[SMALL_FAT_SIZE];

FATBUFF * KS_FAR __fat_hash_table_0[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_1[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_2[LARGE_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_3[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_4[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_5[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_6[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_7[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_8[SMALL_FAT_HASHSIZE];
FATBUFF * KS_FAR __fat_hash_table_9[SMALL_FAT_HASHSIZE];

byte * KS_FAR __fat_primary_cache_0[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_1[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_2[LARGE_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_3[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_4[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_5[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_6[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_7[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_8[SMALL_FAT_HASHSIZE];
byte * KS_FAR __fat_primary_cache_9[SMALL_FAT_HASHSIZE];

dword  KS_FAR __fat_primary_index_0[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_1[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_2[LARGE_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_3[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_4[SMALL_FAT_HASHSIZE];
```

```
dword  KS_FAR __fat_primary_index_5[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_6[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_7[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_8[SMALL_FAT_HASHSIZE];
dword  KS_FAR __fat_primary_index_9[SMALL_FAT_HASHSIZE];

#endif /* (ALLOC_FROM_HEAP) */

BOOLEAN pc_ertfs_config(void)
{
    /* Important: prtfs_cfg must point to a configuration block */
    prtfs_cfg = &rtfs_cfg_core;

    /* Important: the configuration block must be zeroed */
    rtfs_memset(prtfs_cfg, 0, sizeof(rtfs_cfg_core));

    /* Set Configuration values */
    prtfs_cfg->cfg_NDRIVES                  = NDRIVES;
    prtfs_cfg->cfg_NBLKBUFFS                = NBLKBUFFS;
    prtfs_cfg->cfg_BLK_HASHTBLE_SIZE        = BLKHASHSIZE;
    prtfs_cfg->cfg_NUSERFILES               = NUSERFILES;
    prtfs_cfg->cfg_NDROBJS                  = NDROBJS;
    prtfs_cfg->cfg_NFINODES                 = NFINODES;
    prtfs_cfg->cfg_NUM_USERS                = NUM_USERS;

    /* Set FAT size configuration values for each drive */
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[0]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[1]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[2]   = LARGE_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[3]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[4]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[5]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[6]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[7]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[8]   = SMALL_FAT_SIZE;
    prtfs_cfg->cfg_FAT_BUFFER_SIZE[9]   = SMALL_FAT_SIZE;

    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[0]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[1]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[2]  = LARGE_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[3]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[4]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[5]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[6]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[7]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[8]  = SMALL_FAT_HASHSIZE;
    prtfs_cfg->cfg_FAT_HASHTBL_SIZE[9]  = SMALL_FAT_HASHSIZE;

    /* Core that must be provided by the user */
#if (!ALLOC_FROM_HEAP)

    /* Not using malloc() so assign memory arrays to the configuration block */
    prtfs_cfg->mem_drives_structures        = (DDRIVE *)            &__mem_drives_structures[0];
    prtfs_cfg->mem_block_pool               = (BLKBUFF *)           &__mem_block_pool[0];
    prtfs_cfg->mem_block_hash_table         = (BLKBUFF **)          &__mem_block_hash_table[0];
    prtfs_cfg->mem_file_pool                = (PC_FILE *)           &__mem_file_pool[0];
    prtfs_cfg->mem_drobj_pool               = (DROBJ *)             &__mem_drobj_pool[0];
    prtfs_cfg->mem_finode_pool              = (FINODE *)            &__mem_finode_pool[0];
    prtfs_cfg->rtfs_user_table              = (RTFS_SYSTEM_USER *)  &__rtfs_user_table[0];
    prtfs_cfg->fat_buffers[0] = (FATBUFF *) &__fat_buffer_0[0];
    prtfs_cfg->fat_buffers[1] = (FATBUFF *) &__fat_buffer_1[0];
```

```
    prtfs_cfg->fat_buffers[2] = (FATBUFF *)  &__fat_buffer_2[0];
    prtfs_cfg->fat_buffers[3] = (FATBUFF *)  &__fat_buffer_3[0];
    prtfs_cfg->fat_buffers[4] = (FATBUFF *)  &__fat_buffer_4[0];
    prtfs_cfg->fat_buffers[5] = (FATBUFF *)  &__fat_buffer_5[0];
    prtfs_cfg->fat_buffers[6] = (FATBUFF *)  &__fat_buffer_6[0];
    prtfs_cfg->fat_buffers[7] = (FATBUFF *)  &__fat_buffer_7[0];
    prtfs_cfg->fat_buffers[8] = (FATBUFF *)  &__fat_buffer_8[0];
    prtfs_cfg->fat_buffers[9] = (FATBUFF *)  &__fat_buffer_9[0];

    prtfs_cfg->fat_hash_table[0] = (FATBUFF **) &__fat_hash_table_0[0];
    prtfs_cfg->fat_hash_table[1] = (FATBUFF **) &__fat_hash_table_1[0];
    prtfs_cfg->fat_hash_table[2] = (FATBUFF **) &__fat_hash_table_2[0];
    prtfs_cfg->fat_hash_table[3] = (FATBUFF **) &__fat_hash_table_3[0];
    prtfs_cfg->fat_hash_table[4] = (FATBUFF **) &__fat_hash_table_4[0];
    prtfs_cfg->fat_hash_table[5] = (FATBUFF **) &__fat_hash_table_5[0];
    prtfs_cfg->fat_hash_table[6] = (FATBUFF **) &__fat_hash_table_6[0];
    prtfs_cfg->fat_hash_table[7] = (FATBUFF **) &__fat_hash_table_7[0];
    prtfs_cfg->fat_hash_table[8] = (FATBUFF **) &__fat_hash_table_8[0];
    prtfs_cfg->fat_hash_table[9] = (FATBUFF **) &__fat_hash_table_9[0];

    prtfs_cfg->fat_primary_cache[0] = (byte **) &__fat_primary_cache_0[0];
    prtfs_cfg->fat_primary_cache[1] = (byte **) &__fat_primary_cache_1[0];
    prtfs_cfg->fat_primary_cache[2] = (byte **) &__fat_primary_cache_2[0];
    prtfs_cfg->fat_primary_cache[3] = (byte **) &__fat_primary_cache_3[0];
    prtfs_cfg->fat_primary_cache[4] = (byte **) &__fat_primary_cache_4[0];
    prtfs_cfg->fat_primary_cache[5] = (byte **) &__fat_primary_cache_5[0];
    prtfs_cfg->fat_primary_cache[6] = (byte **) &__fat_primary_cache_6[0];
    prtfs_cfg->fat_primary_cache[7] = (byte **) &__fat_primary_cache_7[0];
    prtfs_cfg->fat_primary_cache[8] = (byte **) &__fat_primary_cache_8[0];
    prtfs_cfg->fat_primary_cache[9] = (byte **) &__fat_primary_cache_9[0];

    prtfs_cfg->fat_primary_index[0] = (dword *) &__fat_primary_index_0[0];
    prtfs_cfg->fat_primary_index[1] = (dword *) &__fat_primary_index_1[0];
    prtfs_cfg->fat_primary_index[2] = (dword *) &__fat_primary_index_2[0];
    prtfs_cfg->fat_primary_index[3] = (dword *) &__fat_primary_index_3[0];
    prtfs_cfg->fat_primary_index[4] = (dword *) &__fat_primary_index_4[0];
    prtfs_cfg->fat_primary_index[5] = (dword *) &__fat_primary_index_5[0];
    prtfs_cfg->fat_primary_index[6] = (dword *) &__fat_primary_index_6[0];
    prtfs_cfg->fat_primary_index[7] = (dword *) &__fat_primary_index_7[0];
    prtfs_cfg->fat_primary_index[8] = (dword *) &__fat_primary_index_8[0];
    prtfs_cfg->fat_primary_index[9] = (dword *) &__fat_primary_index_9[0];
        return(TRUE);

#else
    /* Use Malloc do allocated ERTFS data */
    prtfs_cfg->mem_drives_structures = (DDRIVE *)          malloc(prtfs_cfg->cfg_NDRIVES*sizeof(DDRIVE));
    prtfs_cfg->mem_block_pool        = (BLKBUFF*)malloc(prtfs_cfg->cfg_NBLKBUFFS*sizeof(BLKBUFF));

    prtfs_cfg->mem_block_hash_table  = (BLKBUFF **)        malloc(prtfs_cfg-
                                          >cfg_BLK_HASHTBLE_SIZE*sizeof(BLKBUFF *));
    prtfs_cfg->mem_file_pool               = (PC_FILE *)            malloc(prtfs_cfg->cfg_NUSERFILES* sizeof(PC_
FILE));
    prtfs_cfg->mem_drobj_pool        = (DROBJ *)           malloc(prtfs_cfg->cfg_NDROBJS* sizeof(DROBJ));

    prtfs_cfg->mem_finode_pool        = (FINODE *)         malloc(prtfs_cfg->cfg_NFINODES* sizeof(FINODE));
    prtfs_cfg->rtfs_user_table       = (RTFS_SYSTEM_USER *) malloc(prtfs_cfg->cfg_NUM_USERS *
                                          sizeof(RTFS_SYSTEM_USER));

    prtfs_cfg->fat_buffers[0] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[0]*sizeof(FATBUFF));
    prtfs_cfg->fat_buffers[1] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[1]*sizeof(FATBUFF));
    prtfs_cfg->fat_buffers[2] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[2]*sizeof(FATBUFF));
```

```
prtfs_cfg->fat_buffers[3] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[3]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[4] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[4]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[5] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[5]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[6] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[6]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[7] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[7]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[8] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[8]*sizeof(FATBUFF));
prtfs_cfg->fat_buffers[9] = (FATBUFF *)  malloc(prtfs_cfg->cfg_FAT_BUFFER_SIZE[9]*sizeof(FATBUFF));

prtfs_cfg->fat_hash_table[0] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[0]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[1] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[1]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[2] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[2]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[3] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[3]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[4] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[4]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[5] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[5]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[6] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[6]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[7] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[7]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[8] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[8]*sizeof(FATBUFF *));
prtfs_cfg->fat_hash_table[9] = (FATBUFF **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[9]*sizeof(FATBUFF *));

prtfs_cfg->fat_primary_cache[0] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[0]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[1] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[1]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[2] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[2]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[3] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[3]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[4] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[4]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[5] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[5]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[6] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[6]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[7] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[7]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[8] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[8]*sizeof(byte *));
prtfs_cfg->fat_primary_cache[9] = (byte **) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[9]*sizeof(byte *));

prtfs_cfg->fat_primary_index[0] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[0]*sizeof(dword));
prtfs_cfg->fat_primary_index[1] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[1]*sizeof(dword));
prtfs_cfg->fat_primary_index[2] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[2]*sizeof(dword));
prtfs_cfg->fat_primary_index[3] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[3]*sizeof(dword));
prtfs_cfg->fat_primary_index[4] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[4]*sizeof(dword));
prtfs_cfg->fat_primary_index[5] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[5]*sizeof(dword));
prtfs_cfg->fat_primary_index[6] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[6]*sizeof(dword));
prtfs_cfg->fat_primary_index[7] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[7]*sizeof(dword));
prtfs_cfg->fat_primary_index[8] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[8]*sizeof(dword));
prtfs_cfg->fat_primary_index[9] = (dword *) malloc(prtfs_cfg->cfg_FAT_HASHTBL_SIZE[9]*sizeof(dword));

      /* Do some sanity checks */
if (!prtfs_cfg->mem_drives_structures    ||
      !prtfs_cfg->mem_block_pool         ||
      !prtfs_cfg->mem_block_hash_table   ||
      !prtfs_cfg->mem_file_pool          ||
      !prtfs_cfg->mem_drobj_pool         ||
      !prtfs_cfg->mem_finode_pool        ||
      !prtfs_cfg->rtfs_user_table)
            goto malloc_failed;

 if (!prtfs_cfg->fat_buffers[0]          ||
      !prtfs_cfg->fat_buffers[1]         ||
      !prtfs_cfg->fat_buffers[2]         ||
      !prtfs_cfg->fat_buffers[3]         ||
      !prtfs_cfg->fat_buffers[4]         ||
      !prtfs_cfg->fat_buffers[5]         ||
      !prtfs_cfg->fat_buffers[6]         ||
      !prtfs_cfg->fat_buffers[7]         ||
      !prtfs_cfg->fat_buffers[8]         ||
      !prtfs_cfg->fat_buffers[9])
```

```
            goto malloc_failed;

        if (!prtfs_cfg->fat_hash_table[0]           ||
            !prtfs_cfg->fat_hash_table[1]           ||
            !prtfs_cfg->fat_hash_table[2]           ||
            !prtfs_cfg->fat_hash_table[3]           ||
            !prtfs_cfg->fat_hash_table[4]           ||
            !prtfs_cfg->fat_hash_table[5]           ||
            !prtfs_cfg->fat_hash_table[6]           ||
            !prtfs_cfg->fat_hash_table[7]           ||
            !prtfs_cfg->fat_hash_table[8]           ||
            !prtfs_cfg->fat_hash_table[9])
                goto malloc_failed;

        if (!prtfs_cfg->fat_primary_cache[0]        ||
            !prtfs_cfg->fat_primary_cache[1]        ||
            !prtfs_cfg->fat_primary_cache[2]        ||
            !prtfs_cfg->fat_primary_cache[3]        ||
            !prtfs_cfg->fat_primary_cache[4]        ||
            !prtfs_cfg->fat_primary_cache[5]        ||
            !prtfs_cfg->fat_primary_cache[6]        ||
            !prtfs_cfg->fat_primary_cache[7]        ||
            !prtfs_cfg->fat_primary_cache[8]        ||
            !prtfs_cfg->fat_primary_cache[9])
                goto malloc_failed;

        if (!prtfs_cfg->fat_primary_index[0]||
            !prtfs_cfg->fat_primary_index[1]        ||
            !prtfs_cfg->fat_primary_index[2]        ||
            !prtfs_cfg->fat_primary_index[3]        ||
            !prtfs_cfg->fat_primary_index[4]        ||
            !prtfs_cfg->fat_primary_index[5]        ||
            !prtfs_cfg->fat_primary_index[6]        ||
            !prtfs_cfg->fat_primary_index[7]        ||
            !prtfs_cfg->fat_primary_index[8]        ||
            !prtfs_cfg->fat_primary_index[9])
                goto malloc_failed;

            return(TRUE);

malloc_failed:
        if (prtfs_cfg->mem_drives_structures)    free(prtfs_cfg->mem_drives_structures );
        if (prtfs_cfg->mem_block_pool)      free(prtfs_cfg->mem_block_pool);
        if (prtfs_cfg->mem_block_hash_table)    free(prtfs_cfg->mem_block_hash_table);
        if (prtfs_cfg->mem_file_pool)           free(prtfs_cfg->mem_file_pool);
        if (prtfs_cfg->mem_drobj_pool)          free(prtfs_cfg->mem_drobj_pool);
        if (prtfs_cfg->mem_finode_pool)         free(prtfs_cfg->mem_finode_pool);
        if (prtfs_cfg->rtfs_user_table)         free(prtfs_cfg->rtfs_user_table);
        if (prtfs_cfg->fat_buffers[0])          free(prtfs_cfg->fat_buffers[0]);
        if (prtfs_cfg->fat_buffers[1])          free(prtfs_cfg->fat_buffers[1]);
        if (prtfs_cfg->fat_buffers[2])          free(prtfs_cfg->fat_buffers[2]);
        if (prtfs_cfg->fat_buffers[3])          free(prtfs_cfg->fat_buffers[3]);
        if (prtfs_cfg->fat_buffers[4])          free(prtfs_cfg->fat_buffers[4]);
        if (prtfs_cfg->fat_buffers[5])          free(prtfs_cfg->fat_buffers[5]);
        if (prtfs_cfg->fat_buffers[6])          free(prtfs_cfg->fat_buffers[6]);
        if (prtfs_cfg->fat_buffers[7])          free(prtfs_cfg->fat_buffers[7]);
        if (prtfs_cfg->fat_buffers[8])          free(prtfs_cfg->fat_buffers[8]);
        if (prtfs_cfg->fat_buffers[9])          free(prtfs_cfg->fat_buffers[9]);
        if (prtfs_cfg->fat_hash_table[0])   free(prtfs_cfg->fat_hash_table[0]);
        if (prtfs_cfg->fat_hash_table[1])   free(prtfs_cfg->fat_hash_table[1]);
        if (prtfs_cfg->fat_hash_table[2])   free(prtfs_cfg->fat_hash_table[2]);
```

```
    if (prtfs_cfg->fat_hash_table[3])     free(prtfs_cfg->fat_hash_table[3]);
    if (prtfs_cfg->fat_hash_table[4])     free(prtfs_cfg->fat_hash_table[4]);
    if (prtfs_cfg->fat_hash_table[5])     free(prtfs_cfg->fat_hash_table[5]);
    if (prtfs_cfg->fat_hash_table[6])     free(prtfs_cfg->fat_hash_table[6]);
    if (prtfs_cfg->fat_hash_table[7])     free(prtfs_cfg->fat_hash_table[7]);
    if (prtfs_cfg->fat_hash_table[8])     free(prtfs_cfg->fat_hash_table[8]);
    if (prtfs_cfg->fat_hash_table[9])     free(prtfs_cfg->fat_hash_table[9]);
    if (prtfs_cfg->fat_primary_cache[0])free(prtfs_cfg->fat_primary_cache[0]);
    if (prtfs_cfg->fat_primary_cache[1])free(prtfs_cfg->fat_primary_cache[1]);
    if (prtfs_cfg->fat_primary_cache[2])free(prtfs_cfg->fat_primary_cache[2]);
    if (prtfs_cfg->fat_primary_cache[3])free(prtfs_cfg->fat_primary_cache[3]);
    if (prtfs_cfg->fat_primary_cache[4])free(prtfs_cfg->fat_primary_cache[4]);
    if (prtfs_cfg->fat_primary_cache[5])free(prtfs_cfg->fat_primary_cache[5]);
    if (prtfs_cfg->fat_primary_cache[6])free(prtfs_cfg->fat_primary_cache[6]);
    if (prtfs_cfg->fat_primary_cache[7])free(prtfs_cfg->fat_primary_cache[7]);
    if (prtfs_cfg->fat_primary_cache[8])free(prtfs_cfg->fat_primary_cache[8]);
    if (prtfs_cfg->fat_primary_cache[9])free(prtfs_cfg->fat_primary_cache[9]);
    if (prtfs_cfg->fat_primary_index[0]) free(prtfs_cfg->fat_primary_index[0]);
    if (prtfs_cfg->fat_primary_index[1]) free(prtfs_cfg->fat_primary_index[1]);
    if (prtfs_cfg->fat_primary_index[2]) free(prtfs_cfg->fat_primary_index[2]);
    if (prtfs_cfg->fat_primary_index[3]) free(prtfs_cfg->fat_primary_index[3]);
    if (prtfs_cfg->fat_primary_index[4]) free(prtfs_cfg->fat_primary_index[4]);
    if (prtfs_cfg->fat_primary_index[5]) free(prtfs_cfg->fat_primary_index[5]);
    if (prtfs_cfg->fat_primary_index[6]) free(prtfs_cfg->fat_primary_index[6]);
    if (prtfs_cfg->fat_primary_index[7]) free(prtfs_cfg->fat_primary_index[7]);
    if (prtfs_cfg->fat_primary_index[8]) free(prtfs_cfg->fat_primary_index[8]);
    if (prtfs_cfg->fat_primary_index[9]) free(prtfs_cfg->fat_primary_index[9]);

    return(FALSE);
#endif
}
```

## APPENDIX D. APPINIT.C

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS run time device driver init file */

```
BOOLEAN pc_ertfs_init(void)
{
int j;
DDRIVE *pdr;
int drives_used;
byte drname[8]; /* Temp buffer for displaying drive letters as strings */

/* Call the user supplied configuration function */
prtfs_cfg = 0;
if (!pc_ertfs_config())
return(FALSE);

if (!pc_memory_init())
      return(FALSE);

pdr = prtfs_cfg->mem_drives_structures;
drives_used = 0;
#if (INCLUDE_FLOPPY)
if (++drives_used >= prtfs_cfg->cfg_NDRIVES)
goto need_more_drives;
/* Floppy as A: */
pdr->driveno = 0;                                       // MAPS DRIVE structure to A:
pdr->dev_table_drive_io                         = floppy_io;
pdr->dev_table_perform_device_ioctl     = floppy_perform_device_ioctl;
STORE_DEVICE_NAME("Floppy unit 0")
 pdr->register_file_address             = (dword) 0;        // Not used for floppy
 pdr->interrupt_number                          = 0;                    // Not used for floppy
 pdr->drive_flags                                       = 0;                    // Set by warmstart
 pdr->partition_number                          = 0;                    //
 pdr->pcmcia_slot_number                = 0;                    // Not used for floppy
 pdr->controller_number                         = 0;
 pdr->logical_unit_number                       = 0;
 pdr++;
#endif

#if (INCLUDE_IDE)
   /* I/O addresses and interrupts used for ATA devices */
   /*       Primary  Secondary Tertiary Quaternary
    *       0x01f0  0x0170     0x01e8   0x0168
    *             0x03f6  0x0376     0x03ee   0x036e
    *             14      15         11       10
    *             Note: -1 for interrupts means used polled
    */

/* Install DRIVE C: As the primary controller, master IDE drive  first partition */
if (++drives_used >= prtfs_cfg->cfg_NDRIVES)
      goto need_more_drives;
/* First IDE as C:*/
pdr->driveno = 2;                       // MAPS DRIVE structure to C:
pdr->dev_table_drive_io     = ide_io;
pdr->dev_table_perform_device_ioctl = ide_perform_device_ioctl;
STORE_DEVICE_NAME("Primary Fixed IDE Master")
pdr->register_file_address  = (dword) 0x1f0; // Primary I/O address
// Note: Set interrupt number to -1 to use polled mode
pdr->interrupt_number       = 14;   //  Primary ATA interrupt
pdr->drive_flags            = DRIVE_FLAGS_PARTITIONED;
pdr->partition_number               = 0;     //
```

```
pdr->pcmcia_slot_number             = 0;
pdr->controller_number              = 0;
pdr->logical_unit_number            = 0;
pdr++;

/* Install DRIVE D: As the primary controller, master IDE drive second partition */
if (++drives_used >= prtfs_cfg->cfg_NDRIVES)
goto need_more_drives;
pdr->driveno = 3;                       // MAPS DRIVE structure to D:
pdr->dev_table_drive_io     = ide_io;
pdr->dev_table_perform_device_ioctl = ide_perform_device_ioctl;
STORE_DEVICE_NAME("Primary Fixed IDE Master 2nd Partition")
pdr->register_file_address  = (dword) 0x1f0; // Primary I/O address
// Note: Set interrupt number to -1 to use polled mode
pdr->interrupt_number       = 14;    // Primary ATA interrupt
pdr->drive_flags            = DRIVE_FLAGS_PARTITIONED;
pdr->partition_number       = 1;     //
pdr->pcmcia_slot_number     = 0;
pdr->controller_number      = 0;
pdr->logical_unit_number    = 0;
pdr++;
#endif

.. etc ..
.. etc ..

/* End User initialization section */

print_device_names();

pdr= prtfs_cfg->mem_drives_structures;
for (j = 0; j < prtfs_cfg->cfg_NDRIVES; j++, pdr++)
{
       if (pdr->dev_table_drive_io)
       {
            prtfs_cfg->drno_to_dr_map[pdr->driveno] = pdr; // MAPS DRIVE structure to DRIVE:
            if (pdr->dev_table_perform_device_ioctl(pdr->driveno, DEVCTL_WARMSTART,
                  PFVOID) 0) != 0)
            prtfs_cfg->drno_to_dr_map[pdr->driveno] = 0;      // It is not there.
                        // so forget it
            }
      }
/* Autoformatting RAM Devices */
pdr= prtfs_cfg->mem_drives_structures;
for (j = 0; j < prtfs_cfg->cfg_NDRIVES; j++, pdr++)
{
if ( pdr->drive_flags&DRIVE_FLAGS_VALID && pdr->drive_flags&DRIVE_FLAGS_FORMAT)
{
      auto_format_disk(pdr, drno_to_string(drname, pdr->driveno);
}
return(TRUE);
```

## APPENDIX E. PORTKERN.C

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS RTOS porting layer file */

```
/*
* portkern.c
*
If you are using POLLED mode, then this file is actually quite portable.  Several reference ports are  provided for x86 environments. To port it to a non supported environment you must provide the following functions and macros:
*      unsigned long rtfs_port_get_ticks();
*   #define  MILLISECONDS_PER_TICK
*
*
* Depending on what drivers are required, one or more of the following
*
* void hook_82365_pcmcia_interrupt(int irq)
* void hook_ide_interrupt(int irq, int controller_number)
* void hook_ide_interrupt(int irq, int controller_number)
* void hook_floppy_interrupt(int irq)
*
*/

#include <rtfs.h>

/* This routine takes no arguments and returns an unsigned long. The routine must return a tick count from the system clock. The macro named MILLISECONDS_PER_TICK must be defined in such a way so that it returns the rate at which the tick increases in milliseconds per tick */
/* This routine is declared as static to emphasize that its use is local to the portkern.c file only */

#define  MILLISECONDS_PER_TICK 1 /* Change this to your environment */

static dword rtfs_port_get_ticks(void)
{
      return(0); /* Return the real periodic clock tick value here */
}

/*
This routine takes no arguments and returns an unsigned long. The routine must allocate and initialize a Mutex, setting it to the “not owned” state. It must return an unsigned long value that will be used as a handle. ERTFS will not interpret the value of the return value. The handle will only used as an argument to the rtfs_port_claim_mutex() and rtfs_port_release_mutex() calls. The handle may be used as an index into a table or it may be cast internally to an RTOS specific pointer. If the mutex allocation function fails this routine must return 0 and the ERTFS calling function will return failure.
*/

dword rtfs_port_alloc_mutex(void)
{
      /* POLLED mode does not require mutexes */
      /* Otherwise implement it */
      return(1);

}
/* This routine takes as an argument a mutex handle that was returned by rtfs_port_alloc_mutex(). If the mutex is already claimed it must wait for it to be released and then claim the mutex and return.
*/

void rtfs_port_claim_mutex(dword handle)
{
      /* POLLED mode does not require mutexes */
      /* Otherwise implement it */
```

```
}

/* This routine takes as an argument a mutex handle that was returned by rtfs_port_alloc_mutex() that was previously
claimed by a call to rtfs_port_claim_mutex(). It must release the handle and cause a caller blocked in rtfs_port_claim_mu-
tex() for that same handle to unblock.
*/

void rtfs_port_release_mutex(dword handle)
{
    /* POLLED mode does not require mutexes */
    /* Otherwise implement it */
}

/* Note: Currently ERTFS only requires these functions if the floppy disk device driver is being used or if the IDE device
driver is being used in interrupt mode rather than polled mode. Additional user supplied device drivers may opt to use
these calls or they may implement their own native signaling  method at the implementer's discretion. If you are not using
the floppy disk driver or the IDE driver in interrupt mode please skip this section.
The signaling code is abstracted into four functions that must be modified by the user to support the target RTOS. The
required functions are: rtfs_port_alloc_signal(), rtfs_port_clear_signal(), rtfs_port_test_signal() and rtfs_port_set_signal().
The requirements of each of these functions is defined below.
*/

/* This routine takes no arguments and returns an unsigned long. The routine must allocate and initialize a signaling de-
vice (typically a counting semaphore) and set it to the "not signaled" state. It must return an unsigned long value that will
be used as a handle. ERTFS will not interpret the value of the return value. The handle will only used as an argument to
the rtfs_port_clear_signal(), rtfs_port_test_signal() and rtfs_port_set_signal() calls.  Only required for the supplied floppy
disk and ide device driver if the ide driver is running in interrupt mode. Otherwise leave this function as it is, it will not be
used. */

dword rtfs_port_alloc_signal(void)
{
    /* Polled mode has only one global signal */
    /* Otherwise implement it */
    return(1);
}

/* This routine takes as an argument a handle that was returned by rtfs_port_alloc_signal(). It must place the signal in an
unsignaled state such that a subsequent call to rtfs_port_test_signal() will not return success until rtfs_port_set_signal()
has been called. This clear function is necessary since it is possible although unlikely that an interrupt service routine
could call rtfs_port_set_signal() after the intended call to rtfs_port_test_signal() timed out. A typical implementation of this
function for a counting semaphore is to set the count value to zero or to poll it until it returns failure. */
int polled_signal = 0;   /* We only need one signal for polled mode since it
                            is single threaded */
void rtfs_port_clear_signal(dword handle)
{
    /* Polled mode has only one global signal */
    /* Otherwise implement clear of the signal specified by handle */
    polled_signal = 0;
}

/* This routine takes as an argument a handle that was returned by rtfs_port_alloc_signal() and a timeout value in millisec-
onds. It must block until timeout milliseconds have elapsed or  rtfs_port_set_signal() has been called. If the test succeeds
must return 0, if it times out it must return a non-zero value.  Only required for the supplied floppy disk and ide device
driver if the ide driver is running in interrupt mode. Otherwise leave this function as it is, it will not be used. */

int rtfs_port_test_signal(dword handle, int timeout)
{
#if (!POLLED_MODE)
    /* Implement timed test of the signal specified by handle here otherwise use the polled mode code provided below */
    return (-1);
#else
```

```
    /* This will work in polled mode */
dword last_time, end_time, curr_time;
    /* Polled mode signal wait with timeout. This is portable but the user must implement the routine    rtfs_port_get_
ticks(void)
    and the macro MILLISECONDS_PER_TICK. */

      /* polled mode only ever uses one signal at a time so we ignore the handle argument */
  if (timeout)
  {
        /* Covert milliseconds to ticks. If 0 set it to 2 ticks */
        timeout = timeout/MILLISECONDS_PER_TICK;
        if (!timeout)
            timeout = 2;
    last_time = end_time = 0;       // keep compiler happy
    curr_time = rtfs_port_get_ticks();
    end_time = curr_time + (dword) timeout;
    // Check for a wrap
    if (end_time < curr_time)
        {
       end_time = (dword) 0xffffffffL;
        }
    last_time = curr_time;
    do
    {
       // See if we timed out
       curr_time = rtfs_port_get_ticks();
       if (curr_time > end_time)
          break;
       // if wrap or clearing of clock then start count over
       if (curr_time < last_time)
          end_time = curr_time + (dword) timeout;
       last_time = curr_time;
    } while (polled_signal == 0);
  }
  if (polled_signal)
  {
     polled_signal -= 1;
     return(0);
  }
  else
  {
    return(-1);
  }
#endif
}
/*
This routine takes as an argument a handle that was returned by rtfs_port_alloc_signal(). It must set the signal such that a
subsequent  call to rtfs_port_test_signal() or a call currently blocked in rtfs_port_test_signal() will return success.
This routine is always called from the device driver interrupt service routine while the processor is executing in the inter-
rupt context.  Only required for the supplied floppy disk and ide device driver if the ide driver is running in interrupt mode.
Otherwise leave this function as it is, it will not be used. */

void rtfs_port_set_signal(dword handle)
{
    /* Implement set of the signal specified by handle here */
    /* This simple implementation is for polled mode */
    polled_signal += 1;
}

/*
This routine takes as an argument a sleeptime value in milliseconds. It must not return to the caller until at least sleeptime
```

```
milliseconds have elapsed. In a multitasking environment this call should yield the task cpu.
*/
void rtfs_port_sleep(int sleeptime)
{
    /* Implement simple task sleep call here */

    /* This simple implementation is for polled mode */
    /* Call the signaling system to timeout. This will in effect sleep */
    rtfs_port_clear_signal(0);
    rtfs_port_test_signal(0, sleeptime);
}

/* This routine takes no arguments and returns an unsigned long. The routine must return an unsigned long value that
will later be passed to rtfs_port_elapsed_check() to test if a given number of milliseconds or more have elapsed. A typical
implementation of this routine would read the system tick counter and return it as an unsigned long. ERTFS makes no as-
sumptions about the value that is returned.
*/

dword rtfs_port_elapsed_zero(void)
{
    return(rtfs_port_get_ticks());
}

/* This routine takes as arguments an unsigned long value that was returned by a previous call to rtfs_port_elapsed_zero()
and a timeout value in milliseconds. If “timeout” milliseconds have not elapsed it should return 0. If “timeout” milliseconds
have elapsed it should return 1. A typical implementation of this routine would read the system tick counter, subtract  the
zero value, scale the difference to milliseconds and compare that to timeout. If the scaled difference is greater or equal to
timeout it should return 1, if less than timeout it should return 0.
*/

int rtfs_port_elapsed_check(dword zero_val, int timeout)
{
dword curr_time;
    timeout = timeout/MILLISECONDS_PER_TICK;
    if (!timeout)
        timeout = 2;
    curr_time = rtfs_port_get_ticks();
    if ( (curr_time - zero_val) > (dword) timeout)
        return(1);
    else
        return(0);
}

/*
This function must return an unsigned long number that is unique to the currently executing task such that each time this
function is called from the same task it returns this same unique number. A typical implementation of this function would
get address of the current task control block, cast it to a long, and return it.
*/

dword rtfs_port_get_taskid(void)
{
    /* For an RTOS environment return a dword that is unique to this task. For example the address of its task control
block */
    /* Otherwise always return 1 For POLLOS */
    return(1);
}

/*
This routine requires porting if you wish to use the interactive test shell.  It must provide a line of input from the terminal
driver. The line must be NULL terminated and it must not contain ending newline or carriage return characters.
*/
```

```
void rtfs_port_tm_gets(byte *buffer)
{
/* Use Get's or some other console input function. If you have no console input function, then leave it blank */
/*     gets(buffer); */
}

/*
This routine requires porting if you wish to display messages to your console.  This routine must print a line of text from
the supplied buffer to the console. The output routine must not issue a carriage or linefeed command unless the text is
terminated with the appropriate control character (\n or \r).
*/

void rtfs_port_puts(byte *buffer)
{
/* Use cputs or some other console output function. If you have no console output function then leave it blank */
/*     cputs(buffer); */
}

/*
This function must exit the RTOS session and return to the user prompt. It is only necessary when an RTOS is running in-
side a shell environment like Windows. This function must be implemented if you are using the ERTFS command library in
an environment that will ever exit, for example if ERTFS is running with an RTOS in a Dos/Windows or Unix environment
you should put a call to your  RTOS's exit code.
Note: Most embedded systems never exit. If your application never exits, then please skip this section.
*/

void rtfs_port_exit(void)
{
/* Exit your application here. If you never exit then leave it blank. */
/*     exit(0); */
}
/*
When the system needs to date stamp a file it will call this routine to get the current time and date. YOU must modify the
shipped routine to support your hardware's time and date routines.
*/

DATESTR *pc_getsysdate(DATESTR * pd)
{
   word  year;     /* relative to 1980 */
   word  month;    /* 1 - 12 */
   word  day;      /* 1 - 31 */
   word  hour;
   word  minute;
   word  sec;      /* Note: seconds are 2 second/per. ie 3 == 6 seconds */

#if (0)
 /* This code will work if you have ANSI time functions. otherwise get  rid of it and look below where the time is wired to
3/28/8.
   You may modify that code to work in your environment. */
   struct tm *timeptr;
   time_t timer;

   time(&timer);
   timeptr = localtime(&timer);

   hour    =   (word) timeptr->tm_hour;
   minute  =   (word) timeptr->tm_min;
   sec =   (word) (timeptr->tm_sec/2);
   /* Date comes back relative to 1900 (eg 93). The pc wants it relative to
      1980. so subtract 80 */
```

```
    year  = (word) (timeptr->tm_year-80);
    month = (word) (timeptr->tm_mon+1);
    day   = (word) timeptr->tm_mday;
#else
    /* This code is useless but very generic */
    /* Hardwire for now */
    /* 7:37:28 PM */
    hour = 19;
    minute = 37;
    sec = 14;
    /* 3-28-88 */
    year = 8;       /* relative to 1980 */
    month = 3;      /* 1 - 12 */
    day = 28;       /* 1 - 31 */

#endif
    pd->time = (word) ( (hour << 11) | (minute << 5) | sec);
    pd->date = (word) ( (year << 9) | (month << 5) | day);

    return (pd);
}

/* This routine must establish an interrupt handler that will call the plain 'C' routine void mgmt_isr(void) when the chip's
management interrupt event occurs. The value of the argument 'irq' is the interrupt number that was put into the 82365
management interrupt selection register and is between 0 and 15. This is controlled by the constant "MGMT_INTER-
RUPT" which is defined in pcmctrl.c */
#if (INCLUDE_82365_PCMCTRL)

/* The routine mgmt_isr(0) must execute when the PCMCIA controller interrupts  this a sample interrupt service routine
that will do this, modify it to match your system's interrupt behavior */
void mgmt_isr(int);
void pcmcia_isr()
{
    mgmt_isr(0);
}

/* This routine must "hook" the interrupt so the above code is jumped to when the PCMCIA controller interrupts */
void hook_82365_pcmcia_interrupt(int irq)
{
}

#endif /* (INCLUDE_82365_PCMCTRL) */

#if (INCLUDE_IDE)
/* The routine ide_isr(controller_number) must execute when the IDE controller associated with that controller number
interrupts.  These are sample interrupt service routines that will do this, modify them to match your system's interrupt
behavior */

void ide_isr(int);
/* This is a sample interrupt service routine for controller 0 */
void ide_isr_0()
{
    ide_isr(0);
}

/* This is a sample interrupt service routine for controller 1 */
void ide_isr_1()
{
    ide_isr(1);
}
```

```
/* hook_ide_interrupt() is called with the interrupt number in the argument irq taken from the user's setting of pdr->interrupt_number in pc_ertfs_init(). Controller number is taken from the pdr->controller_number field as set in pc_ertfs_init() by the user. Hook_ide_interrupt() must establish an interrupt handler such that the plain 'C' function "void ide_isr(int controller_number)" is called when the IDE interrupt occurs. The argument to ide_isr() must be the controller number that was passed to hook_ide_interrupt(), this value is typically zero for single controller system.
*/

void hook_ide_interrupt(int irq, int controller_number)
{
}

#endif /* (INCLUDE_IDE) */

/*  This routine is called by the floppy disk device driver. It must establish an interrupt handler such that the plain 'C' function void  floppy_isr(void) is called when the floppy disk interrupt occurs.  The value in "irq" is always 6, this is the PC's standard mapping of  the floppy interrupt. If this is not correct for your system just ignore the irq argument.
*/
#if (INCLUDE_FLOPPY)
/* This is a sample floppy disk interrupt handler. Modify it to match your target system */
void floppy_isr(int);

void floppy_interrupt()
{
    floppy_isr(0);
}

void hook_floppy_interrupt(int irq)
{
}

#endif /* (INCLUDE_FLOPPY) */
```

## APPENDIX F. PORTIO.C

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS I/O porting layer file */

```
#include <rtfs.h>

/* All of the I/O in and out functions in this file call one of these four  macros to write or read the appropriate I/O location.
If t is convenient to use this scheme and it will work for you then please modify these macros to do the appropriate I/O
operation for the register. If this can't be accomplished just modify the appropriate routines manually.
*/

#define   RTFS_OUTBYTE(ADDRESS,VALUE)
#define   RTFS_INBYTE(ADDRESS) 0
#define   RTFS_OUTWORD(ADDRESS,VALUE)
#define   RTFS_INWORD(ADDRESS) 0
#define   RTFS_OUTDWORD(ADDRESS,VALUE)

/* The code below must be implemented for all targets */

/*rtfs_port_disable() and rtfs_port_enable()
  Used by a few device drivers, floppy, flash chip and pcmctrl,   you may ignore it if not being called, but first check to see if
floppy, flash chip and pcmctrl are still the only drivers using the functions */

/*This function must disable interrupts and return. */
void rtfs_port_disable(void)
{
}

/*This function must re-disable interrupts that were disabled via a call to
rtfs_port_disable(). */

void rtfs_port_enable(void)
{
}

#if (INCLUDE_82365_PCMCTRL)
/* These routines are required only if the 82365 pcmcia controller driver
   is included (pcmctrl.c)
 phys82365_to_virtual(PFBYTE * virt, dword phys)
 write_82365_index_register(byte value)
 write_82365_data_register(byte value)
 read_82365_data_register(void)
*/
/* This routine must take a physical linear 32 bit bus address passed in the "phys" argument and convert it to an address
addressable in the logical space of the CPU, returning that value in "virt". Two sample methods are provided, a flat version
where the virtual address is simply the "phys" address cast to a pointer and a second segmented version for X86 seg-
mented applications.
*/
void phys82365_to_virtual(PFBYTE * virt, dword phys)
{
      *virt = (PFBYTE) phys; /* example for flat unmapped address space */
}
/*
These routines write and read the 82365's index and data registers which, in a standard PC environment, are located in
I/O space at address 0x3E0 and 0x3E1. Non PC architectures typically map these as memory mapped locations some-
where high in memory such as 0xB10003E0 and 0xB10003E1.
*/
void write_82365_index_register(byte value)
{
```

```
    RTFS_OUTBYTE(0x3e0,value);  /* Defined for X86. Replace with your own */
}

void write_82365_data_register(byte value)
{
    RTFS_OUTBYTE(0x3e1,value);  /* Defined for X86. Replace with your own */
}

byte read_82365_data_register(void)
{
byte v;
    v = RTFS_INBYTE(0x3e1);  /* Defined for X86. Replace with your own */
    return (v);
}
#endif /* (INCLUDE_82365_PCMCTRL) */

/*
These routines are only required if INCLUDE_IDE is turned on (ide_drv.c)
byte ide_rd_status(dword register_file_address)
byte ide_rd_sector_count(dword register_file_address)
byte ide_rd_alt_status(dword register_file_address, int contiguous_io_mode)
byte ide_rd_error(dword register_file_address)
byte ide_rd_sector_number(dword register_file_address)
byte ide_rd_cyl_low(dword register_file_address)
byte ide_rd_cyl_high(dword register_file_address)
byte ide_rd_drive_head(dword register_file_address)
byte ide_rd_drive_address(dword register_file_address)
void ide_wr_dig_out(dword register_file_address, byte value)
void ide_wr_sector_count(dword register_file_address, byte value)
void ide_wr_sector_number(dword register_file_address, byte value)
void ide_wr_cyl_low(dword register_file_address, byte value)
void ide_wr_cyl_high(dword register_file_address, byte value)
void ide_wr_drive_head(dword register_file_address, byte value)
void ide_wr_command(dword register_file_address, byte value)
void ide_wr_feature(dword register_file_address, byte value)
unsigned long rtfs_port_bus_address(void * p)
void ide_insw(dword register_file_address, unsigned short *p, int nwords)
void ide_outsw(dword register_file_address, unsigned short *p, int nwords)

These routines are only required if INCLUDE_IDE and INCLUDE_UDMA are turned on
dword rtfs_port_ide_bus_master_address(int controller_number)
byte ide_rd_udma_status(dword bus_master_address)
void ide_wr_udma_status(dword bus_master_address, byte value)
byte ide_rd_udma_command(dword bus_master_address)
void ide_wr_udma_command(dword bus_master_address, byte value)
void ide_wr_udma_address(dword bus_master_address, dword bus_address)
*/

#if (INCLUDE_IDE)
/* These routines are required only if using the IDE device driver */
/*  This function must return the byte in location 7 (IDE_OFF_STATUS)
    of the ide register file at register_file_address */
byte ide_rd_status(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 7)));
}

/* This function must return the byte in location 2 (IDE_OFF_SECTOR_COUNT)
    of the ide register file at register_file_address */
byte ide_rd_sector_count(dword register_file_address)
{
```

```
    return(RTFS_INBYTE((word) (register_file_address + 2)));
}

/*  This function must return the byte in location 0x206 (IDE_OFF_ALT_STATUS)
    of the ide register file at register_file_address The offset in the
    register file may also be 14 rather then 0x206 if the device is running
    in contiguous I/O mode.
*/

byte ide_rd_alt_status(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 0x206)));
}
/* This function must return the byte in location 1 (IDE_OFF_ERROR)
    of the ide register file at register_file_address.
*/
byte ide_rd_error(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 1)));
}

/* This function must return the byte in location 3 (IDE_OFF_SECTOR_NUMBER)
    of the ide register file at register_file_address
*/
byte ide_rd_sector_number(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 3)));
}
/* This function must return the byte in location 4 (IDE_OFF_CYL_LOW)
    of the ide register file. register_file_address
*/
byte ide_rd_cyl_low(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 4)));
}
/*  This function must return the byte in location 5 (IDE_OFF_CYL_HIGH)
    of the ide register file. register_file_address
*/
byte ide_rd_cyl_high(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 5)));
}

/*  This function must return the byte in location 6 (IDE_OFF_DRIVE_HEAD)
    of the ide register file. register_file_address
*/
byte ide_rd_drive_head(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 6)));
}

/*  This function must return the byte in location 0x207 (IDE_OFF_DRIVE_ADDRESS)
    of the ide register file at register_file_address. The offset in the
    register file may also be 15 rather then 0x207 if the device is running
    in contiguous I/O mode.
*/
byte ide_rd_drive_address(dword register_file_address)
{
    return(RTFS_INBYTE((word) (register_file_address + 0x207)));
}
```

```
/* This function must place the byte in value at location 0x206 (IDE_OFF_ALT_STATUS) of the ide register file at
   register_file_address.The offset in the register file may also be 14 rather then 0x206 if the device is running in
   contiguous I/O mode.
*/

void ide_wr_dig_out(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 0x206), value);
}
/*    This function must place the byte in value at location
      2 (IDE_OFF_SECTOR_COUNT) of the ide register file at
      register_file_address
*/
void ide_wr_sector_count(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 2), value);
}

/*    This function must place the byte in value at location
      3 (IDE_OFF_SECTOR_NUMBER) of the ide register file at
      register_file_address
*/
void ide_wr_sector_number(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 3), value);
}

/* This function must place the byte in value at location
      4 (IDE_OFF_CYL_LOW) of the ide register file at
      register_file_address
*/
void ide_wr_cyl_low(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 4), value);
}
/* This function must place the byte in value at location
   5 (IDE_OFF_CYL_HIGH) of the ide register file at register_file_address
*/
void ide_wr_cyl_high(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 5), value);
}

/* This function must place the byte in value at location
   6 (IDE_OFF_DRIVE_HEAD) of the ide register file at register_file_address
*/
void ide_wr_drive_head(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 6), value);
}
/* This function must place the byte in value at location
      7 (IDE_OFF_COMMAND) of the ide register file at register_file_address */
void ide_wr_command(dword register_file_address, byte value)
{
      RTFS_OUTBYTE((word) (register_file_address + 7), value);
}

/*    This function must place the byte in value at location
      1 (IDE_OFF_FEATURE) of the ide register file at register_file_address
*/
void ide_wr_feature(dword register_file_address, byte value)
```

```
{
      RTFS_OUTBYTE((word) (register_file_address + 1), value);
}

/*    This function must read nwords 16 bit values from the data register at offset 0 of the ide register file and place them in
successive memory locations starting at p. Since large blocks of data are transferred from the drive in this way this routine
should be optimized. On x86 based systems the repinsw instruction should be used, on non x86 platforms the loop should
be as tight as possible.
*/

void ide_insw(dword register_file_address, unsigned short *p, int nwords)
{
   while (nwords—)
      *p++ = (word)RTFS_INWORD(register_file_address);
}
/*
      This function must write nwords 16 bit values to  the data
      register at offset 0 of the ide register file. The data is
      taken from successive memory locations starting at p. Since large
      blocks of data are transferred from the drive in this way this
      routine should be optimized. On x86 based systems the repoutsw
      instruction should be used, on non x86 platforms the loop should
      be as tight as possible.
*/
void ide_outsw(dword register_file_address, unsigned short *p, int nwords)
{
   while (nwords—)
      RTFS_OUTWORD(register_file_address, *src++);
}

#if (INCLUDE_UDMA)
/* These routines are required only if using an ultra-dma controller. */
/*    This function must determine if the specified controller is a PCI bus
      mastering IDE controller and if so return the location of the
      control and status region for that controller. If it is not a bus
      bus master controller it should return zero.
*/
dword rtfs_port_ide_bus_master_address(int controller_number)
{
      return (0); /* Must be implemented. 0 return val disables UDMA */
}

/*    This function must read the status byte value at location
      2 the bus master control region
*/
byte ide_rd_udma_status(dword bus_master_address)
{
byte value;
      value = RTFS_INBYTE((word) (bus_master_address + 2));
      return (value);
}

/*    This function must write the byte value to location
      2 of the bus master control region
*/
void ide_wr_udma_status(dword bus_master_address, byte value)
{
      RTFS_OUTBYTE((word) (bus_master_address + 2), value);
}

/*    This function must read the command byte value at location
```

```
        0 of the bus master control region
*/
byte ide_rd_udma_command(dword bus_master_address)
{
byte value;
        value = RTFS_INBYTE((word) bus_master_address);
        return (value);
}
/*      This function must write the byte value to location
        0 of the bus master control region
*/
void ide_wr_udma_command(dword bus_master_address, byte value)
{
        RTFS_OUTBYTE((word) bus_master_address, value);
}
/*      This function must write the byte dword to location
        4 of the bus master control region
*/
void ide_wr_udma_address(dword bus_master_address, dword bus_address)
{
        RTFS_OUTDWORD((word)(bus_master_address+4), bus_address);
}


/* This function must take a logical pointer and convert it to an
   unsigned long representation of its address on the system bus */
unsigned long rtfs_port_bus_address(void * p)
{
dword laddress;
    laddress = (dword) p;
    return(laddress);
}

#endif /* (INCLUDE_UDMA) */
#endif /*  (INCLUDE_IDE) */
```

## APPENDIX G: APPCMDSH.C

Contains many examples of calling the ERTFS-PRO API

EBS – ERTFS-PRO (Real Time File Manager) Copyright EBS, Inc. 2003

This is a representative sample of the ERTFS interactive command shell

*Note: This module contains many examples of calling the ERTFS-PRO API */

```
#include <rtfsapi.h>

DISPATCHER cmds[] =
   {
   { USTRING_TSTSHCMD_26, dohelp,  USTRING_TSTSHHELP_26 },          /*HELP */
   { USTRING_TSTSHCMD_01, docat, USTRING_TSTSHHELP_01 },            /* CAT PATH */
   { USTRING_TSTSHCMD_02, dochsize,  USTRING_TSTSHHELP_02},         /* CHSIZE FILENAME NEWSIZE */
   { USTRING_TSTSHCMD_03, docwd,  USTRING_TSTSHHELP_03 },           /*  CD PATH or CD for PWD */
   { USTRING_TSTSHCMD_12, dochkdsk,  USTRING_TSTSHHELP_12},         /* CHKDSK D: <0,1> */
   { USTRING_TSTSHCMD_13, doclose,  USTRING_TSTSHHELP_13 },         /* CLOSE FDNO */
   { USTRING_TSTSHCMD_14, docopy, USTRING_TSTSHHELP_14 },           /*  COPY PATH PATH */
   { USTRING_TSTSHCMD_15,dorm,  USTRING_TSTSHHELP_15 },             /*  DELETE PATH */
   { USTRING_TSTSHCMD_16, dodeltree,  USTRING_TSTSHHELP_16},        /* DELTREE PATH */
   { USTRING_TSTSHCMD_17, dodevinfo,  USTRING_TSTSHHELP_17},        /* " DEVINFO */
   { USTRING_TSTSHCMD_18, dodiff, USTRING_TSTSHHELP_18 },           /*  DIFF PATH PATH */
   { USTRING_TSTSHCMD_19, dols,  USTRING_TSTSHHELP_19 },            /* DIR [PATH] */
   { USTRING_TSTSHCMD_20, doselect,  USTRING_TSTSHHELP_20 }         ,/* DSKSEL D: */
   { USTRING_TSTSHCMD_21, echo,  USTRING_TSTSHHELP_21 },            /*  ECHO [args] */
   { USTRING_TSTSHCMD_23, dofillfile, USTRING_TSTSHHELP_23},        /* FILLFILE PATH PATTERN NTIMES */
   { USTRING_TSTSHCMD_24, doformat, USTRING_TSTSHHELP_24}           ,/* FORMAT
   { USTRING_TSTSHCMD_25, dogetattr, USTRING_TSTSHHELP_25}          ,/* GETATTR FILE" /
   { USTRING_TSTSHCMD_27, dolstop,  USTRING_TSTSHHELP_27  },        /* LSTOPEN  */
   { USTRING_TSTSHCMD_28, domkdir,  USTRING_TSTSHHELP_28 },         /* MKDIR PATH */
   { USTRING_TSTSHCMD_30, 0,  USTRING_TSTSHHELP_30 },                     /* QUIT */
   { USTRING_TSTSHCMD_31,  doread,  USTRING_TSTSHHELP_31 },         /*READ FDNO */
   { USTRING_TSTSHCMD_32,domv,  USTRING_TSTSHHELP_32 },             /* RENAME PATH NEWNAME */
   { USTRING_TSTSHCMD_33, dormdir,  USTRING_TSTSHHELP_33 },         /* RMDIR PATH */
   { USTRING_TSTSHCMD_34, rndop,  USTRING_TSTSHHELP_34 },           /* RNDOP PATH RECLEN */
   { USTRING_TSTSHCMD_35,  doseek,  USTRING_TSTSHHELP_35 },         /*SEEK FDNO RECORD */
   { USTRING_TSTSHCMD_36, dosetattr,  USTRING_TSTSHHELP_36},        /*SETATTR D:PATH
                                 RDONLY|HIDDEN|SYSTEM|ARCHIVE|NORMAL" */
   { USTRING_TSTSHCMD_37, dostat,  USTRING_TSTSHHELP_37 }           ,/*STAT PATH*/
   { USTRING_TSTSHCMD_38, dowrite,  USTRING_TSTSHHELP_38 },         /*"WRITE FDNO QUOTED DATA */
   { USTRING_TSTSHCMD_39, doregress,  USTRING_TSTSHHELP_39 },       /* REGRESSTEST D: */
   { USTRING_TSTSHCMD_40, domkhostdisk,  USTRING_TSTSHHELP_40 },    /*MKHOSTDISK win9xpath */
   { USTRING_TSTSHCMD_41, domkrom,  USTRING_TSTSHHELP_41 }          ,/*MKROM */
   { 0 }
   };


void  tst_shell(void)                     /* __fn__ */
{
   DISPATCHER *pcmd;
   int  agc = 0;
   byte *agv[20];
   int i;

   rtfs_memset((PFBYTE)rnds, 0, sizeof(RNDFILE)*MAXRND);

   for (i = 0 ; i < MAXRND; i++)
      rnds[i].fd = -1;
   dohelp(agc, agv);

/* Execute commands from the command table until the users types "QUIT" */
```

```
    pcmd = lex( &agc, &agv[0]);
    while (pcmd)
    {
        if (!pcmd->proc)
                return;
        i = pcmd->proc(agc, &agv[0]);
        pcmd = lex( &agc, &agv[0]);
    }
}


/* DSKSEL PATH */
int doselect(int agc, byte **agv)                       /*__fn__*/
{
    if (agc == 1)
    {
            if (!pc_set_default_drive(*agv))
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_03,PRFLG_NL); /* “Set Default Drive Failed” */
            return(0);
        }
        return(1);
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_04, PRFLG_NL); /* “Usage: DSKSELECT D: “ */
        return(0);
    }
}


/* RNDOP PATH RECLEN */
int rndop(int agc, byte **agv)                          /*__fn__*/
{
    RNDFILE *rndf;

    if (agc == 2)
    {
        rndf = fndrnd( -1);
        if (!rndf)
            {RTFS_PRINT_STRING_1(USTRING_TSTSH_05, PRFLG_NL);} /* “No more random file slots “ */
        else
        {
            rtfs_cs_strcpy(rndf->name, *agv);
            agv++;
            rndf->reclen  = (int)rtfs_atoi(*agv);
            if ((rndf->fd = po_open(rndf->name, (PO_BINARY|PO_RDWR|PO_CREAT),
                        (PS_IWRITE | PS_IREAD) ) ) < 0)
            {
                RTFS_PRINT_STRING_2(USTRING_TSTSH_06,rndf->name,PRFLG_NL); /* “Cant open :” */
                /* Note: rndf->fd is still -1 on error */
            }
            else
                return (1);
        }
    }
    RTFS_PRINT_STRING_1(USTRING_TSTSH_07,PRFLG_NL); /* “Usage: RNDOP D:PATH RECLEN” */
    return (0);
}
```

```
/* CLOSE fd */
int doclose(int agc, byte **agv)                    /*__fn__*/
{
   PCFD fd;
   RNDFILE *rndf;

   if (agc == 1)
   {
      fd = rtfs_atoi(*agv);
      rndf = fndrnd( fd );
      if (!rndf)
         {RTFS_PRINT_STRING_1(USTRING_TSTSH_08,PRFLG_NL);} /* “Cant find file” */
      else
      {
         if (po_close(fd) < 0)
             {RTFS_PRINT_STRING_1(USTRING_TSTSH_09,PRFLG_NL);} /* “Close failed” */
          else
          {
             rndf->fd = -1;
             return (1);
          }
      }
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_10,PRFLG_NL); /* “Usage: CLOSE fd” */
   return (0);
}

/* SEEK fd recordno */
int doseek(int agc, byte **agv)                     /*__fn__*/
{
   PCFD fd;
   RNDFILE *rndf;
   int recno;
   long foff;
   long seekret;

   if (agc == 2)
   {
      fd = rtfs_atoi(*agv);
      rndf = fndrnd(fd);
      if (!rndf)
         {RTFS_PRINT_STRING_1(USTRING_TSTSH_11,PRFLG_NL);} /* “Cant find file” */
      else
      {
         agv++;
         recno = rtfs_atoi(*agv);
         foff = (long) recno * rndf->reclen;

         if (foff !=(seekret = po_lseek(fd, foff, PSEEK_SET ) ))
         {
             RTFS_PRINT_STRING_1(USTRING_TSTSH_12, PRFLG_NL); /* “Seek operation failed “ */
         }
         else
            return (1);
      }
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_13,PRFLG_NL); /* “Usage: SEEK fd recordno” */
   return (0);
}

/* READ fd */
```

```
int doread(int agc, byte **agv)                         /*__fn__*/
{
   PCFD fd;
   RNDFILE *rndf;
   int res;

   if (agc == 1)
   {
      fd = (PCFD) rtfs_atoi(*agv);
      rndf = fndrnd(  fd );
      if (!rndf)
         {RTFS_PRINT_STRING_1(USTRING_TSTSH_14,PRFLG_NL);} /* “Cant find file” */
      else
      {
         if ( (res = po_read(fd,(byte*)rndf->buff,(word)rndf->reclen)) != rndf->reclen)
         {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_15, PRFLG_NL); /* “Read operation failed “ */
         }
         else
         {
            RTFS_PRINT_STRING_2(USTRING_SYS_NULL, rndf->buff,PRFLG_NL);
            return (1);
         }
      }
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_16,PRFLG_NL); /* “Usage: READ fd” */
   return (0);
}

/* WRITE fd “data” */
int dowrite(int agc, byte **agv)                        /*__fn__*/
{
   PCFD fd;
   RNDFILE *rndf;
   int res;

   if (agc == 2)
   {
      fd = rtfs_atoi(*agv++);
      rndf = fndrnd(  fd );
      if (!rndf)
         {RTFS_PRINT_STRING_1(USTRING_TSTSH_17,PRFLG_NL);} /* “Cant find file” */
      else
      {
         pc_cppad((byte*)rndf->buff,(byte*) *agv,(int) rndf->reclen);
         rndf->buff[rndf->reclen] = CS_OP_ASCII(‘\0’);
         if ( (res = po_write(fd,(byte*)rndf->buff,(word)rndf->reclen)) != rndf->reclen)
         {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_18, PRFLG_NL); /* “Write operation failed “ */
         }
         else
         {
            return (1);
         }
      }
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_19,PRFLG_NL); /* “Usage: WRITE fd data “ */
   return (0);
}

/* LSTOPEN */
```

```
int dolstop(int agc, byte **agv)                        /*__fn__*/
{
   int i;
   RTFS_ARGSUSED_INT(agc);
   RTFS_ARGSUSED_PVOID((PFVOID)agv);

   for (i = 0 ; i < MAXRND; i++)
   {
      if (rnds[i].fd != -1)
           {
                RTFS_PRINT_LONG_1((dword)rnds[i].fd, 0);
          RTFS_PRINT_STRING_2(USTRING_TSTSH_30, rnds[i].name, PRFLG_NL);
           }
   }
   return (1);
}

/* MKDIR PATH */
int domkdir(int agc, byte **agv)                        /*__fn__*/
{
   if (agc == 1)
   {
      if (pc_mkdir(*agv))
          return (1);
      else
         return(0);
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_20,PRFLG_NL); /* “Usage: MKDIR D:PATH” */
   return (0);
}

/* RMDIR PATH */
int dormdir(int agc, byte **agv)                        /*__fn__*/
{

   if (agc == 1)
   {
      if (pc_rmdir(*agv))
          return (1);
      else
         return(0);
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_21,PRFLG_NL); /* “Usage: RMDIR D:PATH\n” */
   return (0);
}

/* DELTREE PATH */
int dodeltree(int agc, byte **agv)                      /*__fn__*/
{

   if (agc == 1)
   {
      if (pc_deltree(*agv))
          return (1);
      else
         return(0);
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_22,PRFLG_NL); /* “Usage: DELTREE D:PATH” */
   return (0);
}
```

```
/* DELETE PATH */
byte rmfil[EMAXPATH];
byte rmpath[EMAXPATH];
byte tfile[EMAXPATH];
byte _rmpath[EMAXPATH];
int dorm(int agc, byte **agv)                          /*__fn__*/
{
   DSTAT statobj;

   if (agc == 1)
   {
      /* Get the path */
      rtfs_cs_strcpy(&rmfil[0],*agv);

      if (pc_gfirst(&statobj, &rmfil[0]))
      {
         do
         {
            /* Construct the path to the current entry ASCII 8.3 */
            pc_ascii_mfile((byte*) &tfile[0],(byte*) &statobj.fname[0], (byte*) &statobj.fext[0]);
            pc_mpath((byte*)&_rmpath[0],(byte*) &statobj.path[0],(byte*)&tfile[0] );
                     /* Convert to native character set if needed */
                     CS_OP_ASCII_TO_CS_STR(&_rmpath[0],&rmpath[0]);
            if (!pc_isdir(rmpath) && !pc_isvol(rmpath))
            {
               RTFS_PRINT_STRING_2(USTRING_TSTSH_23,rmpath,PRFLG_NL); /* “deleting  —> “ */
               if (!pc_unlink( rmpath ) )
                  RTFS_PRINT_STRING_2(USTRING_TSTSH_24,rmpath,PRFLG_NL); /* “Can not delete: “ */
            }
         }  while(pc_gnext(&statobj));
         pc_gdone(&statobj);
      }
      return(1);
   }
   RTFS_PRINT_STRING_1(USTRING_TSTSH_25,PRFLG_NL); /* “Usage: DELETE D:PATH” */
   return(0);
}

/* RENAME PATH NEWNAME */
int domv(int agc, byte **agv)                          /*__fn__*/
{
   if (agc == 2)
   {
      if (pc_mv(*agv , *(agv+1)))
          return (1);
      else
         return(0);
   }

   RTFS_PRINT_STRING_1(USTRING_TSTSH_26,PRFLG_NL); /* “Usage:  RENAME PATH NEWNAME” */
   return (0);
}


/* CD PATH */
int docwd(int agc, byte **agv)                          /*__fn__*/
{
   byte lbuff[100];

   if (agc == 1)
   {
```

```
        if (pc_set_cwd(*agv))
            return(1);
        else
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_27,PRFLG_NL); /* “Set cwd failed” */
            return(0);
        }
    }
    else
    {
        if (pc_pwd(rtfs_strtab_user_string(USTRING_SYS_NULL),lbuff))
        {
            RTFS_PRINT_STRING_2(USTRING_SYS_NULL,lbuff,PRFLG_NL);
            return(1);
        }
        else
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_28,PRFLG_NL); /* “Get cwd failed” */
            return(0);
        }
    }
}

/* DIR PATH */
int dols(int agc, byte **agv)                           /*__fn__*/
{
    int fcount;
    int doit, addwild;
    byte *ppath,*p;
    dword blocks_total, blocks_free;
    dword nfree;
    byte null_str[2];
    ppath = 0;

       addwild = 0;
    if (agc == 1)
      {
       ppath = *agv;
            p = ppath;
            /* If the second char is ‘:’ and the third is ‘\0’ then we have
               D: and we will convert to D:*.* */
            CS_OP_INC_PTR(p);
            if (CS_OP_CMP_ASCII(p,’:’))
            {
                  CS_OP_INC_PTR(p);
                  if (CS_OP_IS_EOS(p))
                        addwild = 1;
            }
      }
    else
    {
            null_str[0] = null_str[1] = 0; /* Works for all char sets */
        /* get the working dir of the default dir */
        if (!pc_pwd(null_str,(byte *)shell_buf))
            {
             RTFS_PRINT_STRING_1(USTRING_TSTSH_29,PRFLG_NL); /* “PWD Failed “ */
            return(1);
            }
        ppath = (byte *)shell_buf;
            p = ppath;
```

```
        /* If not the root add a \\ */
            doit = 1; /* Add \\ if true */
            if (CS_OP_CMP_ASCII(p,'\\'))
            {
                CS_OP_INC_PTR(p);
                if (CS_OP_IS_EOS(p))
                    doit = 0;
            }
            if (doit)
            {
                p = CS_OP_GOTO_EOS(p);
                CS_OP_ASSIGN_ASCII(p,'\\');
                CS_OP_INC_PTR(p);
                CS_OP_TERM_STRING(p);
            }
            addwild = 1;
    }
      if (addwild)
      {
            p = ppath;
            p = CS_OP_GOTO_EOS(p);
            /* Now tack on *.* */
            CS_OP_ASSIGN_ASCII(p,'*');
            CS_OP_INC_PTR(p);
            /* Add .* for non vfat */
            CS_OP_ASSIGN_ASCII(p,'.');
            CS_OP_INC_PTR(p);
            CS_OP_ASSIGN_ASCII(p,'*');
            CS_OP_INC_PTR(p);
            CS_OP_TERM_STRING(p);
      }


    /* Now do the dir */
    fcount = pc_seedir(ppath);

   /* And print the bytes remaining */
    nfree = pc_free(ppath, &blocks_total, &blocks_free);
/* printf( "       %-6d File(s)   %-8ld KBytes free\n", fcount, blocks_free/2 ); */

      RTFS_PRINT_STRING_1(USTRING_TSTSH_30, 0); /* "       " */
    RTFS_PRINT_LONG_1((dword) fcount, 0);
    RTFS_PRINT_STRING_1(USTRING_TSTSH_31, 0); /* " File(s) " */
    RTFS_PRINT_LONG_1((dword) blocks_free/2, 0);
    RTFS_PRINT_STRING_1(USTRING_TSTSH_32, PRFLG_NL); /* " Blocks Free " */
    return(1);
}

/* List directory, and return number of matching file */
int pc_seedir(byte *path)                               /*__fn__*/
{
    int fcount = 0;
    DSTAT statobj;
      byte display_buffer[100];
    /* Get the first match */
    if (pc_gfirst(&statobj, path))
    {
        while (TRUE)
        {
            fcount++;
```

```
                rtfs_print_format_dir(display_buffer, &statobj);

            /* Get the next */
            if (!pc_gnext(&statobj))
                break;

        }

        /* Call gdone to free up internal resources used by statobj */
        pc_gdone(&statobj);
    }
    return(fcount);
}


int doregress(int agc, byte **agv)
{


    if (agc == 1)
      {
            pc_regression_test(*agv);
            return(1);
      }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_110,PRFLG_NL); /*  "Usage: REGRESSTEST D:" */
        return(0);
    }
}

/* CAT PATH */
int docat(int agc, byte **agv)                          /*__fn__*/
{
    PCFD fd;
    int nread;

    if (agc == 1)
    {
        if ((fd = po_open(*agv, (word)(PO_BINARY|PO_RDONLY),(word) (PS_IWRITE | PS_IREAD) ) ) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_33, *agv,PRFLG_NL); /* "Cant open " */
            return(0);
        }
        else
        {
            do
            {
                nread = po_read(fd,(byte*)shell_buf,512);
                    shell_buf[nread] = '\0';
                RTFS_PRINT_STRING_2(USTRING_SYS_NULL,shell_buf,0);
            }while(nread > 0);

            po_close(fd);
            return(1);
        }
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_34,PRFLG_NL); /* "Usage: CAT PATH" */
        return(0);
```

```
    }
}



/*  */
int dochkdsk(int agc, byte **agv)                       /*__fn__*/
{
int write_chains;
CHKDISK_STATS chkstat;
    if (agc != 2)
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_35,PRFLG_NL); /* “Usage: CHKDSK DRIVE: WRITECHAINS” */
        RTFS_PRINT_STRING_1(USTRING_TSTSH_36,PRFLG_NL); /* “Example:CHKDSK A: 1” */
        RTFS_PRINT_STRING_1(USTRING_TSTSH_37,PRFLG_NL); /* “Runs chkdsk on A: writes lost chains” */
        RTFS_PRINT_STRING_1(USTRING_TSTSH_38,PRFLG_NL); /* “Example:CHKDSK A: 0” */
        RTFS_PRINT_STRING_1(USTRING_TSTSH_39,PRFLG_NL); /* “Runs chkdsk on A: does not write lost chains” */
        return(0);
    }
    write_chains = (int) rtfs_atoi( *(agv+1) );
    pc_check_disk(*agv, &chkstat, 1, write_chains, write_chains);
    return(0);
}


/* CHSIZE PATH NEWSIZE */
int dochsize(int agc, byte **agv)                       /*__fn__*/
{
    PCFD fd;
    long newsize;

    if (agc == 2)
    {
        if ((fd = po_open(*agv, (word)(PO_BINARY|PO_WRONLY),(word) (PS_IWRITE | PS_IREAD) ) ) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_40, *agv ,PRFLG_NL); /* “Cant open file: “ */
            return(0);
        }
        newsize = rtfs_atol( *(agv+1) );
        if (po_chsize(fd, newsize) != 0)
            RTFS_PRINT_STRING_1(USTRING_TSTSH_41, PRFLG_NL); /* “Change size function failed” */
        po_close(fd);
        return(1);
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_42,PRFLG_NL); /* “Usage: CHSIZE PATH NEWSIZE” */
        return(0);
    }
}


/* COPY PATH PATH */
int docopy(int agc, byte **agv)                         /*__fn__*/
{
    PCFD in_fd;
    PCFD out_fd;
    word nread;
    BOOLEAN forever = TRUE;         /* use this for while(forever) to quiet anal compilers */
```

```
    if (agc == 2)
    {
        if ((in_fd = po_open(*agv,(word) (PO_BINARY|PO_RDONLY),(word) (PS_IWRITE | PS_IREAD) ) ) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_43, *agv,PRFLG_NL); /* “Cant open file: “ */
            return(0);
        }
        if ((out_fd = po_open(*(agv+1),(word) (PO_BINARY|PO_WRONLY|PO_CREAT),(word) (PS_IWRITE | PS_IREAD) )
) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_44, *(agv+1),PRFLG_NL); /* “Cant open file” */
            return(0);
        }
        while (forever)
        {
            nread = (word)po_read(in_fd,(byte*)shell_buf, 1024);
            if (nread > 0 && nread != (word)~0)
            {
                if (po_write(out_fd,(byte*)shell_buf,(word)nread) != (int)nread)
                {
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_45,PRFLG_NL); /* “Write failure “ */
                    break;
                }
            }
            else
                break;
        }
        po_close(in_fd);
        po_close(out_fd);
        return(1);
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_46,PRFLG_NL); /* “Usage: COPY FROMPATH TOPATH\n” */
        return(0);
    }
}

/* DIFF PATH PATH */
int dodiff(int agc, byte **agv)                         /*__fn__*/
{
    PCFD in_fd;
    PCFD in_fd1;
    int nread;
    int nread1;
    int i;
    byte buff[256];
    byte buff1[256];
    BOOLEAN forever = TRUE;         /* use this for while(forever) to quiet anal compilers */

    if (agc == 2)
    {
        if ((in_fd = po_open(*agv, (PO_BINARY|PO_RDONLY), (PS_IWRITE | PS_IREAD) ) ) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_47, *agv,PRFLG_NL); /* “Cant open file: “ */
            return(0);
        }
        if ((in_fd1 = po_open(*(agv+1), (PO_BINARY|PO_RDONLY), (PS_IWRITE | PS_IREAD) ) ) < 0)
        {
            RTFS_PRINT_STRING_2(USTRING_TSTSH_48, *(agv+1),PRFLG_NL); /* “Cant open file” */
            return(0);
```

```
        }
        while (forever)
        {
            nread = po_read(in_fd,(byte*)buff,(word)255);
            if (nread > 0)
            {
                nread1 = po_read(in_fd1,(byte*)buff1,(word)nread);
                if (nread1 != nread)
                {
difffail:
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_49,PRFLG_NL);     /* “Files are different” */
                    po_close (in_fd);
                    po_close (in_fd);
                    return(0);
                }
                for(i = 0; i < nread; i++)
                {
                    if (buff[i] != buff1[i])
                        goto difffail;
                }
            }
            else
                break;
        }
        nread1 = po_read(in_fd1,(byte*)buff1,(word)nread);
        if (nread1 <= 0)
            {
                        RTFS_PRINT_STRING_2(USTRING_TSTSH_50,*agv,0); /* “File: “ */
                        RTFS_PRINT_STRING_2(USTRING_TSTSH_51,*(agv+1),0); /* “ And File: “ */
                        RTFS_PRINT_STRING_1(USTRING_TSTSH_52,PRFLG_NL); /* “ are the same” */
                    }
        else
        {
            {
                        RTFS_PRINT_STRING_2(USTRING_TSTSH_53, *(agv+1), 0); /* “File: “ */
                        RTFS_PRINT_STRING_2(USTRING_TSTSH_54, *agv, PRFLG_NL); /* “ is larger than File: “ */
                    }
            goto difffail;
        }
        po_close(in_fd);
        po_close(in_fd1);
        return(1);
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_55, PRFLG_NL); /* “Usage: DIFF PATH PATH “ */
    }
     return (0);
}


/* FILLFILE PATH PATTERN NTIMES */
int dofillfile(int agc, byte **agv)                         /*__fn__*/
{
    PCFD out_fd;
    byte  workbuf[255];
    word bufflen;
    int ncopies;

    if (agc == 3)
    {
```

```
        ncopies = (int) rtfs_atoi( *(agv+2) );
        if (!ncopies)
            ncopies = 1;

        if ((out_fd = po_open(*agv,(word) (PO_BINARY|PO_WRONLY|PO_CREAT),(word) (PS_IWRITE | PS_IREAD) ) ) <
0)
        {
           RTFS_PRINT_STRING_2(USTRING_TSTSH_56, *agv, PRFLG_NL); /* “Cant open file” */
           return(0);
        }
        rtfs_cs_strcpy(workbuf, *(agv+1));
        rtfs_strcat(workbuf, “\r\n”);
        bufflen = (word) rtfs_strlen(workbuf);

        while (ncopies—)
        {
           if (po_write(out_fd,(byte*)workbuf,(word)bufflen) != (int)bufflen)
           {
              RTFS_PRINT_STRING_1(USTRING_TSTSH_57, PRFLG_NL); /* “Write failure” */
              po_close(out_fd);
              return(0);
           }
        }
        po_close(out_fd);
        return(1);
    }
    else
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_58, PRFLG_NL); /* “Usage: FILLFILE PATH PATTERN NTIMES” */
        return(0);
    }
}

/* STAT PATH */
int dostat(int agc, byte **agv)                         /*__fn__*/
{
struct stat st;
byte display_buffer[100];
    if (agc == 1)
    {
        if (pc_stat(*agv, &st)==0)
        {
                rtfs_print_format_stat(display_buffer, &st);
           RTFS_PRINT_STRING_1(USTRING_TSTSH_59, 0); /* “MODE BITS :” */
           if (st.st_mode&S_IFDIR)
              RTFS_PRINT_STRING_1(USTRING_TSTSH_60, 0); /* “S_IFDIR|” */
           if (st.st_mode&S_IFREG)
              RTFS_PRINT_STRING_1(USTRING_TSTSH_61, 0); /* “S_IFREG|” */
           if (st.st_mode&S_IWRITE)
              RTFS_PRINT_STRING_1(USTRING_TSTSH_62, 0); /* “S_IWRITE|” */
           if (st.st_mode&S_IREAD)
              RTFS_PRINT_STRING_1(USTRING_TSTSH_63, 0); /* “S_IREAD\n” */
                 RTFS_PRINT_STRING_1(USTRING_SYS_NULL, PRFLG_NL); /* “” */

           return (1);
        }
        else
        {
           RTFS_PRINT_STRING_1(USTRING_TSTSH_65, PRFLG_NL); /* “FSTAT failed” */
           return(0);
        }
```

```
    }
    RTFS_PRINT_STRING_1(USTRING_TSTSH_66, PRFLG_NL); /* “Usage: FSTAT D:PATH” */
    return (0);
}

/* FORMAT D:*/
int doformat(int agc, byte **agv)                       /*__fn__*/
{
byte buf[10];
byte working_buffer[100];
DDRIVE *pdr;
int driveno;
dword partition_list[3];
byte path[10];
DEV_GEOMETRY geometry;

    RTFS_ARGSUSED_INT(agc);
    RTFS_ARGSUSED_PVOID((PFVOID)agv);


    /* “Enter the drive to format as A:, B: etc “ */
    rtfs_print_prompt_user(UPROMPT_TSTSH3, path);
    driveno = pc_parse_raw_drive(path);
    if (driveno == -1 || !(pdr = pc_drno_to_drive_struct(driveno)))
    {
inval:
        /*  “Invalid drive selection to format, press return” */
        rtfs_print_prompt_user(UPROMPT_TSTSH4, working_buffer);
        return(-1);
    }

    if ( !(pdr->drive_flags&DRIVE_FLAGS_VALID) )
        goto inval;

    // int driveno, BOOLEAN ok_to_automount, BOOLEAN raw_access_requested, BOOLEAN call_crit_err
    // call it twice to clear change conditions
    if (!check_media(driveno, FALSE, TRUE, FALSE))
    if (!check_media(driveno, FALSE, TRUE, FALSE))
    {
        /* “Format - check media failed. Press return” */
        rtfs_print_prompt_user(UPROMPT_TSTSH5, working_buffer);
        return(-1);
    }

    /* This must be called before calling the later routines */
    if (!pc_get_media_parms(path, &geometry))
    {
        /* “Format: get media geometry failed. Press return” */
        rtfs_print_prompt_user(UPROMPT_TSTSH6, working_buffer);
        return(-1);
    }

    /* Call the low level media format. Do not do this if formatting a
       volume that is the second partition on the drive */
    /* “Format: Press Y to format media “ */
    rtfs_print_prompt_user(UPROMPT_TSTSH7, buf);
       if (tstsh_is_yes(buf))
    {
        RTFS_PRINT_STRING_1(USTRING_TSTSH_67, PRFLG_NL); /* “Calling media format” */
        if (!pc_format_media(path, &geometry))
        {
```

```
            /* “Format: Media format failed. Press return” */
            rtfs_print_prompt_user(UPROMPT_TSTSH8, working_buffer);
            return(-1);
        }
    }

    /* Partition the drive if it needs it */
    if ( pdr->drive_flags&DRIVE_FLAGS_PARTITIONED )
    {
        /* “Format: Press Y to partition media “ */
        rtfs_print_prompt_user(UPROMPT_TSTSH9, buf);
            if (tstsh_is_yes(buf))
        {
                if (geometry.dev_geometry_lbas)
                {
                    do
                    {
                    /* “Format: Press Y to USE LBA formatting, N to use CHS “ */
                    rtfs_print_prompt_user(UPROMPT_TSTSH10, buf);
                    }
                    while (!tstsh_is_yes(buf) && !tstsh_is_no(buf));

                 if (tstsh_is_no(buf))
                    {
                        geometry.dev_geometry_lbas = 0;
                    }
                }

                if (geometry.dev_geometry_lbas)
                {
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_68, 0); /* “The drive is contains this many logical blocks “
*/
                    RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_lbas, PRFLG_NL);
                /* “Format: Select the number of lbas for the first partition :” */
                rtfs_print_prompt_user(UPROMPT_TSTSH11, working_buffer);
                    partition_list[0]  = (dword)rtfs_atol(working_buffer);
                    partition_list[1]  = 0;

                if (partition_list[0] != geometry.dev_geometry_lbas)
                    {
                        RTFS_PRINT_STRING_1(USTRING_TSTSH_69, 0); /* “This many logical blocks remain” */
                    RTFS_PRINT_LONG_1((dword) (geometry.dev_geometry_lbas - partition_list[0]), PRFLG_NL);
                        /* “Format: Select the number of lbas for the second partition :” */
                         rtfs_print_prompt_user(UPROMPT_TSTSH12, working_buffer);
                    partition_list[1]  = (dword)rtfs_atol(working_buffer);
                        partition_list[2]  = 0;
                }
                    if ((partition_list[0] == 0) || ((dword)(partition_list[0]+partition_list[1])
                          > geometry.dev_geometry_lbas))
                    {
                        /* “Format: Bad input for partition values. Press return” */
                       rtfs_print_prompt_user(UPROMPT_TSTSH13, working_buffer);
                    return(-1);
                   }
                }
                else
                {
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_70, 0); /* “The drive contains this many cylinders” */
                    RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_cylinders, PRFLG_NL);
                    /* “Format: Select the number of cyls for the first partition :” */
                rtfs_print_prompt_user(UPROMPT_TSTSH14, working_buffer);
```

```
                    partition_list[0]  = (word)rtfs_atoi(working_buffer);
                    partition_list[1]  = 0;

                    if (partition_list[0] != geometry.dev_geometry_cylinders)
                {
                    RTFS_PRINT_STRING_1(USTRING_TSTSH_71, 0); /* “There are this many cylinders remainng” */
                        RTFS_PRINT_LONG_1((dword) geometry.dev_geometry_cylinders - partition_list[0], PRFLG_NL);
                    /* “Format: Select the number of cyls for the second partition :” */
                rtfs_print_prompt_user(UPROMPT_TSTSH15, working_buffer);
                    partition_list[1]  = (dword)rtfs_atol(working_buffer);
                        partition_list[2]  = 0;
                    }
                    if ((partition_list[0] == 0) || ((dword)(partition_list[0]+partition_list[1])
                        > geometry.dev_geometry_cylinders))
                {
                        /* “Format: Bad input for partition values. Press return” */
                       rtfs_print_prompt_user(UPROMPT_TSTSH13, working_buffer);
                        return(-1);
                   }
                }
            if (!pc_partition_media(path, &geometry, &partition_list[0]))
             {
                   /* “Format: Media partition failed. Press return” */
              rtfs_print_prompt_user(UPROMPT_TSTSH16, working_buffer);
               return(-1);
                }
         }
        }

        /* Put the DOS format */
        /* “Format: Press Y to format the volume “ */
        rtfs_print_prompt_user(UPROMPT_TSTSH17, buf);
        if (tstsh_is_yes(buf))
        {
        if (!pc_format_volume(path, &geometry))
              {
                 /* “Format: Format volume failed. Press return” */
             rtfs_print_prompt_user(UPROMPT_TSTSH18, working_buffer);
                  return(-1);
              }
        }
        return (0);
}


/* GETATTR PATH*/
int dogetattr(int agc, byte **agv)                              /*__fn__*/
{
byte attr;

    if (agc == 1)
    {
        if (pc_get_attributes(*agv, &attr))
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_72,0); /* “Attributes: “ */
            if (attr & ARDONLY)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_73,0); /* “ARDONLY|” */
            if (attr & AHIDDEN)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_74,0); /* “AHIDDEN|” */
            if (attr & ASYSTEM)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_75,0); /* “ASYSTEM|” */
```

```
            if (attr & AVOLUME)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_76,0); /* “AVOLUME|” */
            if (attr & ADIRENT)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_77,0); /* “ADIRENT|” */
            if (attr & ARCHIVE)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_78,0); /* “ARCHIVE|” */
            if (attr == ANORMAL)
                RTFS_PRINT_STRING_1(USTRING_TSTSH_79,0); /* “NORMAL FILE (No bits set)” */
            RTFS_PRINT_STRING_1(USTRING_SYS_NULL, PRFLG_NL); /* “” */
            return(0);
        }
        else
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_81, PRFLG_NL); /* “get attributes failed” */
            return(0);
        }
    }
    RTFS_PRINT_STRING_1(USTRING_TSTSH_82, PRFLG_NL); /* “Usage: GETATTR D:PATH” */
    return (0);
}

/* GETATTR PATH*/
int dosetattr(int agc, byte **agv)                          /*__fn__*/
{
byte attr;

    attr = 0;
    if (agc == 2)
    {
        if (!pc_get_attributes(*agv, &attr))
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_83, PRFLG_NL); /* “Can not get attributes” */
            return(0);
        }

        if (rtfs_cs_strcmp(*(agv+1),rtfs_strtab_user_string(USTRING_TSTSH_84))==0) /* “RDONLY” */
            attr |= ARDONLY;
        else if (rtfs_cs_strcmp(*(agv+1),rtfs_strtab_user_string(USTRING_TSTSH_85))==0) /* “HIDDEN” */
            attr |= AHIDDEN;
        else if (rtfs_cs_strcmp(*(agv+1),rtfs_strtab_user_string(USTRING_TSTSH_86))==0) /* “SYSTEM” */
            attr |= ASYSTEM;
        else if (rtfs_cs_strcmp(*(agv+1),rtfs_strtab_user_string(USTRING_TSTSH_87))==0) /* “ARCHIVE” */
            attr |= ARCHIVE;
        else if (rtfs_cs_strcmp(*(agv+1),rtfs_strtab_user_string(USTRING_TSTSH_88))==0) /* “NORMAL” */
            attr =  ANORMAL;
        else
            goto usage;

        if (pc_set_attributes(*agv, attr))
            return(0);
        else
        {
            RTFS_PRINT_STRING_1(USTRING_TSTSH_89, PRFLG_NL); /* “Set attributes failed” */
            return(0);
        }
    }
usage:
    RTFS_PRINT_STRING_1(USTRING_TSTSH_90, PRFLG_NL); /* “Usage: SETATTR D:PATH RDONLY|HIDDEN|SYST
EM|ARCHIVE|NORMAL” */

    return (0);
```

```
}

/* DEVINFO */
int dodevinfo(int agc, byte **agv)                    /*__fn__*/
{
    RTFS_ARGSUSED_INT(agc);
    RTFS_ARGSUSED_PVOID((PFVOID)agv);

        print_device_names();

    return (1);
}
```

## APPENDIX H: ERTFS COMMAND SHELL COMMAND REFERENCE

**Summary of Command Shell Command References**

FORMAT - Format a disk.
CHKDSK - Perform a check disk procedure on a drive
DEVINFO - Print the names of all disks and their drive designators
MKHOSTDISK - Duplicate a windows or NT subdirectory inside on an ERTFS file system
MKROM - Create a ROMDISK Image
DIR - Print a directory listing
MKDIR - Create a directory
RENAME - Rename a file
RMDIR - Delete a directory
DELETE - Delete a file
CAT - Display contents of a file
CD - Set or display working directory
COPY - Copy a file to another
DELTREE - Delete a directory and its descendants
DIFF - Compare two files
DSKSEL - Set default drive
FILLFILE - Create a file and fill it with a pattern
GETATTR - Print a file's attributes
SETATTR - Change a file's attributes
STAT - Print file properties
CHSIZE - Truncate or extend a file
RNDOP - Open a random access file
CLOSE - Close a random access file
READ - Read and display a random access record
WRITE - Write data to a random access file
SEEK - Seek to a record in a random access file
HELP - Display all commands
QUIT - Exit the command shell

**FORMAT** - *Format a disk.* This routine will prompt you for the disk letter of the device that you would like to format. It prompts you first for the drive ID, and then asks if you would like to perform low level media formatting, disk partitioning and DOS volume formatting. All ERTFS volumes and writable media may be initialized this way.

**Example:** FORMAT

Enter the drive to format as A:, B: etc : C:

Press Y to format media : Y

Press Y to partition media : Y

The drive is contains 1024 cylinders

Select the number of cyls for first partition : 1024

Press Y to format the volume: Y

---

**MKHOSTDISK** - (command shell running in windows test environment only). This command gives the user a method to duplicate a native windows or NT subdirectory inside an ERTFS file system volume. The file system is formatted like a disk volume but it resides in a single Win/NT file named HOSTDISK.DAT. This tool is useful for populating an ERTFS volume that you may then experiment with.

Another important use of this tool is to allow the import of a whole WINDOWS/NT subtree that can later be emitted as an embeddable rom disk image. (See the MKROM command).

**Example:** MKHOSTDISK C:\MYROMDISK

---

**MKROM** - (command shell running in windows test environment only). This command takes the contents of a HOSTDISK volume (see MKHOSTDISK) and emits a file named drromdsk.h in the current Windows directory that may be used by the rom-disk driver, to provide an exact, read only image of that volume in the target system. This command, when used along with the MKHOSTDISK command allows you to create an exact rom image of a Windows/NT subdirectory, and it's descendants in the embedded system.

**Example:** MKROM

---

**CHKDSK** - *Perform a check disk procedure on a drive*

**Example:** CHKDSK C: 0- check drive C: don't write lost chains.

CHKDSK C: 1 - check drive C: write lost chains to .CHK files

---

**DIR** – *Print a directory listing*

**Example:** DIR *.*

---

**DEVINFO** – Print the names of all disks and their drive designators.

**Example:** DEVINFO

ERTFS Device List

==========

Device name: HOST DISK HOSTDISK.DAT Is mounted on G:

Device name: RAM DISK Is mounted on I:

Device name: STATIC ROM DISK Is mounted on J:

Device name: FLASH DISK Is mounted on H:

---

**MKDIR** - *Create a directory.* This command creates a directory.

**Example:** MKDIR \USR\NEWDIR

---

**RENAME** - *Rename a file.* This command will rename a file.

**Example:** RENAME C:\TES\JOSUF.TXT JOSEPH.TXT

---

**RMDIR -** *Remove a subdirectory.*

**Example:** RMDIR C:\TEMPDIR**CAT** - Display contents of a file

This command displays the contents of a file to the console.

Example:

CAT A:\use\ASCII\budget.txt

---

**DELETE** - *Delete a file*

This command deletes a file.

**Example:** DELETE A:\use\ASCII\budget.txt

---

**CD -** *Set or display working directory. T*his command sets the default directory if an argument is supplied, otherwise it displays the current working directory.

**Example:** CD - Display working directory

**Example:** CD \usr\data - Change working directory

---

**COPY** - *Copy a file to another location.* This command copies the source file to the destination.

**Example:** COPY A:FILE.DAT B:FILE.DAT

---

**DELTREE** - *Delete a directory and its descendants*

**Example:** DELTREE C:\TEMP

---

**DIFF** - *Compare two files.* This command compares two files and prints whether or not they are the same.

**Example:** DIFF A:FILE1.DAT B:FILE1.DAT

---

**DSKSEL -** *Set default drive*

This command sets the default drive so that subsequent commands that do not explicitly contain a drive specifier will refer to this drive.

**Example:** DSKSEL D:

---

**FILLFILE** - *Create a file and fill it with a pattern.* This command creates a file and repeatedly fills it with a pattern. It is useful when you wish to create some files for experimenting with on an otherwise empty volume.

**Example: c**reate and fill the file file.dat with the pattern “THIS IS A TEST” 1000 times.

FILLFILE FILE.DAT “THIS IS A TEST” 1000

---

**GETATTR** - *Print a file’s attributes.* This command calls the pc_get_attributes library routine and prints the results.

**Example:** GETATTR FILE.DAT

---

**SETATTR** - *Change a file’s attributes.* This command calls the pc_set_attributes library routine to change a file’s attributes

**Example:** SETATTR FILE:DAT RDONLY*

The following values may be type for the attribute, RDONLY,HIDDEN,SYSTEM,ANORMAL

---

**STAT** - *Print file properties.* This command calls the stat library routine and prints the results.

**Example:** STAT A:FILE.DAT

---

**CHSIZE** - *Truncate or extend a file*

**Example**: CHSIZE A:DATAFILE 4096 - Change DATAFILE's size to 4096 bytes

---

**RNDOP -** *Open a random access file.* This routine will open or reopen a file for use by our random access file I/O test commands READ, WRITE and SEEK. It must be given the file name and the record size for the file. The record size is stored internally and is used to pad write operations to the correct width. (Record size should not exceed 512). The Use CLOSE to close a file that was opened with RNDOP and LSTOPEN to display all open files. RNDOP does not return the file handle so use LSTOPEN. *Note: The file handles are always return 0, 1, 2, 3 .... Use this knowledge if you want to use random access files in a script.*

**Example:** RNDOP TESTFIL 200

---

**CLOSE** - *Close a random access file.* This command closes a random access file that was opened with **RNDOP**. See **RNDOP** for a discussion of random access files.

**Example:** CLOSE 1

---

**LSTOPEN** - *Display all open random access files.* This command lists all open random access files along with their file handles. This is especially handy since after the initial OPEN all accesses are done VIA the handle, and it is easy to forget which handle goes with which file.

**Example:** LSTOPEN

---

**READ** - *Read and display a random access record* . This command reads data from the random access file and prints its value to the console. (See **WRITE** for how to write data to the file, **SEEK** for how to seek to a record in the file, LSTOPEN to list all random access files by handle and, RNDOP for how to open a random access file.).

**Example:**

RNDOP \TEST\FILE 100- open(returns handle=0)

SEEK 0 0

WRITE 0 "This is record zero"

SEEK 0 1

WRITE 0 This is record one"

SEEK 0 0

READ 0 - This will print "This is record zero"

CLOSE 0

---

**WRITE -** *Write data to a random access file.* This command writes data to the current record of a random access file. The data is filled to the correct width (with spaces) internally. Multi word strings should be quoted.

**Example:** See the READ command for an example

---

**SEEK** - *Seek to a record in a random access file*

This command seeks to a record number in a random access file. It takes a file handle and a record number as an argument.

Example: See example for the READ command

---

**HELP** - *Display all commands*

**Example:** HELP

---

**QUIT** - *Exit the command shell.* This command exits the command shell and returns to the caller.

## APPENDIX J: ERTFS SYSTEM ERRORS

If an error occurs during an ERTFS API call, errno will be set to the values specific to that call or it may be set to any one of the following system detected errors:

**Device check system errors:**

| | | |
|---|---|---|
| PEINVALIDDRIVEID | - | Invalid driver ID requested |
| PEDEVICECHANGED | - | Device is/was flushed but has been swapped |
| PEDEVICEFAILURE | - | Device driver reports device not functional |
| PEDEVICEINIT | - | Device not initialized in pc_ertfs_init |
| PEDEVICENOMEDIA | - | No media is installed |
| PEDEVICEUNKNOWNMEDIA | - | Device driver reports unknown media type |
| PEIOERRORREADMBR | - | Error reading the master boot record |
| PEINVALIDMBR | - | No partition signature found in MBR |
| PEINVALIDMBROFFSET | - | Drives partition offset not found in MBR |
| PEIOERRORREADBPB | - | Error reading the bios parameter block |
| PEINVALIDBPB | - | Signature not found in the bios parameter block |
| PEIOERRORREADINFO32 | - | Error reading the FAT32 info block |

**Internal system Errors:**

| | | |
|---|---|---|
| PEINTERNAL | - | Inconsistent state for unexplained reasons |
| PEINVALIDCLUSTER | - | Invalid cluster number encountered |
| PEINVALIDCLUSTER | - | Invalid cluster number encountered |
| PEINVALIDBLOCKNUMBER | - | Invalid block number encountered |
| PEINVALIDDIR | - | Invalid directory entry encountered |

**Resource allocation system Errors:**

| | | |
|---|---|---|
| PERESOURCE | - | Out of internal directory allocation structures |
| PERESOURCEBLOCK | - | Out of block buffers |
| PERESOURCEFATBLOCK | - | Out of FAT block buffers |

**Device I/O read system errors:**

| | | |
|---|---|---|
| PEIOERRORREADFAT | - | I/O error when reading from the FAT area |
| PEIOERRORREADBLOCK | - | I/O error reading from the directory area |

**Device I/O write system errors:**

| | | |
|---|---|---|
| PEIOERRORWRITEBLOCK | - | I/O error writing to the directory area |
| PEIOERRORWRITEFAT | - | I/O error flushing FAT ram buffers to disk |
| PEIOERRORWRITEINFO32 | - | I/O error writing INFO32 structure during FAT flush |

**FailSafe Operating Mode errors:**

| | | |
|---|---|---|
| PEFSCREATE | - | Disk mount failed, could not create a journal file |
| PEJOURNALFULL | - | The API operation failed because there is no room left in the journal file |
| PEIOERRORWRITEJOURNAL | - | Disk mount or API operation failed because a write to the journal file failed |
| PEFSRESTOREERROR | - | Disk mount failed, could not restore from journal file |
| PEFSRESTORENEEDED | - | Disk mount failed, a restore is needed but the AUTORESTORE feature is not enabled |
| PEIOERRORREADJOURNAL | - | Disk mount or API operation failed because a read from the journal file failed |