Rtfs

Device Driver and Porting Guide

Revised June 2008

For best online viewing experience we recommend using Adobe Acrobat's **Bookmarks** tab for navigating

EBS Inc. 39 Court Street Groton MA 01450 USA

http://www.ebsembeddedsoftware.com

Table of Contents

Introduction

This section takes a detailed look at the steps involved in porting Rtfs to a new target.

Porting Rtfs to a new target involves the following steps.

Configure target specific CPU configurations – Edit the file rtfscommon/include/rtfsarch.h to configure Rtfs for your target CPU.

Implement target specific tick, mutex and task management adaptation – Provide a file named portkern.c for your target operating system and implement the required functionality. The subdirectory named rtfstargets contains several instances of portkern.c that may already be configured for your OS or that you may modify.

When using certain EBS stock drivers you must implement target specific signaling, interrupt, DMA, and register access functions – Modify some functions in portkern.c and add in an additional file named portio.c for your target hardware. Portio.c is not required unless you are using certain Rtfs stock drivers.

Optional target specific console input and output functions – You may implement within portkern.c terminal line input and output functions. If your target can support terminal input and output, you will be able to view diagnostic output and run the included interactive shell programs for both testing and managing the contents of your media. (*Note: If a network port is available, you may use telnet instead for your console input/output functions.*)

Optional target specific memory allocation functions - The core Rtfs library does not require dynamic memory allocation but it may be configured to take advantage of it if the constant named **RTFS_CFG_ALLOC_FROM_HEAP** is enabled in rtfsconf.h.

PORTKERN.C - OS Specific support functions

Some porting requirements depend on which device drivers are used – If not all device drivers are being used then all target specific resources need not be implemented. These needs are pointed out in this document, so read it carefully before proceeding to avoid unnecessary work.

DYNAMIC MEMORY SUPPORT

- Dynamic memory functions must be implemented to use Rtfs command shells and test code.
- Dynamic memory functions must be implemented if you want Rtfs to use dynamic memory allocation.

See the Initialization and shutdown and Media driver interface sections of the API Reference Guide for more information about dynamic versus static configuration.

*void *__rtfs_port_malloc__(int nbytes)*

This routine can be implemented as return(malloc(nbytes));

*void __rtfs_port_free__(void *ptofree)*

This routine can be implemented as free(ptofree);

PERIODIC CLOCK SUPPORT

Rtfs requires a periodic clock for measuring device watchdog timeouts. Also if Rtfs is running in **POLLED** mode, it needs a clock source for timed signal tests.

The periodic clock support functions must be implemented.

dword __rtfs_port_get_ticks__(void)

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.

*void **rtfs_port_sleep**(int sleeptime)*

This routine takes as an argument the time to sleep 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 cpu. In non-RTOS environments we provide a sample implementation of portkern.c with this function implemented using **rtfs_port_get_ticks()**.

*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. Rtfs makes no assumptions about the value that is returned. *Note: we provide sample implementations of this function that should work unmodified in either an RTOS or non-RTOS environment 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: we provide sample implementations of this function that should work unmodified in either an RTOS or non-RTOS environment as long the routine **rtfs_port_get_ticks()** has been implemented.*

CLOCK CALENDAR SUPPORT

Clock calendar must be provided by the application level callback function **rtfs_sys_callback()**. See the Application callbacks section of the API Reference Guide for more information.

MUTEX SEMAPHORE SUPPORT

The mutex semaphore support functions must be implemented if Rtfs is to be used in a multi-threaded environment.

Note: Rtfs 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 Rtfs, then please skip this section.

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

- **rtfs_port_alloc_mutex()**
- **rtfs_port_free_mutex()**
- **rtfs_port_claim_mutex()**
- **rtfs_port_release_mutex()**
- **rtfs_port_release_mutex()**

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. Rtfs 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 Rtfs calling function will return failure. *NOTE: If dynamic creation and deletion of semaphores is not supported a simulation must be devised.*

void **rtfs_port_free_mutex**(dword handle)

This routine takes as an argument a mutex handle that was returned by **rtfs_port_alloc_mutex()**. It must free the mutex. *NOTE: If dynamic creation and deletion of semaphores is not supported a simulation must be devised.*

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.

IDENTIFYING THE CURRENT TASK ID

The identify current task ID function must be implemented if Rtfs is to be used in a multi-threaded environment and you wish for each thread to have it's own persistent notion of current working directory, current working drive ID and last reported errno value. (ie: If **RTFS_CFG_NUMUSERS** is greater than 1). If this function is not implemented Rtfs will still function in a multitasking environment but these state variables will be shared by all threads, Making errno unreliable and producing unexpected behavior if one of the threads changes the working drive or directory.

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: This function requires no modification to run in polled mode.

THREAD LOCAL STORAGE FOR USER CONTEXT STRUCTURES

Thread local storage is a more efficient way to map threads to user structures. These functions may be implemented if your RTOS supports thread local storage.

NOTE: If your RTOS does not support thread local storage set **INCLUDE_THREAD_SETENV_SUPPORT** *to zero in rtfscommon\include\rtfsarch.h and omit these functions.*

void **rtfs_port_set_task_env**(void *pusercontext)

This routine takes as an argument the address of the Rtfs user context structure reserved for this thread. It must store the address in thread local storage so that the same value will be returned by **rtfs_port_get_task_env()** when called from this same thread.

void ***rtfs_port_get_task_env**(void)

This routine retrieves the last value that was passed to **rtfs_port_set_task_env()** by the currently executed thread.

Note: If **rtfs_port_set_task_env()** *was not previously called for this thread, this function must return 0.*

AUTOMATIC RELEASE OF CONTEXT STRUCTURES WHEN TASKS EXIT

When a thread accesses Rtfs it permanently reserves a user structure. User structures must be released when threads exit or the user structure pool will be exhausted. This function is called to register a exit handler if the RTOS supports thread exit handlers.

NOTE: If your RTOS does not support thread exit handler set **INCLUDE_THREAD_EXIT_CALLBACK** *to zero in rtfscommon\include\rtfsarch.h and omit this functions.*

void **rtfs_port_set_task_exit_handler**(void)

Rtfs calls this function when a thread first calls the Rtfs API. If task exit functions are supported this function should instruct the operating system to call the Rtfs API function **pc_free_user()** when the task exits. If the operating system can not do this then the application must make certain that **pc_free_user()** is called from the thread before it exits.

CONSOLE INPUT AND OUTPUT SUPPORT

Console input and output support must be provided by the application level callback function **rtfs_sys_callback()**. See the Application callbacks section of the API Reference Guide for more information.

SHUTDOWN SUPPORT

Shutdown support is only required if your application calls **pc_ertfs_shutdown()**.

void **rtfs_port_shutdown**(void)

Resources allocated by Rtfs are released before it exits. This callback is provided so the porting layer and device drivers can de-allocate resources they have allocated. **rtfs_port_shutdown()** may also disable devices and device interrupts

EVENT SIGNALING SUPPORT

(Event signaling is only required to support certain stock device drivers. Skip this section if you are not using these drivers).

*NOTE: Rtfs does not automatically free signals that device drivers request. If you plan to support shutdown, then an internal method must be devised to release signals when **rtfs_port_shutdown()** is called.*

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 within a task context. They are always signaled from the interrupt context.

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. Rtfs will not interpret the value of the return value. The handle will only be 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 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 **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.*

FLOPPY DISK INTERRUPT SUPPORT

(required for stock floppy disk driver only. If not skip this section.)

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.

IDE INTERRUPT SUPPORT

(required for stock IDE only. If not skip this section.)

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

82365 PCMCIA CONTROLLER INTERRUPT SUPPORT

(required for stock PCMCIA driver only. If not 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" defined in pcmctrl.c.

Creating Your Own Device Drivers

The Driver mount function

Device drivers are bound to Rtfs by calling the function **pc_rtfs_media_insert()**. This function simultaneously provides Rtfs with all of the media information and all of the device driver access points Rtfs needs for the media. Rtfs and the device driver are unbound when **pc_rtfs_media_alert(RTFS_ALERT_EJECT,..)** is called.

Please consult the documentation in the API reference guide for the function **pc_rtfs_media_insert()** and the functions named **device_io(), device_erase(), device_ioctl(), device_configure_media()** and **device_configure_volume()**, in the *Media Driver Callback section.*

For Rtfs supplied sample device drivers we provide a function named BLK_DEV_xxxxxx_Mount(void), where xxxxx is the name of the device driver, that fills in a rtfs_media_insert_args structure and calls **pc_rtfs_media_insert()** to attach the drive. We then call BLK_DEV_xxxxxx_Mount() from the startup code.

See the function named BLK_DEV_hostdisk_Mount(void) in \rtfsdrivers\hostdisk\drhostdisk.c for and example.

The Device block data transfer function

*Please refer to the documentation for the function named **device_io()** in the Media Driver Callback section of the API Reference Guide.*

The Device I/O control function

*Please refer to the documentation for the function named **device_ioctl()** in the Media Driver Callback section of the API Reference Guide.*

Please review the following table if you are converting older Rtfs device drivers to support **pc_rtfs_media_insert()**.

Strategies for converting OP-CODE handlers from version four and earlier Rtfs device drivers to version 6.	
OP-CODE	Description
DEVCTL_WARMSTART	Obsolete. Device driver must be initialized before **pc_rtfs_media_insert()** is called.
DEVCTL_CHECKSTATUS	Obsolete. System software must call **pc_rtfs_media_alert(RTFS_ALERT_EJECT,..)** when the device is removed. Rtfs calls **rtfs_sys_callback(RTFS_CBS_POLL_DEVICE_READY)** before every device access. Systems may use this callback to poll for device removal events and call **pc_rtfs_media_alert(RTFS_ALERT_EJECT,..)** if any are detected.
DEVCTL_GET_GEOMETRY	Obsolete, device geometry is passed to **pc_rtfs_media_insert()**.
DEVCTL_FORMAT	Obsolete, see the manual page for **device_ioctl()** regarding **RTFS_IOCTL_FORMAT**.
DEVCTL_REPORT_REMOVE	Obsolete use **pc_rtfs_media_alert(RTFS_ALERT_EJECT,..)** instead.
DEVCTL_POWER_LOSS	Obsolete
DEVCTL_POWER_RESTORE	Obsolete

PORTIO.C - Hardware abstraction layer

(required Only if using certain stock device drivers. If not skip this section.)

Portio.c contains functions that must be customized to be used with your target hardware. These functions are only required for certain device drivers, the table below names the functions and the device drivers they are used for. Functions that require further explanation are described in the device specific sections of this document. Please ignore any functions that are not required for device drivers you are using. If you must implement any of these functions, a fully populated version of this file is provided in the subdirectory named rtfstargets/x86.io. You may want to copy this file and edit it to fit your needs.

Hardware interface functions implemented in portio.c that may required customization	
Function Name	Required only for these device drivers
rtfs_port_disable	FLOPPY/82365/FLASH
rtfs_port_enable	FLOPPY/82365/FLASH
phys82365_to_virtual	82365 PCMCIA
write_82365_index_register	82365 PCMCIA
write_82365_data_register	82365 PCMCIA
read_82365_data_register	82365 PCMCIA
ide_rd_status	IDE/ATA
ide_rd_data	IDE/ATA
ide_rd_sector_count	IDE/ATA
ide_rd_alt_status	IDE/ATA
ide_rd_error	IDE/ATA
ide_rd_sector_number	IDE/ATA
ide_rd_cyl_low	IDE/ATA
ide_rd_cyl_high	IDE/ATA
ide_rd_drive_head	IDE/ATA
ide_rd_drive_address	IDE/ATA
ide_wr_dig_out	IDE/ATA
ide_wr_data	IDE/ATA
ide_wr_sector_count	IDE/ATA
ide_wr_sector_number	IDE/ATA
ide_wr_cyl_low	IDE/ATA
ide_wr_cyl_high	IDE/ATA
ide_wr_drive_head	IDE/ATA
ide_wr_command	IDE/ATA

ide_wr_feature	IDE/ATA
ide_insw	IDE/ATA
ide_outsw	IDE/ATA
rtfs_port_ide_bus_master_address	IDE/ATA
ide_detect_80_cable	IDE/ATA
Hardware interface functions implemented in portio.c that may required customization	
Function Name	Required only for these device drivers
ide_rd_udma_status	IDE/ATA
ide_wr_udma_status	IDE/ATA
ide_rd_udma_command	IDE/ATA
ide_wr_udma_command	IDE/ATA
ide_wr_udma_address	IDE/ATA
rtfs_port_bus_address	IDE/ATA
read_mmc_word	Multi Media Card
write_mmc_word	Multi Media Card
read_smartmedia_byte	SMARTMEDIA
write_smartmedia_byte	SMARTMEDIA

INTERRUPT ENABLE AND DISABLE SUPPORT

The interrupt enable and disable functions are required only if the floppy disk driver, the 82365 PCMCIA controller, or the flash chip memory technology driver are 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

DRIDEATA.C - ATA/IDE and Compact Flash Support

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, you must populate several register access functions in portio.c and provide an interrupt service layer in portkern.c if you wish to run the device in interrupt driven mode versus polled mode.

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

See the porting instructions for portkern.c.

REGISTER ACCESS FUNCTIONS REQUIRED BY THE DRIVER

All ide register access functions are implemented in the file named portio.c. The register access functions access registers in a register file whose base is contained in the register_file_address field of the driver function. This field may be initialized in **pc_ertfs_init()** when the IDE device driver is registered. Since most embedded applications have only a single ATA controller at a fixed address, the register_file_address field may typically be ignored and the access functions may be hardwired to access the proper memory addresses.

byte ide_rd_status(dword register_file_address)	Read the byte at location 7 (**IDE_OFF_STATUS**) of the ide register file.
byte ide_rd_status(dword register_file_address)	Read the byte at location 0 (**IDE_OFF_DATA**) of the ide register file.
byte ide_rd_sector_count(dword register_file_address)	Read the byte at location 2 (**IDE_OFF_SECTOR_COUNT**) of the ide register file.
byte ide_rd_alt_status(dword register_file_address, int contiguous_io_mode)	Read the byte at location 0x206 (**IDE_OFF_ALT_STATUS**) of the ide register file. If the contiguous_io_mode argument is 1, then read the byte at location 14 rather than 0x206.

byte ide_rd_error(dword register_file_address)	Read the byte at location 1 (**IDE_OFF_ERROR**)of the ide register file.
byte ide_rd_sector_number(dword register_file_address)	Read the byte at location 3 (**IDE_OFF_SECTOR_NUMBER**) of the ide register file.
byte ide_rd_cyl_low(dword register_file_address)	Read the byte at location 4 (**IDE_OFF_CYL_LOW**) of the ide register file.
byte ide_rd_cyl_high(dword register_file_address)	Read the byte at location 5 (**IDE_OFF_CYL_HIGH**) of the ide register file.
byte ide_rd_drive_head(dword register_file_address)	Read the byte at location 6 (**IDE_OFF_DRIVE_HEAD**) of the ide register file.
byte ide_rd_drive_address(dword register_file_address, int contiguous_io_mode)	Read the byte at 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 read the byte at location 15 rather than 0x207.
void ide_wr_dig_out(dword register_file_address, int contiguous_io_mode, byte value)	Write the byte to location 0x206 (**IDE_OFF_ALT_STATUS**) of the ide register file. If the contiguous_io_mode argument is 1, write the byte to offset 14 rather than 0x206.
void ide_wr_data(dword register_file_address, word value)	Write the word to location 0 (**IDE_OFF_DATA**) of the ide
void ide_wr_sector_count(dword register_file_address, byte value)	Write the byte to location 2 (**IDE_OFF_SECTOR_COUNT**) of the ide register file.
void ide_wr_sector_number(dword register_file_address, byte value)	Write the byte to location 3 (**IDE_OFF_SECTOR_NUMBER**) of the ide register file.
void ide_wr_cyl_low(dword register_file_address, byte value)	Write the byte to location 4 (**IDE_OFF_CYL_LOW**) of the ide register file.
void ide_wr_cyl_high(	Write the byte to location 5

dword register_file_address, byte value)	(**IDE_OFF_CYL_HIGH**) of the ide register file.
void ide_wr_drive_head(dword register_file_address, byte value)	Write the byte to location 6 (**IDE_OFF_DRIVE_HEAD**) of the ide register file.
void ide_wr_command(dword register_file_address, byte value)	Write the byte to 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(dword 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(dword 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.

IDE ULTRA DMA MODE SUPPORT

This section describes functions that must be customized in order to use the ATA/ATAPI device driver in ultra dma mode. These support functions reside in portio.c and must be modified for your target if you wish to use ULTRA DMA

If you don't wish to use ULTRA DMA, you may set the compile time constant INCLUDE_UDMA to 0 in portconf.h and ignore this section.

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 it must 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 a byte 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 a byte 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 a dword to location 4 of the bus master control region.
unsigned long rtfs_port_bus_address (void * p)	This function must map a pointer to an unsigned long physical address.

<table>
<tr>
<td>BOOLEAN ide_detect_80_cable
(
int controller_number
)</td>
<td>This function must determine if the ATA cable is 80 wires or 40 wires. If an 80 wire cable is installed it should return TRUE, otherwise FALSE. In an embedded system this function can be hardwired based on the hardware configuration. If an 80 wire cable is installed the ide device driver uses the highest performance mode supported by the attached drive.
<table>
<tr><td>mode 6</td><td>133 MB</td></tr>
<tr><td>mode 5</td><td>100 MB/s</td></tr>
<tr><td>mode 4</td><td>66 MB/s</td></tr>
<tr><td>mode 3</td><td>44 MB/s</td></tr>
</table>
</td>
</tr>
</table>

SUPPORTING REMOVABLE ATA DEVICES

This section describes what is required to support hot swapping of removable TRUE IDE and PCIMCIA compact flash 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 media change event. Six routines related hot-swapping are provided: **trueide_card_changed()**, **trueide_card_installed()** and **trueide_report_card_removed()**, **pcmctrl_card_changed()**, **pcmctrl_card_installed()** and **pcmctrl_card_down()**. The requirements for each of these routines are 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 **pcmctrl_card_changed**(int pcmcia_slotno)

This routine is analogous to the trueide_card_changed function. The pcmcia subsystem must report if a removal or insertion event has occurred.

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);
}
```

BOOLEAN **pcmctrl_card_installed**(int pcmcia_slotno)

This routine is analogous to the **trueide_card_installed()** function. The pcmcia subsystem must report if a card is installed.

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. The 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);
}
```

void **pcmctrl_card_down**(int pcmcia_slotno)

This routine is analogous to the **trueide_card_installed()** function. The pcmcia subsystem must clear any state variables and shut off power to the drive.
Note: Failure to shut off power to the slot will cause device accesses on subsequent reinsertions to fail.

DRFLSFTL.C - Linear Flash Support

Rtfs linear flash device support is provided through a portable Flash Translation layer (FTL) implemented in drflsftl.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 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 (MTD) 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 in 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, the total size, the erase block size, 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. **flash_probe()** must return 1 if a device is found, zero otherwise.

These values must be set by the **flash_probe()** routine:

flashchip_TotalSize - Set this to the total size of the flash in bytes.
flashchip_BlockSize - 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.

void * **mtd_MapWindow** (RTFDrvFlashData *DriveData, dword BlockIndex, dword WindowIndex)

Map a 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 region must be at least **flashchip_WindowSize** bytes wide.
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.
mtd_MapWindow() must return a pointer that can be used to read and write the flash at the requested location

int **flash_erase_block**(dword BlockIndex)

Erase a block of size **flashchip_BlockSize** at BlockIndex to the erased (all 1's) state. **flash_erase_block()** must return 0 on success and -1 on failure.

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

Write 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 and 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. 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.

The 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 modified. **FLASH_STARTING_ADDRESS** must be changed to the base address of your flash memory and set **FLASHWINDOWSIZE** to one of the following: If the whole address range of your flash part is completely visible then set **FLASHWINDOWSIZE** to the size of the flash (in bytes). If it is addressable only through a memory window that is smaller than 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.

DRPCMCIA.C - Pcmcia Support

Rtfs 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 Rtfs PCMCIA subsystem must be used. The subsystem consists of the files drpcmcia.c, which is portable and should not need modifications, and drpcmctl.c, a device driver for INTEL 82365 compatible pcmcia controller chips which will need some porting.

SUPPORTING A NON 82365 BASED PCMCIA CONTROLLER

If you are not using an 82365 class controller you must provide the functionality normally provided in drpcmctl.c with equivalent functions. These functions are described here. 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:
*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.*
*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 always return FALSE. 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 off 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.*

IMPLEMENTING CARD EVENT HANDLERS FOR EXTERNALLY SUPPLIED PCMCIA CONTROLLER DRIVERS

If you wish to support hot swapping of PCMCIA 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. That 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);
               }
            }
         }
      }
 }
```

PORTING THE 82365 PCMCIA CONTROLLER

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)

See the porting instructions for portkern.c.

void 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."

void **write_82365_index_register**(byte value)
void **write_82365_data_register**(byte value)
byte **read_82365_data_register**()

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.

DRFLOPPY.C - Floppy Disk Support

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)

See the porting instructions for portkern.c.

void **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