Rtfs

Application Notes

Revised June 2008

EBS Inc. 39 Court Street Groton MA 01450 USA
http://www.ebsembeddedsoftware.com

TABLE OF CONTENTS

Notes on direct DMA to disk applications

Introduction

The RtfsProPlus API may be used to create applications that can use DMA to burst application data directly to and from the disk. We take advantage of the following features of RtfsProPlus to achieve this:

- Cluster pre-allocation
- Contiguous cluster allocation
- Reading, writing and appending to data files without performing any data transfers
- Real time, guaranteed zero disk access, file extent tracking and allocation.
- Specialized usage of API routines.

How it's done:

The routine **pc_efilio_fpos_sector** returns the raw sector number at the current file pointer and the number of contiguous sectors in the file, starting at, and including that sector. **pc_efilio_fpos_sector** may be used in conjunction with features of **pc_cfilio_write**, **pc_efilio_read**, **pc_cfilio_read** and **pc_efilio_read** to achieve full DMA data transfer to and from data files from the application layer.

The following pages explain by example the steps that are needed to build DMA enabled applications. Three examples are provided:

Example 1: Filling a file using DMA operations controlled at the application layer.
Example 2: Reading a file using DMA operations controlled at the application layer.
Example 3: Writing and Reading a circular file using DMA operations controlled at the application layer.

These examples are provided along with commentary in this manual. The source code for the examples is contained within the file **efishell.c**.

Example 1: Filling a file using DMA operations controlled at the application layer.

This code fragment demonstrates the basic methods needed to use RtfsProPlus to create files that you will populate using direct memory access.

For this demonstration we are using ideal conditions that demonstrate the principals involved without going into excess detail.

For these ideal conditions we will use a DMA transfer size that is an even multiple of the cluster size, and set the minimum pre-allocation size to the same as the transfer size. This way there is a one to one to one to one correspondence of pre-allocated requests, get block number requests, DMA data to the disk requests, expand the file and advance the file pointer requests.

If these ideal conditions were not met we would be forced to repeat one or more of the steps within an internal loop, but the principals are the same.

The steps are:
First open the file with the minimum allocation field set to two clusters
Then in a repeating loop (512 times)

1. Call **pc_efilio_write** with a null buffer and zero write count. This calling sequence instructs **pc_efilio_write** to pre-allocate clusters if it is currently at the end of file and more clusters are required. It does not advance the file pointer or change the file size.
2. Call **pc_efilio_fpos_sector** to retrieve the raw sector number at the current file pointer and the number of contiguous sectors.
3. Simulate a DMA transfer to the disk with **pc_raw_write**, which writes contiguous sectors to the disk at the raw sector returned from **pc_efilio_fpos_secto**r.
4. Call **pc_efilio_write** with a null buffer and a write count equal to 512 times the number of sectors we just transfered. This instructs **pc_efilio_write** to make the previously pre-allocated clusters a permanent part of the file and to adjust the file size and advance the file pointer.

A few things to keep in mind are:

1. No disk IO takes place during steps one, two and four. So these steps are essentially real time processes.
2. Because this is only a simulation we are waiting for step three to complete before we perform step four to update the file structure and then step one and two again to prepare for another "DMA" transfer. This "synchronous" operation is not necessary. The algorithm could be implemented in several other ways.

For example:

1. Step three could simply start a DMA transfer and return from the loop. Steps four, one and two could then be executed asynchronously when the DMA transfer completed.
2. Step three could start the DMA transfer and steps four, one and two could then be executed immediately. This would have the affect of creating a DMA write behind a scheme where the file size is up to date, the file pointer is up to date, the DMA circuit is armed to perform the transfer, and where the next sector number and sector count are ready so the DMA circuit can be re-armed immediately after completion of the current iteration. This could be repeated as many times as desired to provide a simple method of queuing multiple unattended DMA transfers to populate the file.

```
void demonstrate_dma_to_efile(byte *filename)
{
   int fd, i, driveno;
   dword cluster_size_bytes,dma_data_size_blocks,ltemp;
   byte *pdma_buffer, drive_name[8];
   EFILEOPTIONS my_options;

   // We'll simulate ideal conditions for this demonstration by
   // setting dma transfer size and minumum allocation size
   // the same, at two clusters
   driveno = pc_get_default_drive(drive_name);
   cluster_size_bytes = (dword) pc_cluster_size(drive_name);
   if (!cluster_size_bytes)
      return;  // ????

   rtfs_memset(&my_options, 0, sizEOF(my_options));
   // Set the minumum allocation size to two clusters
   my_options.min_clusters_per_allocation = 2;
   // And the dma transfer size to two clusters
   dma_data_size_blocks = (2 * cluster_size_bytes)/512;
   // Open the file
   fd = pc_efilio_open(filename,
(word)(PO_BINARY|PO_RDWR|PO_TRUNC|PO_CREAT),
                           (word)(PS_IWRITE | PS_IREAD),&my_options);
   if (fd<0)
      return;

   // Perform 512 simulated cluster sized dma transfers while
   // extending the file
   pdma_buffer = (byte *) pro_test_malloc(dma_data_size_blocks*512);
   for(i=0;i<512;i++)
   {
   dword raw_sector_number,raw_sector_count;
      // force pc_efilio_write to preallocate clusters if neccessary
      // without changing the file size or advancing the file pointer
      if (!pc_efilio_write(fd, 0,0,&ltemp))
         break;
      // Get the raw disk sector associated with the file pointer
      if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                       &raw_sector_count))
         break;
     // This should never happen because we are cluster aligned
      if (raw_sector_count < dma_data_size_blocks)
         break;
      // raw_sector_number contains the sector at the file pointer
      // raw_sector_count is the count of contiguous sectors there
      // truncate the raw sector count if it is greater than our
      // transfer size. This won't happen in this demonstration
      // because we set min_clusters_per_allocation to 1, but
      // if we had used a larger minimum value raw_sector_count would
      // reflect that we are preallocating larger chunks
      if (raw_sector_count > dma_data_size_blocks)
```

```
            raw_sector_count = dma_data_size_blocks;

        // Now use the RtfsProPlus raw block write routine to
        // Simulate a dma transfer
        if (!pc_raw_write(driveno,  pdma_buffer, raw_sector_number,
                                    raw_sector_count, TRUE))
            break;
        // Now call write with a null data buffer to advance the file
        // pointer and update the file size
        if (!pc_efilio_write(fd, 0,raw_sector_count*512,&ltemp))
            break;
    }
    pc_efilio_close(fd);
    pro_test_free(pdma_buffer);
}
```

Example 2: Reading a file using DMA operations controlled at the application layer.

This code fragment demonstrates the basic methods needed to use RtfsProPlus to read files using direct memory access.

For this demonstration we will use a DMA transfer size that is larger than the transfer size used in example one, and it is not an even multiple of the cluster size. This size is chosen to demonstrate that the transfer size does not have to be cluster aligned and that RtfsProPlus will return to us the number of contiguous sectors at the file pointer.

The steps are:
First re-open the file (presumably the file created in example one) Then in a repeating loop (repeating as many times as required)

1. Call **pc_efilio_fpos_sector** to retrieve the raw sector number at the current file pointer and the number of contiguous sectors.
2. Simulate a DMA transfer from the disk with **pc_raw_read**, which reads contiguous sectors from the disk at the raw sector returned from **pc_efilio_fpos_sector**.
3. Call **pc_efilio_read** with a null buffer and a read count equal to 512 times the number of sectors we just transferred. This instructs **pc_efilio_read** to advance the file pointer.

A few things to keep in mind are:

1. No disk IO takes place during steps 1, and 3. So these steps are essentially real time processes.
2. Because this is only a simulation we are waiting for step 2 to complete before we perform step three and step one again to prepare for another DMA transfer.

This synchronous operation is not necessary. The algorithm could be implemented in several other ways.

For example:

1. Step two could simply start a DMA transfer and return from the loop. Steps three and one could then be executed asynchronously when the DMA transfer completed.
2. Step two could start the DMA transfer and steps three and one could then be executed immediately. This would have the affect of creating a DMA read behind a scheme where the DMA circuit is armed to perform the transfer, and where the next sector number and sector count are ready so the DMA circuit can be re-armed immediately after completion of the current iteration. This can be repeated as many times as desired and provides a simple method of queuing up multiple unattended DMA transfers to read the file. The file could even be closed before the transfers complete.

```
void demonstrate_dma_from_efile(byte *filename)
{
   int fd, driveno;
   dword cluster_size_bytes,dma_data_size_blocks,ltemp;
   byte *pdma_buffer, drive_name[8];
   EFILEOPTIONS my_options;


   // Re-open the file
   rtfs_memset(&my_options, 0, sizEOF(my_options));
   fd = pc_efilio_open(filename, (word)(PO_BINARY|PO_RDWR),
                          (word)(PS_IWRITE | PS_IREAD),&my_options);
   if (fd<0)
      return;

   // We'll set a dma transfer size of 8 clusters minus
   // one block to demonstrate that the transfers have to
   // be block aligned, but not cluster aligned

   // In this example we'll use pc_fd_to_driveid to get the drive number
   // and drive name from the open file.
   driveno =  pc_fd_to_driveid(fd,drive_name);
   cluster_size_bytes = (dword) pc_cluster_size(drive_name);
   if (!cluster_size_bytes)
      return;  // ????
   dma_data_size_blocks = ((8 * cluster_size_bytes) - 1)/512;


   // Perform as many dma transfers as needed to read the whole file
   pdma_buffer = (byte *) pro_test_malloc(dma_data_size_blocks*512);
   for(;;)
   {
   dword raw_sector_number,raw_sector_count;
      // Get the raw disk sector associated with the file pointer
      // The second argument is TRUE (check for read)
      if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                          &raw_sector_count))
         break;
     // This will happen when the file pointer is at the end
      if (!raw_sector_count)
```

```
            break;
        // raw_sector_number contains the sector at the file pointer,
        // raw_sector_count is the count of contiguous sectors there,
        // truncate the raw sector count if it is greater than our
        // transfer size. This will happen because the file is more
        // than likely contiguous and raw_sector_count returns the
        // number of contiguous sectors
        if (raw_sector_count > dma_data_size_blocks)
            raw_sector_count = dma_data_size_blocks;

        // Now use the RtfsProPlus raw block write routine to
        // Simulate a dma transfer from the drive
        if (!pc_raw_read(driveno,  pdma_buffer, raw_sector_number,
                                    raw_sector_count, TRUE))
            break;
        // Now call read with a null data buffer to advance the file
        // pointer
        if (!pc_efilio_read(fd, 0,raw_sector_count*512,&ltemp))
            break;
    }
    pc_efilio_close(fd);
    pro_test_free(pdma_buffer);
}
```

Example 3: Writing and Reading a circular file using DMA operations controlled at the application layer.

This code fragment demonstrates the basic methods needed to use RtfsProPlus to read and write circular files using direct memory access.

For this demonstration we will use a DMA transfer size that is larger than the transfer size used in example one, and that is not an even multiple of the cluster size. This size is chosen to demonstrate that the transfer size does not have to be cluster aligned and that RtfsProPlus will return to us the number of contiguous sectors at the file pointer.

The steps are:
First re-open the file (presumably the file created in example one)

Then in a repeating loop (repeating as many times as required)

1. Call **pc_efilio_fpos_sector** to retrieve the raw sector number at the current file pointer and the number of contiguous sectors.
2. Simulate a DMA transfer from the disk with **pc_raw_read**, which reads contiguous sectors from the disk at the raw sector returned from **pc_efilio_fpos_sector**.
3. Call **pc_cfilio_read** with a null buffer and a read count equal to 512 times the number of sectors we just transferred. This instructs **pc_cfilio_read** to advance the file pointer.

A few things to keep in mind are:

1. No disk IO takes place during steps 1, and 3. So these steps are essentially real time processes.
2. Because this is only a simulation we are waiting for step 2 to complete before we perform step three and step one again to prepare for another DMA transfer. This synchronous operation is not necessary. The algorithm could be implemented in several other ways.

For example:

1. Step two could simply start a DMA transfer and return from the loop. Steps three and one could then be executed asynchronously when the DMA transfer completed.
2. Step two could start the DMA transfer and steps three and one could be then be executed immediately. This would have the affect of creating a DMA read behind scheme where the DMA circuit is armed to perform the transfer and the next sector number and sector count are ready so the DMA circuit can be re-armed immediately after completion of the current iteration. This can be repeated as many times as desired and provides a simple method of queuing up multiple unattended DMA transfers to read the file. The file could even be closed before the transfers complete.

```
void demonstrate_dma_with_cfile(byte *filename, BOOLEAN use_cfile_mode)
{
   int fd, i, driveno;
   dword cluster_size_bytes,ltemp, nwritten;
   dword dma_data_size_blocks;
   byte *dma_buffer;
   byte drive_name[8];
   EFILEOPTIONS my_options;
   dword raw_sector_number,raw_sector_count;

   driveno = pc_get_default_drive(drive_name);
   cluster_size_bytes = (dword) pc_cluster_size(drive_name);
   if (!cluster_size_bytes)
      return;  // ????

   // Open a circular file
   // Set the wrap point at megabyte
   rtfs_memset(&my_options, 0, sizEOF(my_options));
   my_options.circular_file_size_lo = (1024*1024);
   // Use a 1 megabyte minimum allocation and force
   // contiguous allocation so it's completely contiguous
   my_options.min_clusters_per_allocation =
      (my_options.circular_file_size_lo+cluster_size_bytes-1)/
      cluster_size_bytes;
   my_options.allocation_policy = PCE_FORCE_CONTIGUOUS;
   // The default is PCE_CIRCULAR_BUFFER mode, where the write
   // pointer can overtake the read pointer. In PCE_CIRCULAR_FILE
   // mode writes are truncated if the write pointer catches the
   // read pointer.
   if (use_cfile_mode)
      my_options.allocation_policy |= PCE_CIRCULAR_FILE;
```

```
fd = pc_cfilio_open((byte *)filename,
                (word)(PO_BINARY|PO_RDWR|PO_CREAT|PO_TRUNC)
                ,&my_options);
if (fd<0)
   return;

// Allocate a 128 k buffer
dma_data_size_blocks = 256;
dma_buffer  = (byte *) pro_test_malloc(dma_data_size_blocks*512);

// Start by writing zero bytes to the file. This should force a
// Pre-alloc
if (!pc_cfilio_write(fd, 0, 0, &ltemp))
   return;
// Check the readable raw sectors now there should be 0 blocks
if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                   &raw_sector_count))
   return;
if (raw_sector_count != 0)
   return;
// Check the writable raw sectors now there should be 2048 blocks
if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                   &raw_sector_count))
   return;
if (raw_sector_count != 2048)
   return;

if (!pc_raw_write(driveno,dma_buffer,raw_sector_number,256,TRUE))
  return;
// Check the readable raw sectors there should still be 0 blocks
if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                   &raw_sector_count))
   return;
// Now tell RtfsProPlus that we wrote 256 blocks
if (!pc_cfilio_write(fd, 0, (256*512), &nwritten))
   return;
// Check the readable raw sectors now there should be 256 blocks
if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                   &raw_sector_count))
   return;
if (raw_sector_count != 256)
   return;
// Check the writable raw sectors now there should be 2048-256 blocks
if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                   &raw_sector_count))
   return;
if (raw_sector_count != (2048-256))
   return;
// Now write 6 more chunks to get within 256 blocks of the wrap point
for (i = 0; i < 6; i++)
{
   if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                      &raw_sector_count))
```

```
        return;
    if (raw_sector_count < 256)
        return;
    if (!pc_raw_write(driveno,dma_buffer,raw_sector_number,256,TRUE))
        return;
    if (!pc_cfilio_write(fd, 0, (256*512), &nwritten))
        return;
}
// Make sure we are where we think we should be
if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                    &raw_sector_count))
    return;
if (raw_sector_count != 256)
    return;
// Now write 128 blocks
if (!pc_raw_write(driveno,dma_buffer,raw_sector_number,128,TRUE))
    return;
if (!pc_cfilio_write(fd, 0, (128*512), &nwritten))
    return;
// Ever vigilant, make sure RtfsProPlus indicates that we can only
// write 128 contiguous blocks
if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                    &raw_sector_count))
    return;
if (raw_sector_count != 128)
    return;
// Now write up to the wrap point
if (!pc_raw_write(driveno,dma_buffer,raw_sector_number,128,TRUE))
    return;
if (!pc_cfilio_write(fd, 0, (128*512), &nwritten))
    return;
// Check the readable raw sectors now there should be 2048 blocks
if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                    &raw_sector_count))
    return;
if (raw_sector_count != 2048)
    return;
// Check the writable raw sectors since we're back at the beginning
if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                    &raw_sector_count))
    return;
if (use_cfile_mode)
{// PCE_CIRCULAR_FILE mode. Should be no blcks available tp write
    if (raw_sector_count != 0)
        return;
}
// PCE_CIRCULAR_BUFFER mode. there should be 2048 blocks
else if (raw_sector_count != 2048)
    return;
// Now show that the writeable blocks track the read pointer
// in PCE_CIRCULAR_FILE mode.
if (use_cfile_mode)
{ // Now consume some blocks an make sure they are freed up for the
```

```
    // writer
    for (i = 1; i < 10; i++)
    {
      if (!pc_efilio_fpos_sector(fd, TRUE, TRUE, &raw_sector_number,
                                 &raw_sector_count))
          return;
      if (raw_sector_count < 128)
          return;
      if (!pc_raw_read(driveno,dma_buffer,raw_sector_number,128,TRUE))
          return;
      if (!pc_cfilio_read(fd, 0, (128*512), &nwritten))
          return;
      // Check the writable raw sectors
      if (!pc_efilio_fpos_sector(fd, FALSE, TRUE, &raw_sector_number,
                                  &raw_sector_count))
          return;
      if (raw_sector_count != (dword)(i * 128))
          return;
      }
  }

  pc_cfilio_close(fd);
  pro_test_free(dma_buffer);
}
```

Notes on Usage and Configuration

Configuring for asynchronous mounting.

HEREHERE

The following code fragment is a simple application which demonstrates the use of RtfsProPlus asynchronous mounting:

The application performs the following steps:

- Use the Rtfs callback interface to configure the drive for asynchronous mounting.
- Use the Rtfs callback interface to signal when asynchronous mount completion should be carried out.
- Spawn a thread that waits for a signal from the callback and then completes the mount.
- Run a dummy application loop that creates a directory and immediately removes it.
- Since asynchronous mounting is enabled, when the drive is not currently mounted the mkdir API call will initiate an asynchronous mount start and return failure.
- When mkdir fails for this reason the application waits for the background thread to complete the asynchronous mount and signal that the disk has been re-mounted and the application may resume.

To keep things simple we use a simple state variable to coordinate things.

The main points to notice are:

- The call to **pc_mkdir** fails because the drive is not configured to auto-mount
- When **pc_mkdir** fails **errno** is set to **PENOTMOUNTED**
- The failed **pc_mkdir** call initiates an asynchronous mount and invokes the callback routine with the starting argument set to 1
- The callback routine signals to the background thread that it should call **pc_async_continue** as many times as needed in order to complete the mount process
- When the mount completes, the callback routine is called again and signals that the mount is complete
- The application layer uses the signal to resume operation

To request asynchronous mounting you must modify the function rtfs_app_callback() in he applications rtfscalllback.c file.

The lines in this function that read:
case RTFS_CBA_ASYNC_MOUNT_CHECK:
break;

Should be changed to:
case RTFS_CBA_ASYNC_MOUNT_CHECK:
return(1);

To signal when asynchronous mount completion should be carried out you must modify the function rtfs_app_callback() in he applications rtfscalllback.c file.

The lines in this function that read:
case RTFS_CBA_ASYNC_START:
break;

Should be changed to the foreground thread to step the asynch state machine.

```
int rtfs_app_callback(int cb_code, int iarg0, int iargs1, int iargs2, void *pvargs)
{
   switch (cb_code) {
    case RTFS_CBA_INFO_MOUNT_STARTED:   /* iarg0 == drive number. Mount starting */
break;
                        case RTFS_CBA_INFO_MOUNT_FAILED:    /* iarg0 == drive number. Mount failed
*/
                        break;
                        case RTFS_CBA_INFO_MOUNT_COMPLETE:  /* iarg0 == drive number. Mount
succeeded */
                        break;
                        case RTFS_CBA_ASYNC_MOUNT_CHECK:          /* iarg0 == drive number. Return 1
to request asynchronous mounting */
                        break;
                        case RTFS_CBA_ASYNC_START:                /* iarg0 == cycle the statemachine
*/
                        break;
      case RTFS_CBA_ASYNC_DRIVE_COMPLETE: /* iarg0 == drive number. iarg1 == operation iarg2
== status */
                        break;
      case RTFS_CBA_ASYNC_FILE_COMPLETE:  /* iarg0 == file number. iarg1 == status */
                        break;
      case RTFS_CBA_DVR_EXTRACT_RELEASE:  /* iarg0 == file number. iarg1 == abort if 1,
overwritten if 0 */
                        break;
           }
           return(0);
}
```

The sample code follows:

```
int mount_state = 0; /* 0 == not mounted, 1 == in progress, 2 == complete */
static void __cdecl MountProc( void *args );
void demonstrate_async_mount_config(void);
void   demonstrate_mount_request_cb(int driveno, int starting);
int  driveno = 2; /* Running on the C: drive */
char *drivename = "C:";

void demonstrate_async_mount_application(void)
{
   mount_state = 0; /* not mounted */
   demonstrate_async_mount_config();
   _beginthread( MountProc, 0, NULL );
   for (;;)
   {
      if (!pc_mkdir((byte *)"test"))
      {
         if (get_errno() == PENOTMOUNTED)
         {
            printf("Mkdir failed. Wait for mount\n");
            while (mount_state != 2)
               rtfs_port_sleep(100);
         }
         else
         {
            printf("Some other error occured\n");
            return;
         }
      }
      else
      { /* Mkdir worked. remove the directory and close the disk out */
         pc_rmdir((byte *)"test");
         pc_dskfree(driveno);
      }
   }
}

void demonstrate_async_mount_config(void)
{
DRIVE_CONFIGURE config;

   rtfs_memset(&config, 0, sizEOF(config));
   /* Special asynchronous mount configurations. Everything else
      Is fairly generic */
   config.drive_operating_policy=DRVPOL_EXTENDED_IO|DRVPOL_ASYNC_MOUNT;
   config.drive_mount_request_cb = demonstrate_mount_request_cb;

   config.free_ctxt_num_hash_slots = 32;
   config.user_buffer_size_blocks = 128;
   config.num_fat_buffers = 2;
   config.num_dir_clusters = 0;
   config.num_small_file_clusters = 0;
   config.free_ctxt_region_hash_tbl =
```

```
        (void *) malloc(config.free_ctxt_num_hash_slots*sizEOF(void *));
    config.user_buffer =
        (byte *) malloc(config.user_buffer_size_blocks*512);
    config.fat_buffer_structures  =
       (struct fatbuff *) malloc(config.num_fat_buffers*sizEOF(struct fatbuff));
    config.fat_buffer_data =
        (byte *) malloc(config.num_fat_buffers*512);

    pc_diskio_configure((byte *)drivename, &config);
}

void   demonstrate_mount_request_cb(int driveno, int starting)
{
    if (starting)
    {
        printf("Callback signaling to start mount\n");
        mount_state = 1; /* in progress */
    }
    else
    {
        printf("Callback signaling mount completed\n");
        mount_state = 2; /* complete */
    }

}
```

Configuring and using RtfsProPlus for deterministic operation.

RtfsProPlus provides methods that control the maximum number of disk accesses made per API call and the amount of data transferred per disk access. When it is configured and used in this manner it provides a deterministic operating environment in which a single seek and data transfer of the underlying media controls the worst case latency of any individual call to RTFS. RtfsProPlus guarantees it will not introduce "hidden" disk accesses while it operates.

The rules for using RtfsProPlus in this manner are as follows:

- Use block aligned data sizes during file IO operations. Failure to do this can add up to two additional disk IO calls to a write call or one additional disk IO call to a read.
- If there are both synchronous and asynchronous fuctions to perform a task, choose the asynchronous method.
- Use cluster aligned data sizes during transactional file overwrites. Failure to do this can add up to four additional disk IO calls.
- If you are using **pc_cfilio_extract** try to extract data on cluster aligned boundaries. Failure to do this can add up to four additional disk IO calls.
- Always use asynchronous routines instead of their synchronous counterparts
- Set the **DRVPOL_ASYNC_MOUNT** policy bit in the disk configuration value **drive_operating_policy** and use asynchronous mounting techniques in the manner describe in the previous section.
- Set the **DRVPOL_NO_AUTOFLUSH** policy bit in the disk configuration value **drive_operating_policy**. This instructs RTFS to not add unexpected delays by automatically flushing buffers after successful completion of directory management API calls. It is now your responsibility to flush from the applications layer using the asynchronous flush API call.
- Make sure the user buffer that is set by the disk configuration value, **user_buffer_size_blocks,** is large enough to hold at least one cluster. This will almost certainly be the case, and it is advisable to set it much larger, but failure to do this will force directory maintenance routines to perform multiple individual block accesses.
- Avoid using **pc_deltree**, **pc_chksk** and **pc_enumerate**.

Rtfs Design Document

Extended File - Principals

RtfsProPlus provides an extended filio IO (efilio) package and a circular file (cfilio) package. The extended IO (efilio) package contains two components, a high performance, near zero latency packages for 32 bit file operations, and a 64 bit metafile IO package derived from the 32 bit file IO package with similar performance. The circular file (cfilio) package provides support for reading and writing from circular files and for efficiently extracting linear segments from circular files.

Extended File - Design Goals

The design goal for the extended IO package is to provide a high performance, reliable, deterministic file system capable of supporting 32 and 64 bit files. This underlying high performance file system has the following features:

Zero disk latency cluster allocation and linking - During write operations all cluster management is done on in-memory region caches. This allows write operations to operate in near real time. All disk operations that pertain to cluster management are deferred until the file is closed or flushed. The flush may be performed asynchronously.

Zero disk latency seek operations - A seek may be performed from any location in a file to any location in a file without ever accessing the disk. This makes file seek operations, essentially, real time operations. In addition this feature makes it practical to develop applications that rely on file seek operations but still perform deterministically.

Contiguous file support using cluster pre-allocation - The user may set the policy for file extent allocation. The policy may enforce guaranteed contiguous allocation of extents or it may request that the first free available fragments be used. In addition, the user can set a minimum allocation unit for the file to provide large contiguous file regions even when data is presented to the API in smaller quantities.

Optimized FAT access techniques – FAT tables are read and written via multi-block transfers to the disk whenever possible. A FAT cache is also maintained for frequently accessed FAT blocks.

Asynchronous Operation - Asynchronous versions are provided for all API calls that could potentially perform multiple disk accesses. Routines such as disk mount, file reopen, file flush and disk flush require multiple disk accesses and can introduce non-deterministic delays in the application.

Deterministic Operation – The combined asynchronous and Zero disk latency API calls provide a fully deterministic run time environment.

RtfsProPlus uses this underlying high performance file IO technology to implement a high performance 32/64 bit file IO package and a circular file IO for video and data acquisition applications.

The Freespace Manager

RtfsPro and **RtfsProPlus** both utilize a ram based free space manager. This subsystem stores disk free space maps and cluster chain maps in run length encoded lists. When a disk is mounted, the free space list is read from disk and converted to in-memory structures that run length encode the free space map. After this initial scan, RtfsPro and RtfsProPlus no longer access the disk or FAT cache to allocate disk blocks.

The File Fragment Manager

RtfsProPlus utilizes a ram based file extent manager. The extent maps of open files are stored in run length encoded lists. This makes it possible for **RtfsProPlus** to guarantee that no accesses of the disk or FAT cache are needed to seek within or append to the end of the file.

Asynchronous Operations

RtfsProPlus maintains internal run length encoded lists of allocated file extents and free disk blocks. These compact lists are memory based and thus provide deterministic operation for most operations.

There are however several non-deterministic operations that the library must perform. These operations synchronize RTFS's internal map with the external on-media FAT representation of the volume.

These operations include:

- **Disk mount** - When a disk is mounted RTFS must scan the file application table and preload some mapped views of the volume.
- **Failsafe Volume Synchronize** – Several regions of the volume are updated requiring multiple accesses.
- **Disk flush** – Several writes may be required.
- **File reopen** - When a file is reopened, if its cluster chain is not already cached, the chain must be read from the disk and converted to an internal form. (*Note: This delay may be eliminated by using the* **PCE_LOAD_AS_NEEDED** *option*)
- **File flush** - When a file has been expanded or contracted, when the file is flushed, the changes to the chain structure must be converted from the internal form to on-disk FAT representation.
- **File delete** - Deletes are really a combined reopen and flush pair, having the non-deterministic characteristics of both reopen and flush.

Rtfs Pro Plus provides asynchronous routines to perform these FAT synchronization operations. These routines provide a deterministic, iterative alternative to the standard synchronous method. When these asynchronous routines are used in combination with the deterministic extended file IO API, deterministic behavior can be assured for all disk management operations.

Asynchronous Volume Mount Operations

When the disk is first mounted, a scan of the file allocation table is performed. This routine is optimized to usually take less than ten seconds to run to completion, even on very large media. However, for many applications it is unacceptable to block the application thread for this long during the boot process or when new media is accessed.

Rtfs Pro Plus provides an asynchronous mount procedure to resolve this problem. The application begins the mount process by establishing the **DRVPOL_ASYNC_MOUNT** operating policy or by calling the **pc_diskio_async_mount_start** subroutine. The mount process is completed by the application repeatedly calling **pc_async_continue** until it reports completion or failure. The volume's files and directories will not be accessible while the asynchronous mount is completing but the application is free to interleave other tasks with the mount completion operation.

Asynchronous mounts provide a simple solution during the system boot process. The problem is a bit more complex to asynchronously re-mount hot swapped media. A set of guidelines and callback techniques are provided to aid in this. For more information on this see **pc_diskio_configure** in the API Reference Guide.

Asynchronous FAT Flush Operations

RtfsProPlus maintains a small memory based buffer pool of file allocation table pages. It must be flushed at regular intervals to synchronize changes made by the API with the disk image.

The number of pages requiring flushing is usually very small but the number is application dependant and the time required to flush all blocks is non-deterministic.

Normally RTFS automatically flushes the FAT cache at appropriate times such as when a directory is created or when a file is closed. These flushes cause small delays in execution that may not be tolerated in all applications.

To resolve this problem **RtfsProPlus** has an asynchronous FAT flush routine.

To use asynchronous FAT flushing the application must first disable automatic FAT flushing by calling **pc_diskio_configure**. The application can then schedule a background task to periodically call the asynchronous completion routine or it can insert calls to the routine in its round robin scheduling loop.

Asynchronous File Operations

During extended file operations that read, write, seek, and append files, RTFS

operates in a deterministic manner. RTFS guarantees that no disk accesses will be made to track or extend a file's extent map during these operations, so the disk accesses are limited to data transfer operations exclusively. These optimizations are made using an internal data representation, not the underlying FAT based disk representation.

File open and file flush operations are responsible for converting the file's FAT based cluster representation to the internal form when a file is reopened and converting it back to FAT based cluster representation when it is flushed.

On a typical FAT32 volume with a cluster size of 32 K, file cluster chains will consume approximately one FAT block per 2 megabytes of data. One gigabyte of data (2 million blocks) requires 256 blocks of the FAT table (128 Kbytes).

For larger files, the cluster chains can be long enough that the time to open, flush or delete files using synchronous methods can interfere with smooth execution of the application. To resolve this problem **RtfsProPlus** provides routines that perform these operations asynchronously. These routines break the operations into multiple separate smaller operations that consume less time per iteration. The routines may be called either as part of a round robin (polling) loop or from a separate thread. While asynchronous operations are in progress on a file most accesses to that file are prohibited, but asynchronous operations on directories and other files are allowed.

Asynchronous Journal File Operations

Journal file flush and commit operations may be run asynchronously to periodically flush the journal and synchronize the FAT volume with the Journal. Journal file flush operations are coordinated with other asynchronous operations to ensure that completed asynchronous operations won't be committed to the fat volume.

Circular Files

RtfsProPlus supports circular files through a set of **cfilio** API calls. Circular files are intended, mainly, to support developers in the important digital video segment. Though since circular files act as ring buffers to data blocks they may be used in any application that buffers data to disks.

Separate file pointers are maintained for read and write operations. These file pointers range from zero to the wrap point (circular file size) of the circular file. They map directly to the linear offset in the file of the read and write pointer. A stream view of the circular file is also provided. In stream view, the file pointers are the sixty four bit offsets of the read and write file pointers from the origin when the file was created. In stream view, the circular file provides a sliding window overlay on the circular buffer from the sixty four bit index of the last byte written backward to the 64 bit index of the oldest byte in the file. The read pointer may only reside within the current sliding window and stream view seek functions are provided to move both the read pointer and the write pointer within this sliding window.

Two variations of circular files are also provided, one mode is optimized for digital video and similar applications, and the other mode is optimized for data acquisition applications. The two modes differ only by how they handle data overrun conditions.

Circular files are designed to support digital video and similar applications. Circular files provide the ability to record data to the file while simultaneously allowing random access within the file to render the contents or to extract portions of the buffer to a permanent archive. A special API call is provided that extracts portions of the circular file to traditional linear files for permanent storage. When portions of the circular buffer have been extracted to permanent storage they temporarily exist in two locations, the circular file and the linear extract file. For example, a DVR device may be playing a 30 minute sequence of video from inside a twenty four hour video stream and the user may choose to archive the program for permanent storage while continuing to view the program. **RtfsPro** performs this extract operation in real time, in most cases without copying any data, and causing processor and disk bandwidth problems for the data collect and data display processes.

Circular files opened in circular file mode are more useful in traditional data acquisition applications. In circular file mode RTFS truncates circular file write operations if the write pointer will overtake the read pointer. The write call returns a short write count to the user indicating to the application that not all data could be written without losing data. This mode is useful for streaming data acquisition systems that must take data overruns into account. This may be either to log and clear the overrun error or to tune the data acquisition process by either increasing the buffer capacity or balancing the load. Buffering depths may be made arbitrarily large, allowing very deep buffering for high volume burst mode data acquisition applications. The application can also take advantage of the library to coordinate communications handshaking with the buffering process. The circular file stat function may be queried to determine how much unfilled buffer space is available between the current write pointer and the last unread data. Applications may use this depth indicator to hold off a burst mode data source while a data consumer process offloads data from the FIFO.

The FAT64 Sixty Four Bit Metafile API

Because disks are becoming larger and larger, bigger data sets, such as full length videos, are becoming common. 64 bit file support that goes beyond the FAT32 4-gigabyte file size barrier is necessary and **RtfsProPlus** provides a 64 bit abstraction layer that accomplishes this, by providing 64 bit files and a 64 bit capable File IO API while remaining FAT32 compatible.

FAT64 files are stored on disk, not as files, but as subdirectories that contain file data segments. The file's data is stored in one or more data segment files within the subdirectory containing the file's name. These segment files are simple 8.3 files named, "FAT64000.F64", "FAT64001.F64", "FAT64002.F64" and so on. Each of these files holds two gigabytes of data so 64 bit Meta files can grow to almost infinite length.

The FAT64 abstraction layer hides the complexity of 64 bit metafiles from the **RtfsProPlus** API so that the API calls needed for accessing 64 bit files are identical to the ones used for accessing standard 32 bit files. Because FAT64 file IO is really just segmented access to high performance 32 bit files there is only a marginal performance cost for using 64 bit files versus native 32 bit files.

These 64 bit files may be exchanged with standard PCs, or other FAT32 enabled devices.

The FAT64 abstraction is transportable to standard PCs but without some help, PC applications will not recognized FAT64 files as anything more than subdirectories containing multiple 2 gigabyte files.

Several approaches could be used to make FAT64 files transparently accessible to standard PC applications:

1. Utilities can be devised to copy a FAT64 file from a FAT32 volume to a single NTFS 64 bit file.
2. A file system driver plug-in can be devised to make a FAT64 enabled volume appear as a 64 bit capable file system.
3. Applications may be modified to be made FAT64 aware at the application layer, calling FAT64 aware libraries to perform file IO.

Rtfs Test Procedures and API programming Examples

Rtfs is tested using interactive test shells in combination with several automated regression tests.

Rtfs regression test procedures

The Rtfs test suite includes a baseline regression test suite and several unit test suites that verify correct operation of Rtfs subsystems.

The baseline regression test is part of the API and may either be invoked from the Rtfs command shell or by calling the tests entry point. It is documented on the following page.

The unit test suites are also included with Rtfs but they are not part of the standard API, they are normally controlled from the Rtfs interactive control shell.

The unit tests are described in the sections that follow the **pc_regression_test()** manual page.

A separate application note contains a tutorial on Rtfs interactive command shells and running Rtfs regression tests from the command shells

The pc_regression_test() API entry point

Basic	x	ProPlus	x
Pro	x	ProPlus DVR	x

FUNCTION

Rtfs baseline feature set regression test suite. This subroutine calls all or most of the API routines while stress testing the system for driver bugs, memory leaks and freespace leaks.

SUMMARY

#include <rtfs.h>
BOOLEAN pc_regression_test(byte *driveid, BOOLEAN do_clean)

driveid	Name of the volume "A:" "B:" etc.
do_clean	If **do_clean** is **TRUE**, the test suite will loop multiple times. The number of loops is determined by the compile time constants **INNERLOOP** and **OUTERLOOP** in the source file **apiregr.c**. If **do_clean** is **FALSE**, the test suite executes once and returns without deleting the files and directories that it created.

DESCRIPTION

This routine verifies the basic operation of the Rtfs. It, along with other tests described later, is used to verify Rtfs before it is released.
The regression test may be used in the field to validate ports to new targets and to stress test device drivers. The tests that are performed by **pc_regression_test()** are described in the next sections.

If **pc_regression_test()** detects an error condition it calls the routine **regress_error()** passing an integer containing one of the 30 or so possible error conditions. By default this routine prints the error number to the console and invokes **ERTFS_ASSERT_TEST().**

- To run the regression test from your application:
  - Make sure Rtfs is initialized by calling pc_ertfs_run()
  - Call pc_regression_test()
- To run the regression test from the command shell:
  - typing REGRESS D:
  - *Note: The next section provides examples of invoking this and other tests from the command shell.*

RETURNS

TRUE	Success
FALSE	Invalid drive specified in an argument. **ERTFS_ASSERT_TEST()** will loop endlessly or trap to a debugger for other errors.

Application Level Error Return Codes

None	

Rtfs regression test procedure reference

The source code for the command shells and regression tests may be found in the following files:

rtfscommon\source\apiregrs.c	Rtfs baseline feature set regression test suite.
rtfspackages\apps\prfstest.c	RTFS standard Failsafe regression test
rtfspackages\apps\preftest.c	RTFSProPlus Extended file and FAT64 regression test suite.
rtfspackages\apps\prasytest.c	RTFSProPlus asynchronous feature set regression test suite.
rtfspackages\apps\prcftest.c	RTFSProPlus circular file regression test suite.
rtfspackages\apps\prlinexttest.c	RTFSProPlus test of file cluster manipulation routines.
rtfspackages\apps\prtranstest.c	RTFSProPlus transaction file regression test suite
rtfscommon\apps\appcmdshrd.c	Interactive command shell for common Rtfs subroutines.
rtfscommon\apps\appcmdshwr.c	Interactive command shell for common Rtfs subroutines.
rtfscommon\apps\appcmdfs.c	Interactive command shell for common Failsafe routines
rtfspackages\apps\efishellrd.c	Interactive command shell for RtfsProPlus subroutines
rtfspackages\apps\efishellwr.c	Interactive command shell for RtfsProPlus subroutines

This application note describes:

- The automated tests that are performed on Rtfs before it is released.
- The procedures that perform the tests.
- The procedures in command shells that are used in the testing and development.
- This application notes may be used in conjunction with the source code described to find sample code that will do almost anything you can imagine doing with Rtfs.

The content of this document is derived from the source code. Each section lists the name of a source file and the tests that are performed by it. Within each section a description is presented of the tests that are performed.

The sections describing application shells are different. They provide the function entry point and a brief description of many of the commands that are provided by the command shells. These descritptions may be used to help select example code from the source code and to derive test scenarios for your application using the command shell.

Regression Test for features common to all Rtfs releases

Test File: rtfscommon/source/apiregress.c

- Procedure:BOOLEAN pc_regression_test()
- Description: Rtfs baseline feature set regression test suite.
  - This subroutine calls all or most of the API routines while stress testing the system for driver bugs, memory leaks and freespace leaks.
  - The routine may be inoked by typing REGRESS D: from the command shell

- Procedure: Basic File Regression Test pseudo-code
  - Check Disk Freespace
  - Make the test directory
  - loop
  - Step into the test directory
  - Make another subdiretory
  - loop
  - Make N deep subdirectories
    - Change into each
    - compare what we know is in the directory with
    - what pc_get_cwd returns.
  - End loop
  - In the lowest level directory perform file tests (explained below)
    - test long file names
    - test large files (4 Gig)
    - test file write with append mode
  - loop
    - create a file
    - open it with multiple file descriptors
    - loop
    - write to it on multiple FDs
    - loop
    - seek and read on multiple FDs, while testing values
    - flush
    - truncate
    - close
  - end loop
  - loop
    - rename file
  - delete file
  - end loop
  - either
    - loop N times
    - change to parent directory
    - delete subdirectory
    - end loop
  - or
    - deltree the test directory
  - Check Disk Freespace again and compare with original

- Procedure: Long file name test
  - Verify proper operation of file create, reopen and delete on files with name lengths that vary between 32 and 255 characters.
  - Verify attempting to create filename with a length greater that 255 characters fails.

- Procedure: Additional long file name tests
  - Verify proper alias name creation.
  - Verify proper handling of file names with all illegal characters.
  - Verify proper handling of all reserved names (null, con etc).
  - Verify reserved names with extensions. (null.xxx, con.xxx etc).
  - Verify proper handling of leading 0xE5 character in JIS file names.
  - Verify proper handling of 2 byte JIS characters at the beginning, middle and end of file names and file extensions.
  - The test performs the following operations.
    - The source code contains a table of long file names in both ASCII Jis and Unicode, and the alias name that should result when the long file name is created.
    - If the long name is illegal the associated alias name in the table also is.
    - If file creation using the long name fails and the alias is NON-NULL then it is an error
    - If file creation using the long name succeeds and the alias is NULL then it is an error
    - If file creation succeed then the alias name and the long name are written to the file.
    - The file is re-opened by it's long name and long file name and alias names are read in and compared to the table.
    - The file is re-opened by it's alias name and long file name and alias names are read in and compared to the table.
    - The file is re-opened by it's Unicode long name and long file name and alias names are read in and compared to the table.
    - The file name is searched for by it's long name with **pc_gfirst()** and the named are read using and long file name and alias names are read using **pc_gread()** and compared to the table.
    - The file name is searched for by it's alias with **pc_gfirst()** and the named are read using and long file name and alias names are read using **pc_gread()** and compared to the table.
    - The file name is searched for by it's Unicode name with **pc_gfirst()** and the named are read using and long file name and alias names are read using **pc_gread()** and compared to the table.
    - This process is repeated several times and in the process.
      - The file is created using it's long name and it's Unicode name.
      - All illegal sequences and reserved names are tested for JIS/ASCII and Unicode.
      - Files are reopened using long name, alias and Unicode names.
      - Files are deleted using long name, alias and Unicode names.

- Files are scanned using **pc_gfirst()** using long name, alias and Unicode names.

- *Note: The table contains approximately 250 file names and their aliases. Please consult the source code in **rtfscommon\source\apiregrs.c** for the subroutine **do_more_long_file_tests()** to view the table.*

- Procedure: Test correct operation of append
  - Verify that **po_open()** with append mode, correctly appends to the end of the file.
  - Verify imediately after a file is reopened.
  - Verify after the file pointer has been moved.

- Procedure: Test correct operation of **po_lseek()**
  - Verify that **po_lseek()** works correctly with negative and positive offsets.
  - Verify that **po_lseek()** processes error file pointer truncation correctly.
  - Verify that seek operations perform according to the following table:

Verify return values for various seek scenarios. Seeks are performed on a file that is 16 bytes long. Test with origin **PSEEK_CUR** start with the file pointer positioned midway through the file at offset 8.			
Seek origin	Seek offset	Expected return value	expected errno
PSEEK_SET	0	0	0
	-1	-1	PEINVALIDPARMS
	8	8	0
	16	16	0
	17	16	0
PSEK_CUR (from origin == 8)	0	8	0
	-9	0	0
	-8	0	0
	8	16	0
	9	16	0
PSEEK_END	0	16	0
	1	-1	PEINVALIDPARMS
	-9	9	0
	-16	0	0
	-17	0	0

- Procedure: Verify proper operation of files opened with buffering.
  - Open the same file twice, simultaneously in buffered mode.
  - Verify that the file buffer remains coherent when reads, writes and seeks are are performed on the two file descriptors.

- Procedure: Verify proper operation of the **pc_gread()** API function.
  - Write data to a file and verify that the data can be accessed during a directory enumeration using the **pc_gread()** API call.

- Procedure: Test correct behavior with large (4 Gig) files.
  - Verify proper operation of a 32 bit files when it is almost full (4 Gigabytes).
  - Verify that file writes and file extends behave appropriately when the maximum file size is reached.
  - Verify that file read behaves appropriately when the maximum file size is reached.
  - Verify that file seek behaves appropriately over the range including when the maximum file size is reached.

- Procedure: Test file behavior when handles are opened multiple times.
  - Test file create, reopen, close, flush and delete operations files.
  - Verify proper operation of exclusive and shared open modes.
  - Verify proper operation when a file is opened multiple times simulteously.
  - Verify proper operation of functions (delete, chsize) that require exclusive access to a file.

- Procedure: Test correct errno setting.
  - Every API call that is made by the basic regression test is followed by a check of the errno setting.
  - Verify that errno is cleared when there is no error.
  - Verify that errno contains the correct value when API calls return an error status.

- Procedure: Test correct operation of the check disk utility.
  - Create lost cluster chains and verify that checkdisk can find and remove the lost chains.
- Procedure:Cluster conversion test.
  - Verify proper operation of cluster to sector and sector to cluster conversion routines: **pc_diskio_info**, **pc_cluster_to_sector** and **pc_sector_to_cluster**.

Regression Tests for RtfsProPlus features

Test File: rtfspackages/apps/preftest.c

- Procedure: **pc_efilio_test(void)**
  - RTFSProPlus extended file and FAT64 regression test suite.
  - Test suite performs tests to verify the correct operation of features provided with RTFSProPlus.
- Procedure: **pc_efilio_test** - Extended IO regression test entry point
  - This routine calls subroutines that test RTFSProPlus subsystems for proper operations. Where appropriate tests are executed on both FAT64 files and normal files.
  - The following subroutines execute to test subsystems:

    - test_efile_open_options - Test extended file open options.
    - test_efile_data_tracking- Test extended file data operations.
    - test_asynchronous_api - Test asynchronous operations
    - test_fat64_aware_api - Verify correct FAT64 file awareness in the traditional RTFS FAT file system API.
    - Test correct operation of read write and seek and flush operatins on files opened using **PCE_LOAD_AS_NEEDED** option.
    - Test recovery modes when ProPlus runs out of regions structures during mount.
    - Test recovery modes when ProPlus runs out of regions structures with a mounted volume.
    - Test assigning specific clusters to files using **pc_efilio_setalloc** and **pc_diskio_free_lis**.
    - Verify placement using **pc_get_file_extents().**
    - Test option **PCE_FORCE_FIRST** for 32 and 64 bit file.
    - Test option **PCE_FORCE_CONTIGUOUS** for 32 and 64 bit file.
    - Test option **PCE_KEEP_PREALLOC** for 32 and 64 bit file.
- Procedure: **test_efile_data_tracking** - Verify extended read,write,seek operations on 32 bit and 64 bit files
    - Fill a file with a pattern and then perform tests to verify that:
    - The pattern can be read back
    - File reads stop at end of file
    - The pattern can be read back after the file is closed and reopened.
    - File pattern reads are succesful after seeks to the the following boundaries:
        - Beginning of file
        - End of file
        - 2 gigabyte segment boundaries in FAT64 files
    - Verify that file pattern reads are succesful after random seeks to offsets within the file.
    - Verify that all seek methods work, method rotates through: PSEEK_SET,PSEEK_CUR,PSEEK_CUR_NEG, and PSEEK_END.
    - Procedure: **test_fat64_aware_api** - Verify ERTFS API is FAT64 file aware.
    - Verify that the following ERTFS API calls correctly identify files as FAT64 files and perform appropriate FAT64 algorithms
        - **pc_stat**
        - **pc_gfirst**
        - **pc_mv**
        - **pc_unlink**
- Procedure: test chsize
    - Test proper operation of **pc_efilio_chsize()** on 32 and 64 bit files

Regression Test for Failsafe features

- Test File: rtfspackages/apps/prfstest.c

- Procedure: pc**_failsafe_test(void)**
    - Description: Test suite performs tests to verify the correct operation of features provided with RTFSProPlusFailsafe.

- The following basic tests ares performed by source code in the file named **prfstest.c:**
  - Tests that Journaling works correctly when a Journal file wrap occurs. When volume synchronize and Journal flush steps are done seperately, the Journal file will wrap and the session start location will be offset when the current frame record reaches the end of the Journal and the frame record at the beginning of the file has been synchronized. This is an unusual event so this test verifies that the wrap algorithm is working correctly.
  - Tests the use of **fs_api_cb_journal_fixed** places the Journal at a fixed place and size. This test creates a contiguous file on a volume and then sets the fixed Journal placement callback coordinates to the file's extents. It then starts Journaling and verifies the correct size and location of the Journal.
  - Tests correct behavior on disk full conditions - Two seperate tests verify that when Failsafe is Journaling to free space it properly resizes the Journal file when the disk becomes full.
  - Verify that the initial jounal size is reduced when Journaling is started on a nearly full disk.
  - Verify that the active Journal size is reduced when a disk becomes nearly full while Journaling
  - The tests are perfromed with and without using the memory based free manager
  - Tests Journal full condition. Verifies correct operation and proper error handling when all records in the Journal file are consumed.
  - Tests Journal file error conditions. Correct operation and proper error handling are tested for several simulated error conditions.
  - Out of date - Volume is changed with Journaling disabled and flushed records in the file.
  - Bad master record - Fields in the master record are manually corrupted and error handling is verified.
  - Bad frame record - Fields in frame records are manually corrupted and error handling is verified.
  - Tests other operating conditions
  - No flush - Verifies that changes that are not flushed are lost
  - Restore from disk - Verifies that flushed but unsynchronized volume changes can be restored
  - Synchronize from active session - Verifies that flushed volume changes can be synchronized
  - Restore after aborted synchronize - Verifies that aborted synchronizes may be completed by restore
  - Simultaneous Journal and syncronize - Verifies that volume changes can be Journaled to one segment while another segment is being synchronized.

- The following Failsafe tests are also available but they are performed by source code in the file named **prasytest.c**
  - Test that interrupting after Journal flush leaves the disk undamaged and unchanged and that the Journaled changes are be restored to the volume by a restore.
  - Simulate a write IO error during the asynchronous first Journal

flush of session.
- Then check Journal file status. The Journal file should be not valid or restore should be not required or recommend.
- Simulate write IO error during asynchronous Journal commit. Then check disk and look for lost chains or some other error.
- Check Journal status and verify restore is required. Now restore disk from Journal and verfiy lost chains or other errors are clear.
- Or invoke the "fs test" command from the RTFSProPlus command shell.

- Procedure: Perform random_test_asynchronous_api().
  - Description: This test verifies that Failsafe behave properly when IO errors are simulated on every block that is read or written during a sequence of operations.
  - Uses block access statistics to capture all block IO activity and set of becnchmarks that log the state of the volume
  - Calibrates itself by creating subdirectories and populating files, and issuing Journal flush and synchronize requests while gathering block access statistics and capturing a set of volume state bechmarks.
  - It then performs the same set of operations in a repetative loop and simulates an IO error for every block read and write request. For each simulated error it verifies that Failsafe performs correctly, either leaving the volume unchanged if no flushes occured before the error, or restoring the volume to state recorded after the last succesful synchronize operation.

- Procedure: Test shrink of Journal file due to disk filling
  - Enable failsafe
  - Create a directory
  - Check the size of failsafe reserved clusters
  - Allocate all free blocks from the memory manager
  - Check that failsafe reserved clusters are still reported as available
  - Create another directory
  - Make sure Failsafe file size was reduced by half and that free cluster counts are correct
  - Flush the Journal and abort
  - Verify that the first directory is on the volume, because of synchronization that occured during resize
  - Restore
  - Verify that the second directory is on the volume, because of restore
  - Verify that free cluster counts are correct in all steps
  - Release blocks from the memory manager
  - Test with free manager enabled and with it disabled

- Procedure: Test initial reduction of Journal file due to disk filling
  - Fill the disk with all but a few cluster
  - Start Journaling and verify that a reduced journal file is produced
  - Test with free manager enabled and with it disabled

Regression Test for asynchronous features

Test File: rtfspackages/apps/prasyest.c

- Description: RTFSProPlus asynchronous feature set regression test suite.
- Test suite performs tests to verify the correct operation of asynchronous features.
- Test suite performs tests to verify the correct operation Failsafe features.

- Procedure: **test_asynchronous** api
  - Verifies proper operation of asynchronous API
  - Verify proper operation of asynchronous volume mount and freespace scan.
  - Verify proper operation of signaling mechanism that an asynchronous mount requires.
  - Verify proper operation of asynchronous file open.
  - Verify proper operation of asynchronous file delete.
  - Verify proper operation of asynchronous file flush.
  - Verify proper operation of asynchronous disk flush.
  - Verify proper error handling and errno generation when drive API calls are made while an asynchronous mount is in progress

Additional Failsafe Regression Testing

Test File: rtfspackages/apps/prasyest.c

- Procedure: Failsafe restore coverage test - This test verifies proper operation of failsafe during every possible block IO operation.
  - Test the operation of Failsafe using compiled in DEBUG diagnostics that simulate disk read and write errors.
  - Provide a realistic simulation, simulated write errors overwrite the block they are writing to with nonsense data.
  - Enable Failsafe.
  - Perform a series of operations like mkdir, open, write and close to populate a subdirectory of a volume.
  - Use internal diagnostics to record the block number and directions of all blocks transfered to or from the disk.
  - Use internal diagnostics to store watchpoints of the sate of the volume after API calls.
  - Remove the subdirectory.
  - Perform the same series of operations but simulate read and write errors and use Failsafe to restore the volume.
  - Use internal diagnostics to simulate an IO error when a specific block number, block count, and direction is specified.
  - Verify that when errors are simulated after a Journal flush, the volume is restorable.
  - Verify that the restored state matches the proper calibrated watchpoint.
  - Verify that when errors are simulated before a Journal flush, the session is not restorable.
  - Verify that when errors are simulated before a Journal flush, the volume is un-changed.

    - Repeat this loop for every block number and direction that was recorded during the calibration phase

- Prodedure: Test that interrupting after Journal flush leaves the disk undamaged and unchanged and that the recorded Journal changes are restored to the volume by a restore.
    - Simulate a write IO error during the first asynchronous Journal flush of session.
    - Check Journal file status. The Journal file should be not valid or restore should be not required or recommend.
    - Simulate write IO error during asynchronous journal commit.
    - Check disk and look for lost chains or some other error.
    - Check Journal status and verify restore is required.
    - Restore disk from Journal.
    - Verify lost chains or other errors are clear.

- Procedure: **random_test_asynchronous_api()**.
    - Description: This test verifies that Failsafe behave properly when IO errors are simulated on every block that is read or written during a sequence of operations.
    - Use block access statistics to capture all block IO activity and set of becnchmarks that log the state of the volume.
    - Calibrates itself by creating subdirectories and populating files, and issuing Journal flush and synchronize requests.
    - While gathering block access statistics and capturing a set of volume state bechmarks. It then performs the same set of operations in a repetative loop and simulates an IO error for every block read and write request. For each simulated error it verifies that Failsafe performs correctly, either leaving the volume unchanged if no flushes occured before the error, or restoring the volume to state recorded after the last succesful synchronize operation.

Regression Tests for Direct cluster Manipulation Routines

Test File: rtfspackages/apps/prlinexttest.c

- Procedure: **test_efilio_extract(void)**
    - Description: Tests **pc_efilio_swap()**, **pc_efilio_extract()**, and **pc_efilio_remove()** file cluster manipulation routines.
    - Test suite performs tests to verify the correct operation of **pc_efilio_swap()**, **pc_efilio_extract()**, and **pc_efilio_remove()** on 32 and 64 bit files.
    - Files are created and filled with data that identifies each file's cluster.
    - Segments of files are moved and removed using the **pc_efilio_swap()**, **pc_efilio_extract()**, and **pc_efilio_remove()** API calls.
    - File chains and data are check to verify proper operation
        - Insert clusters 0-99 of a 100 cluster file in the front of a 10 cluster file
        - Insert clusters 0-99 of a 100 cluster file in the front of a 10 cluster file

- Insert clusters 0-99 of a 100 cluster file at the end of a 10 cluster file
- Insert clusters 0-9 of a 100 cluster file in the front of a 10 cluster file
- Insert clusters 1-10 of a 100 cluster file in the middle of a 10 cluster file
- Insert clusters 1-10 of a 100 cluster file at the end of a 10 cluster file
- Test some combinations of 64 bit files
- Insert clusters 0-99 of a of a 4 gig + 100 cluster file in front of a 10 cluster file
- Insert clusters 0-99 of a of a 4 gig + 100 cluster file at the end of a 10 cluster file
- Insert clusters 0-9 of a 4 gig + 100 cluster file in the front of a 10 cluster file
- Insert clusters 0-9 of a 4 gig + 100 cluster file in the front of a 10 cluster file
- Insert 2 gig-8 clusters of a gig + 100 cluster file to cluster 4 of a 2 gig-1 file
- Insert (no swap) 2 gig-8 clusters of a 4 gig + 100 to cluster 4 of a two_gig-8 cluster file
- Insert & swap 2 clusters from cluster (two_gig-1) of a 4 gig + 100 to cluster (two_gig-1) of a 4 gig + 100 file spans FAT64 segment break
- Insert & swap 1 clusters from cluster (two_gig -1) of a 4 gig + 100 to cluster (two_gig -1) of a 4 gig + 100 file at FAT64 segment 1 end
- Insert & swap 1 clusters from cluster (two_gig) of a 4 gig + 100 to cluster (two_gig) of a 4 gig + 100 file at FAT64 segment 2 start
- Insert & swap 100 clusters from cluster (two_gig-10) of a 4 gig + 100 to cluster (two_gig) of a 4 gig + 100 file insert 100 clusters at FAT64 segment 2 start

Regression Tests for circular file routines

Test File: rtfspackages/apps/prcftest.c

- Procedure: **pc_cfilio_test(void)**
- Description: RTFSProPlus circular file regression test suite. Test suite performs tests to verify the correct operation of circular file features.
- Verify proper operation of circular file API calls.
- Verfiy proper operation of circular file extract API calls.
- May be invoked from the "test" command of the RTFSProPlus command shell.
- The following subroutines executed to test subsystems:
  - **options_circular_file_efio_test** - Verify proper argument handling.
  - **circular_file_test** - Test proper functioning of circular file read, write and seek operations.
  - **circular_file_extract_test** - Test proper functioning of circular file extract

- Procedure:Test **pc_cfilio_open()**
  - Pass invalid arguments (wrap point equals zero),
  - Verify that the open fails and errno is **PEINVALIDPARMS**.
- Description: Test proper functioning of circular file read, write and seek operations
  - Use both normal files and FAT64 files.
  - Use both circular file open modes:
    - **PCE_CIRCULAR_BUFFER** - Write always succeeds.
    - **PCE_CIRCULAR_FILE** - Write truncates if it overtakes the read pointer.
  - Test both seek modes:
    - **pc_cstreamio_lseek** - Read and write pointers are 64 bit quantities that never reset. Offset zero is the first byte written to and the offset of EOF is the last byte written to the file. Valid offsets are the range: (last byte written - cirular file size) to (last byte written).
    - **pc_cfilio_lseek** - Read and write pointers are 64 bit quantities that reset when they reach the circular file wrap point. Offset zero is zero in the file and the offset of EOF is the cirular file size.
  - Verify that seek past EOF stops at EOF when EOF is not yet at the file wrap point.
  - Verify that **PSEEK_END** stops at EOF when EOF is not yet at the file wrap point.
  - Verify that **PSEEK_END** stops at beginning of file if offset is greater than current length of file, when EOF is not yet at the file wrap point.
  - Verify that seek past EOF stops at EOF when EOF is at the file wrap point.
  - Verify that seek past EOF using **PSEEK_CUR** stops at EOF when EOF is at the file wrap point.
  - Verify that read past EOF stops at EOF when EOF is at the file wrap point.
  - Verify that **PSEEK_END** stops at beginning of file if offset is greater than current length of file, when EOF is at the file wrap point.
  - Verify that seek past end stops at the correct logical EOF when the file has wrapped.
  - Verify that read stops at the correct logical EOF when the file has wrapped.
  - Verify that **PSEEK_END** starts at the correct logical EOF and truncates to the correct start of file when the file has wrapped.
  - Verify that **PSEEK_CUR_NEG** starts at the correct location and truncates to the correct start of file when the file has wrapped.
  - Verify that **PSEEK_CUR** starts at the correct location and truncates to the correct end of file when the file has wrapped.
  - Verify that stream mode seeks of to the start of the current data window if the poposed new logical file pointer is to the left of the data window.
  - Verify that stream mode seeks to the end of the current data window if the poposed new logical file pointer is to the right of the data window.

- Verify read and write tracking by verifying that reads from offsets in the file return the data put there by writes to the same offset
- Verify behavior when write pointer catches read pointer in **PCE_CIRCULAR_BUFFER** mode.
- Verify that the write pointer continues past the read pointer.
- Verify that after being overtaken, reads, return 0 bytes
- Verify that after being overtaken, seek places the read pointer at the oldest data in the file.
- Verify behavior when write pointer catches read pointer in **PCE_CIRCULAR_FILE** mode.
- Verify that writes stop when the write pointer stops reaches the read pointer.
- Verify that after reading some bytes that writes continue and then stop again when the write pointer reaches the new read pointer.
- Verify behavior of read and write when the file is wrapped by verifying that data that was written is read back appropriately.
- Verify that the read pointer stops when it reaches the write pointer when the file wrap pointer has been passed multiple times.

- Procedure: **circular_file_extract_test()**
  - Description: Test proper functioning of circular file extract using both normal files and FAT64 files perform the following tests.
  - Procedure One:
    - Fill a circular file with a known pattern.
    - Select a random offset and extract range within the file.
    - Extract the data to a linear file.
    - Compare the contents of the linear file and the region of the circular file it was extracted from. They should match.
  - Procedure Two:
    - Fill a circular file with a known pattern.
    - Select 10 random offsets and extract ranges within the file.
    - *Note: all or portions of the extract regions may overlap.*
    - Extract these extract regions to 10 linear files. Compare the contents of the linear files and the region of the circular file they were extracted from. They should match.

Regression Tests for transaction file overwrite routines

Test File: prtranstest.c

- Procedure: **test_efilio_transactions()**
  - Description: Regression test for RTFS transaction file operations on 32 bit and 64 bit files.
  - Test reading, writing and overwriting of file data with the file opened in **PCE_TRANSACTION_FILE** mode.
  - Simulate power interrupts by aborting disk mount sessions.
  - Verify that aborted transaction file overwrites are completed properly during system restore.
  - Perform the test using random file pointer offsets and random file write sizes.
  - Test transaction overwrites at various boundary conditions including:

        - Beginning of file
        - End of file
    - Test argument handling for **pc_efilio_open** with transaction support for 32 bit and 64 bit files.
    - With Failsafe disabled.
    - With no transaction buffer but legal size.
    - With transaction buffer but illegal size.
    - Everything legal but buffered.
    - Test illegal options for transaction files.
    - **PCE_KEEP_PREALLOC**
    - **PCE_CIRCULAR_FILE**
    - **PCE_CIRCULAR_BUFFER**
    - **PCE_REMAP_FILE**
    - **PCE_TEMP_FILE**
    - Open twice: once transactional, once not.

Command Shell for Common Rtfs functions

Test File: rtfscommon/apps/appcmdshrd.c

- Procedure: **_tst_shell()**
- Description: Interactive command commands that call common Rtfs subroutines.

- Procedure: **pc_shell_show_stats()**
- Displays the disk access statistics and time required to execute the previous command.

- Procedure: **doeject()** - Call a device driver to report device removal.
- Invoke by typing "EJECT" in the extended command shell.

- Procedure: **dounicode()** - Force the command shell to convert names to Unicode and call unicode versions of functions.
- Invoke by typing "UNICODE" in the extended command shell.

- Procedure: **dols()** - Perform a directory enumeration and display file information.
- Invoke by typing "DIR" in the command shell.

- Procedure: **doenum()** - Perform a recursive directory enumeration using **pc_enumerate()** and display file information.
- Invoke by typing "ENUM" in the command shell.

- Procedure: **docat()** - Display the contents of a named file.
- Invoke by typing "CAT" in the command shell.

- Procedure: **dodiff()** - Compare the contents of named files.
- Invoke by typing "DIFF" in the command shell.

- Procedure: **dostat()** - Call **pc_stat()** and display information about a named file.
- Invoke by typing "STAT" in the command shell.

- Procedure: **do_show_file_blocks()** - Call **pc_get_file_extents()** and display the cluster chain of a named named file.
- Invoke by typing "SHOWFILEEXTENTS" in the command shell.

- Procedure: **dodiskclose()** - Call **pc_diskclose()** to abort a mount.
- Invoke by typing "DSKCLOSE" in the command shell.

- Procedure: **doreset()** - Call **pc_ertfs_shutdown()** to shut down and reset Rtfs.
- Invoke by typing "RESET" in the command shell.

- Procedure: **do_show_disk_stats()** - Call **pc_diskio_info()** to retrieve and display extended information about the current volume.
- Invoke by typing "SHOWDISKSTATS" in the command shell.
- Test File: rtfscommon/apps/appcmdshwr.c.

- Procedure: **domkdir()** - Call **pc_mkdir()** to create a directory.
- Invoke by typing "MKDIR" in the command shell.

- Procedure: **dormdir()** - Call **pc_rmdir()** to delete a directory.
- Invoke by typing "RMDIR" in the command shell.

- Procedure: **dodeltree()** - Call **pc_dodeltree()** to recursively delete a directory.
- Invoke by typing "DELTREE" in the command shell.

- Procedure: **dorm()** - Call **pc_rm()** to delete a file. Use enumeration to delete multiple files using wildcard patterns.
- Invoke by typing "DELETE" in the command shell.

- Procedure: **domv()** - Call **pc_mv()** to move or rename a file.
- Invoke by typing "RENAME" in the command shell.

- Procedure: **dochkdsk()** - Call **pc_check_disk()** to check a disk's format.
- Invoke by typing "CHKDSK" in the command shell.

- Procedure: **dochsize()** - Call **po_chsize()** to truncate or grow a file.
- Invoke by typing "CHSIZE" in the command shell.

- Procedure: **docopy()** - Copy one named file to another.
- Invoke by typing "COPY" in the command shell.

- Procedure: **dofillfile()** - Create a file and fill it with a pattern. Useful for debugging, the file can later be read, copied, moved deleted etc.
- Invoke by typing "FILLFILE" in the command shell.

- Procedure: **doformat()** - Manage disk partitions and format volumes.
- Invoke by typing "FORMAT" in the command shell.

- Procedure**: do_show_disk_free()** - Call **pc_diskio_free_list()** to retrieve and display information about the freespace on current volume.
- Invoke by typing "SHOWDISKFREE" in the command shell.

- Procedure: **dotestopenspeed(int agc, byte **agv)**
- Description: Creates or re-opens up to 1000 files in a subdirectory.
- Reports total elapse time and average time per file.
- Invoked by typing "OPENSPEED" in the command shell.

Command Shell for Controlling Failsafe

Test File: rtfscommon/apps/appcmdshfs.c

- Description: Interactive commands that call Rtfs ProPlus failsafe subroutines.

- Procedure: **do_efile_failsafe()**
  - Description: Manually perform Failsafe commands
  - Invoke by typing "fs" in the extended command shell or "FS" in the basic command shell with the following options
    - abort - Abort the current mount without flushing
    - exit - Flush journal file and FAT stop journalling
    - enter - Begin Journalling
    - jcommit - Flush journal file but not FAT. Stops journalling and aborts, leaves a restorable Journal
    - jflush - Flush journal file but not FAT and continue journalling
    - commit - Flush journal file and synchronize FAT volume
    - info - Display information about current Failsafe Session
    - clear - Reset Journal IO statistics
    - restore - Restore the volume from current Journal File
    -

Command Shell for extended Rtfs functions

Test File: rtfspackages/apps/efishellrd.c

- Procedure: **efio_shell()**
- Description: Interactive command shell with commands that call Rtfs ProPlus extended API subroutines.

- Procedure: **do_efile_open()**
- Description: open or re-open a file using **pc_efilio_open().**
- All extended options are supported.
- Invoke by typing "open" in the extended command shell.

- Procedure: **do_efile_read()**
- Description: Read from a file

- Invoke by typing "read" in the extended command shell with the following options:
  - fileindex - The file index printed by the "open" or "copen" command.
  - resetfp - 'y' or 'Y' to seek to the beginning of the file before.
  - xferdata - 'y' or 'Y' to transfer data or 'n' to skip data transfer portion.
  - totalsize - Total number of bytes to read from the file, can be a 64 bit value.
  - readsize - Number of bytes to read per read call.

- Procedure: **do_efile_seek()**
- Description: Seek within a file.
- Invoke by typing "seek" in the extended command shell with the following options:
  - fileindex - The file index printed by the "open" or "copen" command
  - nseeks - Number of seeks to perform on the file (for viewing perfromance).
  - doread - 'y' To perform a one block read operation after every seek (forces a file read).
  - doxfer - If 'Y' and doread is true instructs the command read a block and force a disk seek.

- Procedure: **do_efile_close()**
- Description: Close a file.
- Invoke by typing "close" in the extended command shell with the following options:
  - fileindex - The file index printed by the "open" or "copen" command.
- Completes asynchronously if the command shell is in asynchronous completion mode.

- Procedure: **do_efile_stat()**
- Description: Performs a call to **pc_efilio_fstat()** and display information about the file.

- Procedure: **async_continue_proc()**
- Description: A background thread that completes asynchronous process by calling **pc_async_continue().**
- Used when the command shell is in asynchronous completion mode.

- Procedure: **do_efile_remount()**
- Description: Flushes, closes and remounts a volume.
- Completes asynchronously if the command shell is in asynchronous completion mode.
- Invoke by typing "remount" in the extended command shell.

- Procedure: **do_efile_set_buff()**
- Description: Changes the size of RtfsProPlus's user buffer.
- Helps in measuring the performance of Rtfs as a function of user buffer size.
- Invoke by typing "setbuff" in the extended command shell.

- Procedure: **do_efile_configure()**
- Description: Reconfigures a drive by calling **pc_diskio_configure().**

- Supports all configuration options.
- Invoke by typing "config" in the extended command shell.

- Procedure: **do_efile_async()**
- Description: Enable or disable asynchronous completion modes.
- If asynchronous mode is enable then shell commands use asynchronous methods if applicable.
- Invoke by typing "async" in the extended command shell.
- Supports "inline" completion in which the shell commands step the async engine until the command is completed.
- Supports "manual" completion in which the shell command "complete" must be called to step the async engine.
- Supports "background" completion where a thread perfroms completion (requires porting to most targets).

- Procedure: **do_efile_async_complete()**
- Description: Steps the async engine until all outstanding operations are completed.
- Invoke by typing "complete" in the extended command shell.

- Procedure: **demonstrate_dma_to_efile()**
- Description: This code fragment demonstrates the basic methods needed to use RtfsProPlus to write files using direct memory access.

- Procedure: **demonstrate_dma_from_efile()**
- Description: This code fragment demonstrates the basic methods needed to use RtfsProPlus to read files using direct memory access.

Test File: rtfspackages/apps/efishellwr.c

- Procedure: **efio_shell()**
- Description: Interactive commands that call RtfsProPlus extended API subroutines.

- Procedure: **do_efile_setallocation_hint()**
- Description: Calls **pc_efilio_setalloc()** to assign specific clusters to an open file.
- Invoke by typing "sethint" in the extended command shell.

- Procedure: **do_efile_writefile()**
- Description: Write to a file.
- Invoke by typing "write" in the extended command shell with the following options:
  - fileindex - The file index printed by the "open" or "copen" command.
  - resetfp - 'y' or 'Y' to seek to the beginning of the file before.
  - xferdata - 'y' or 'Y' to transfer from an uninitialized data buffer or 'n' to skip data transfer portion.
  - totalsize - Total number of bytes to read from the file, can be a 64 bit value.
  - writesize - Number of bytes to write per write call.
- Procedure: **do_efile_chsize()**
- Description: Calls **pc_efilio_chsize()** to shrink or grow a 32 or 64 bit file.
- Invoke by typing "chsize" in the extended command shell.

- Procedure: **do_efile_settime()**
- Description: Calls **pc_get_dirent_info()** and **pc_set_dirent_info()** to change the time field of a directory entry.
- Invoke by typing "settime" in the extended command shell.

- Procedure: **do_efile_setsize()**
- Description: Calls **pc_get_dirent_info()** and **pc_set_dirent_info()** to change the size field of a directory entry.
- Invoke by typing "setsize" in the extended command shell.

- Procedure: **do_efile_setcluster()**
- Description: Calls **pc_get_dirent_info()** and **pc_set_dirent_info()** to change the cluster field of a directory entry.
- Invoke by typing "setcluster" in the extended command shell.

- Procedure: **do_efile_flush()**
- Description: Flush a file.
- Invoke by typing "flush" in the extended command shell with the following options:
  - fileindex - The file index printed by the "open" or "copen" command.
- Completes asynchronously if the command shell is in asynchronous completion mode.

- Procedure: **do_efile_delete()**
- Description: Delete a 32 bit or 64 bit file.
- Invoke by typing "delete" in the extended command shell.
- Completes asynchronously if the command shell is in asynchronous completion mode.

- Procedure: **do_efile_tests()**
- Description: Invoke an rtfs Pro Plus regression test.
- Invoke by typing "test testname driveid" in the extended command shell
- Testname may be:
  - efile - Test Extendeded File API.
  - extract - Test pc_efext_extract, swap and remove.
  - cfile - Test Circular files.
  - async - Test Asycnhronous operation and Failsafe.
  - transaction - Test transaction files.
  - failsafe - Test Failsafe.
- Completes asynchronously if the command shell is in asynchronous completion mode.

- Procedure: **do_cfile_open()**
- Description: Open or re-open a circular file using pc_cfilio_open().
- All extended options are supported.
- Invoke by typing "copen" in the extended command shell.

- Procedure: **do_cfile_extract()**
- Description: Use **pc_cfilio_extract()** to extract a portion of a circular file into a linear extract file.
- Invoke by typing "extract" in the extended command shell.