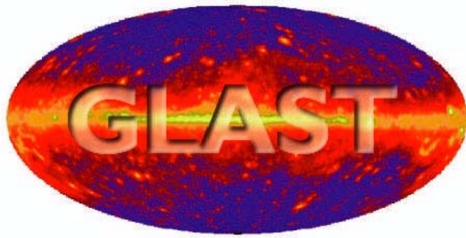


LAT Flight Software

FILE Package User Manual

Type:	User Manual
Version:	V2-2-0
Author:	D.L. Wood
Created:	5 November 2003
Updated:	1 October 2004
Printed:	1 October 2004

The user manual for the LAT flight software FILE package. Describes tools available for using LAT on-board files.

Contents

0 Introduction 4
0.0 LAT File Format 4
0.1 LAT File Header Format 5
0.2 File ID's 6
0.3 File Upload Procedure 7
0.4 Reference Documents 14

1 File Header Tools 15
1.0 File Header Library 15
1.1 File Header Executables 19
1.1.0 file_hdr_prefix 20
1.1.1 file_hdr_show 20

2 File System Tools 21
2.0 File System Library 21
2.1 File System Executables 23
2.2 tffs_show 23
2.3 tffs_boot_partition 23
2.4 tffs_boot_load 24
2.5 tffs_boot_dump 24

3 File ID and Path Tools 26

4 File Upload Tools 29
4.0 File Upload State Machine Library 29

5 Unit Testing 35
5.0 Unit Test Coverage 35
5.0.0 Constituent: file_hdr 35
5.0.1 Constituent: file_swap 35
5.0.2 Constituent: file_path 35
5.0.3 Constituent: file_upl 35
5.1 Running the Unit Test 36

Figures

Figure 1 - LAT On Board File General Format 4
Figure 2 - LAT File Header Format 5
Figure 3 - File ID Format 6
Figure 4 - File Upload State Diagram 8
Figure 5 - File Upload Data Packets 10
Figure 6 - File Upload Start Telecommand Format 11
Figure 7 - File Upload Cancel Telecommand Format 11
Figure 8 - File Upload Commit Telecommand Format 12
Figure 9 - File Upload Data Telecommand Format 13

Tables

Table 1 - File Storage Devices 7

Table 2 - File Upload CCSDS Telecommand Packets....9
Table 3 - File Header MSG Codes....19
Table 4 - File System MSG Codes....22
Table 5 - File Device Macros....26
Table 6 - File Upload MSG Codes....33

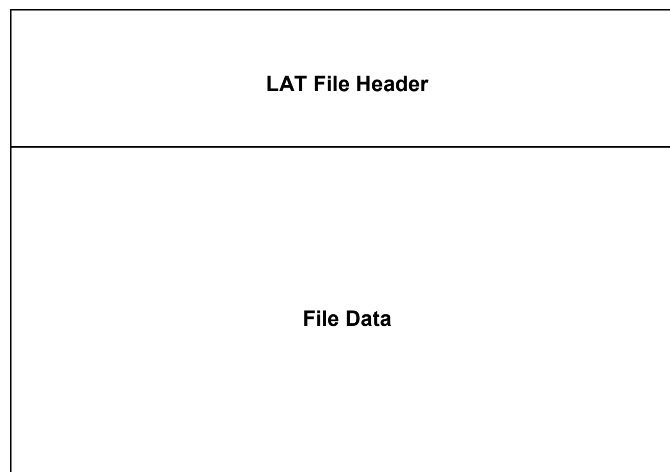
0 Introduction

The *FILE* package contains a collection of libraries and tools for handling on-board files for the LAT instrument. The constituents implement the file formats and policies as described in the LAT ICD and design documentation.

0.0 LAT File Format

Figure 1 shows the general format of LAT on-board files at the top level.

Figure 1 - LAT On Board File General Format



All files are expected to be prefixed with a file header. The file data contents are application specific.

0.1 LAT File Header Format

Figure 2 shows the format of the LAT file header.

Figure 2 - LAT File Header Format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">Header Checksum MSW</td></tr>
<tr><td colspan="16">Header Checksum LSW</td></tr>
<tr><td colspan="3">Version = 2</td><td>Cmp</td><td colspan="3">Spare</td><td colspan="9">Header Length = 28</td></tr>
<tr><td colspan="16">File Type</td></tr>
<tr><td colspan="16">File Key MSW</td></tr>
<tr><td colspan="16">File Key LSW</td></tr>
<tr><td colspan="16">File Data Checksum MSW</td></tr>
<tr><td colspan="16">File Data Checksum LSW</td></tr>
<tr><td colspan="16">File Data Length MSW</td></tr>
<tr><td colspan="16">File Data Length LSW</td></tr>
<tr><td colspan="16">File Creation Timestamp Seconds MSW</td></tr>
<tr><td colspan="16">File Creation Timestamp Seconds LSW</td></tr>
<tr><td colspan="8">File Name 0</td><td colspan="8">File Name 1</td></tr>
<tr><td colspan="8">File Name 2</td><td colspan="8">File Name 3</td></tr>
<tr><td colspan="8">File Name 4</td><td colspan="8">File Name 5</td></tr>
<tr><td colspan="8">File Name 6</td><td colspan="8">File Name 7</td></tr>
</table>

Header Checksum – The checksum of the 32 byte file header. The checksum is calculated using the ZLIB *adler32* algorithm.

Version – The file structure version number. This version of the FILE package only supports version 2 file headers.

Cmp – Indicates if the file data is compressed. A 0 will indicate the file data is not compressed and a 1 will indicate the file data is compressed. The compression method will be dependent on the type of data contained in the file.

Header Length - The number of bytes in the header, not including the file header checksum, which is 4 bytes. Set to 28 bytes. This makes the total file header size 32 bytes.

File Type – Indicates the file content type.

File Key – A 32-bit number which should uniquely define the file, including versioning, within the lifetime of the mission. Assigned by the file creator.

File Data Checksum – The checksum of the file data. The checksum is calculated using the ZLIB *adler32* algorithm.

File Data Length - The number of bytes in the data portion of the file.

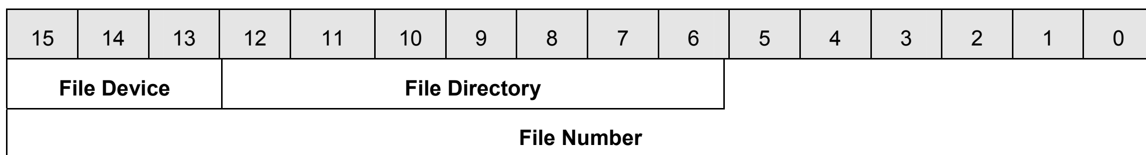
File Creation Timestamp – Time when the file was created by ground processing. The number of seconds elapsed since the time epoch.

File Name – Eight-character name of the file. As assigned by the file creator. If the file name is less than 8 characters then the name should be filled with the space character (0x20 ASCII).

0.2 File ID's

In the LAT command and telemetry system, target files are designated by a 32-bit file ID number, which encodes the path to the file in the target file system. This is how storage locations on the LAT flight system are defined. The format of the file ID word is shown in Figure 3.

Figure 3 - File ID Format



The File Device field designates a particular storage device. Each device number maps to a root device name on the target system. Only absolute paths (beginning with '/') are allowed. The LAT flight software does not maintain the concept of a current working path.

Table 1 - File Storage Devices

File Device Number	Name	Description
0	*/boot*	Boot device
1	*/ram*	RAM disk.
2	*/ee0*	SIB EEPROM partition 0
3	*/ee1*	SIB EEPROM partition 1
4	*/tmp*	Host temp file system.
5	-	Reserved.
6	-	Reserved.
7	-	Reserved.

Each device may have up to 127 directories The file ID is translated on board into a file system path of the form /*<device>*/d*xxx*/f*yyyyyyy*, where *xxx* is the directory number and *yyyyyyy* is the file number. For example file number 4 in directory 10 on EEPROM partition 0 would have a path name */ee0/d010/f0000004*.

The boot device designates file storage areas in SDRAM or EEPROM where the file is stored in absolute format at known addresses. No file system code is necessary to access these file areas. This feature is to support the use of files by the primary and secondary boot codes. The boot device has no defined directories, and the files may be designated by a file number alone.

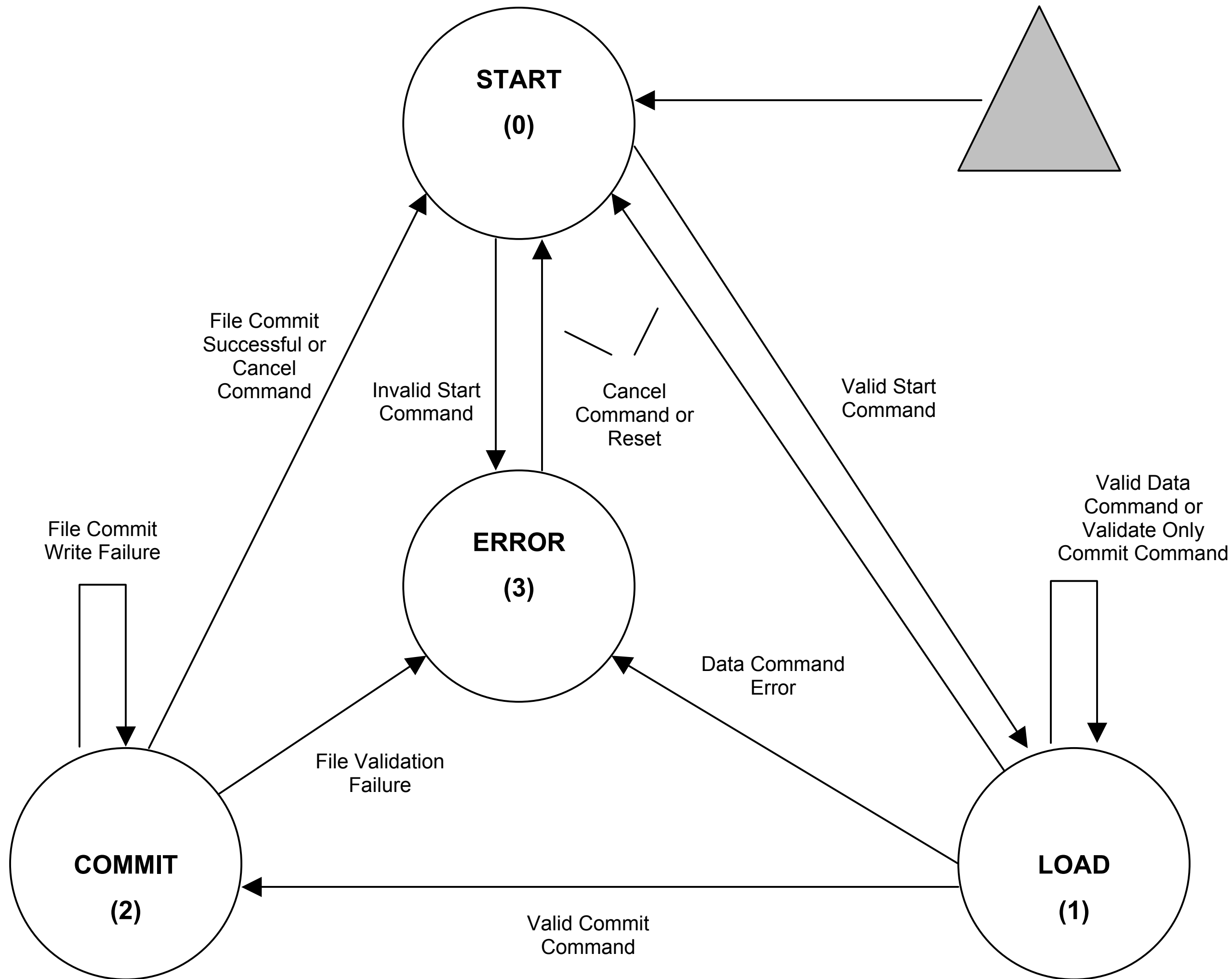
Some of the devices are only for testing and development purposes.

0.3 File Upload Procedure

The LAT file upload procedure involves transferring a file on the ground to a storage location on the LAT flight system. The complete file upload procedure is a coordinated effort among the LAT instrument operations, the GLAST mission operations, the GLAST spacecraft, and the LAT flight software. A file on the ground is ultimately split into multiple CCSDS telecommand packets, which are delivered to LAT flight software through the spacecraft uplink procedure. The *FILE* package provides a library which manages the file reconstruction from the telecommand packets into a coherent whole.

The file upload state diagram is shown below:

Figure 4 - File Upload State Diagram



The *FILE* package library provides several internal control features, as well as the ability to parse the contents of the CCSDS file upload telecommand packets. There are four telecommand packets understood by the file upload state machine library.

Table 2 - File Upload CCSDS Telecommand Packets

APID	Function Code	Description
0x641	0	File Upload Start. Announces the beginning of a new upload and provides total size. The File Upload Start command initializes the file upload state machine. The expected size in bytes comprising the upload is presented in this command. A new string of File Upload Data packets may be initiated after this command successfully completes. A previous upload must not be in progress when this command is sent. The file data commands that follow the upload start command will place file data in a temporary RAM upload buffer.
	1	File Upload Cancel. Cancels an outstanding upload set. The File Upload Cancel command deletes all of the data currently held in the temporary file upload buffer and resets the file upload state machine. This command may be used to cancel an erroneous upload attempt.
	2	File Upload Commit. Signals that the file upload data packet sequence is complete and that the file data is ready for validation and writing to storage. At this point the attached file header is validated, and the file data length and checksum extracted from the header. The file data checksum is validated over the received file data contained in the RAM upload buffer. Once the checks are successful, the file data is written to storage. Any padding which was added to the upload data set may be removed at this time, as the file header contains the original file data size. The File Upload Commit telecommand contains a file ID (see Section 0.2) indicating the storage location for the file. The “Validate Only” option bit for this telecommand lets operators see if the file data is ready for the commit without setting the state of the file upload to **ERROR** if the validation fails. Also, if the validation is successful, the state of the file upload is not set to **COMMIT**, but remains **LOAD**.
	3	File Upload Data. Actual file upload data string packet. Each upload arrives as a string of File Upload Data telecommand packets that use offset parameter to place the piece of file data within the file upload buffer. The file data is extracted sequentially from the original source file and inserted into consecutive upload telecommand packets, with the offset parameter marking the place within the file. As each CCSDS upload packet is received and validated, the file data contents contained in that packet are extracted and copied to the appropriate offset within the RAM file upload buffer.

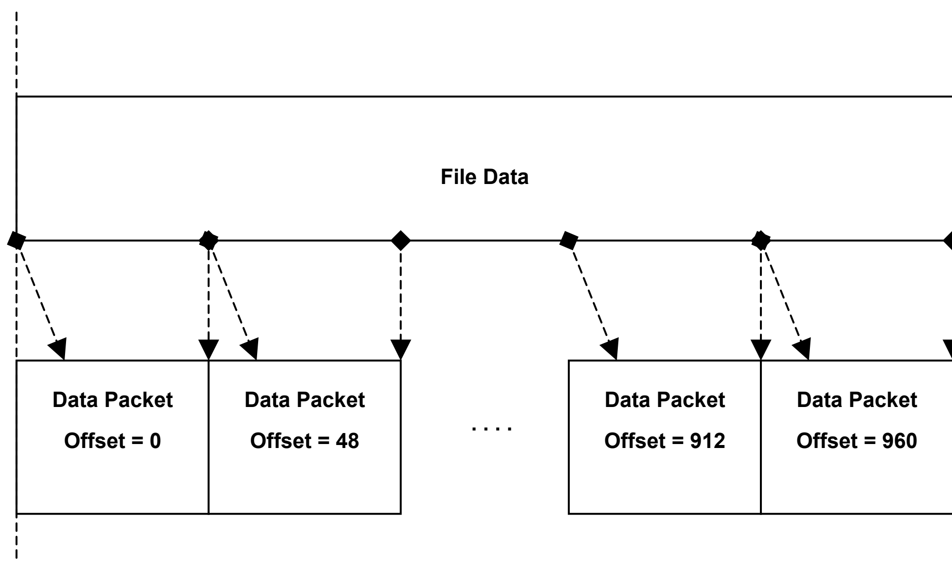
Each file upload is announced with a File Upload Start command. This command first checks to see if no upload is outstanding. The total bytes of file data expected is recorded. At this point, a File Upload Cancel command may be issued at any time to reset the upload process such that all currently received data is flushed and a new File Upload Start command is needed to restart the upload of a file. Once a complete set of File Upload Data packets has been received and validated, then a File Upload Commit command is needed to actually write the data to the final storage destination. Until this command is sent, the data will be held in a temporary buffer.

The contents of a file upload are always stored in a temporary buffer until an explicit File Upload Commit command has been received. The file data are never written directly into a storage location. In both boot and nominal commanding modes, only one temporary file upload buffer exists. Thus, only one upload may be in progress at a given time.

Each File Upload Data telecommand packet contains up to 48 bytes actual file data. The file data is extracted sequentially from the original source file and inserted into consecutive upload telecommand packets, with the offset parameter marking the place within the file. As each CCSDS upload packet is received and validated, the file data contents contained in that packet are extracted and copied to the appropriate offset within the RAM file upload buffer. From there, the file contents may copied to the storage device with a File Upload Commit command.

The example in Figure 5 below shows the construction of an upload transfer of 1000 bytes.

Figure 5 - File Upload Data Packets



The formats of the file upload telecommand packets are shown below.

Figure 6 - File Upload Start Telecommand Format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Version = 0</td><td>T=1</td><td>SH=1</td><td colspan="11">APID = 0x641</td></tr>
<tr><td colspan="2">SF</td><td colspan="14">Sequence Count</td></tr>
<tr><td colspan="16">Packet Length = 9</td></tr>
<tr><td>0</td><td colspan="15">Function Code = 0</td></tr>
<tr><td colspan="16">Upload File Size MSW</td></tr>
<tr><td colspan="16">Upload File Size LSW</td></tr>
<tr><td colspan="16">Packet Checksum</td></tr>
</table>

Upload File Size – The size in bytes of file data which will be contained in the entire upload sequence.

Figure 7 - File Upload Cancel Telecommand Format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Version = 0</td><td>T=1</td><td>SH=1</td><td colspan="11">APID = 0x641</td></tr>
<tr><td colspan="2">SF</td><td colspan="14">Sequence Count</td></tr>
<tr><td colspan="16">Packet Length = 5</td></tr>
<tr><td>0</td><td colspan="15">Function Code = 1</td></tr>
<tr><td colspan="4">LAT Unit</td><td colspan="12">Spare</td></tr>
<tr><td colspan="16">Packet Checksum</td></tr>
</table>

LAT Unit – A code indicating the LAT unit on which the file upload will be cancelled.

Figure 8 - File Upload Commit Telecommand Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Version = 0			**T=1**	**SH=1**	**APID = 0x641**										
SF		**Sequence Count**													
Packet Length = 9															
0	**Function Code = 2**														
LAT Unit				**Spare**											
Spare															**VO**
File Device			**File Directory**												
File Number															
Packet Checksum															

LAT Unit – A code indicating the LAT unit on which the file will be deleted.

VO – The validate only flag. If this bit is set, the file upload commit operation will only attempt to validate the file upload data. It will not continue through with the complete commit operation. If the validation fails, an error will be reported, but the file upload state machine will remain in the **LOAD** state.

File Device – A code indicating the file storage device.

File Directory - A code indicating the file storage directory.

File Number – A code indicating which file in a directory will be written with the contents of the file upload buffer.

Figure 9 - File Upload Data Telecommand Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="3">Version = 0</td><td>T=1</td><td>SH=1</td><td colspan="11">APID = 0x641</td></tr>
<tr><td colspan="2">SF</td><td colspan="14">Sequence Count</td></tr>
<tr><td colspan="16">Packet Length <= 55</td></tr>
<tr><td>0</td><td colspan="15">Function Code = 3</td></tr>
<tr><td colspan="16">File Offset MSW</td></tr>
<tr><td colspan="16">File Offset LSW</td></tr>
<tr><td colspan="16">File Upload Data</td></tr>
<tr><td colspan="16">Packet Checksum</td></tr>
</table>

File Offset – A offset in bytes from the beginning of the file that the first data byte in this packet represents.

0.4 Reference Documents

VxWorks Programmer's Guide 5.5, Wind River Systems, 1999.

Primary Boot Code, LAT Flight Software Design Description Document.

Secondary Boot Code, LAT Flight Software Design Description Document.

LTX User Manual, LAT Flight Software User Manual.

MSG User Manual, LAT Flight Software User Manual.

CCSDS User Manual, LAT Flight Software User Manual.

ZLIB User Manual, LAT Flight Software User Manual.

1 File Header Tools

All on board LAT files are expected to have a small file header attached. The file header always resides at the beginning of a file. The file header maintains useful information about the file which is employed by the LAT flight software and possibly by the mission operations components of the mission.

1.0 File Header Library

The constituent *file_hdr* provides a set of functions for creating and querying a file header in a user memory buffer. The library is available for all supported tags. The library functions always assume that the contents of the packet buffers are in big-endian byte order format.

The library functions all take a *buf* parameter to indicate where the file header is located in the user buffer. Note that the file header library functions provide no memory management capabilities; the functions simply act on buffer pointers provided by the caller. Moreover, the library functions provide no actual file management capabilities. It is the user's responsibility to call the appropriate file read and write functions to handle the retrieval and storage of the actual file header. It is expected that the *buf* parameter point to a buffer that is at least 2 byte (16-bit) aligned. The functions will check the parameter values and return an error if the alignment constraint is not met.

The library functions are concerned primarily with setting and extracting the various header field values.

The function *FILE_hdrCreate()* sets all of the header members at once. A "get" function is provided for most of the header members – for example *FILE_hdrGetChksum().*

This library provides no support for inserting, extracting, or manipulating the user application data portion of the file. The application data must follow contiguously the file header when the file is placed into storage

```
unsigned int FILE_hdrCreate(void *buf, unsigned int length, unsigned int time,
      unsigned int chksum, unsigned short type,
      unsigned int key, FILE_Hdr_Compression_Flag compress, const char *name);

     unsigned int FILE_hdrSizeof(void);

     unsigned int FILE_hdrVerify(const void *buf);

     unsigned int FILE_hdrGetLength(const void *buf, unsigned int *length);

     unsigned int FILE_hdrGetChksum(const void *buf, unsigned int *chksum);

     unsigned int FILE_hdrGetTime(const void *buf, unsigned int *time);

     unsigned int FILE_hdrGetType(const void *buf, unsigned short *type);

     unsigned int FILE_hdrGetKey(const void *buf, unsigned int *key);

     unsigned int FILE_hdrGetCompression(const void *buf, unsigned short
         *compress);

     unsigned int FILE_hdrGetName(const void *buf, char *name);
```

The short example below shows the creation and writing of a file header and file data after first calculating the file checksum data.

```
#include “FILE/FILE_hdr.h”
#include “ZLIB/zlib.h”
#include “MSG/MSG_pubdefs.h”
#include “PBS/MBA.h”

unsigned int status;
void *fileBuf, *hdrBuf;
unsigned int fileSize, hdrSize;
unsigned int chksum;
char fileName[FILE_HDR_NAME_SIZE + 1] = “MYFILE  ”;

/* allocate a header buffer */

hdrSize = FILE_hdrSizeof();
hdrBuf = MBA_alloc(hdrSize);

/* calculate file data checksum */

chksum = adler32(0, NULL, 0);
chksum = adler32(chksum, fileBuf, fileSize);

/* fill in file header */

 status = FILE_hdrCreate(hdrBuf, fileSize, fileTime, fileType, fileKey,
      FILE_HDR_UNCOMPRESSED, fileName);
if(_msg_success(status) == 0)
{
  /* error handler */
}

/* write out the file header */

write(outFile, hdrBuf, hdrSize);

/* write out the file data */

write(outFile, fileBuf, fileSize);
```

The next example shows reading in a file header and validating the file data checksum.

```
#include "FILE/FILE_hdr.h"
#include "ZLIB/zlib.h"
#include "MSG/MSG_pubdefs.h"
#include "PBS/MBA.h"

unsigned int status;
void *fileBuf, *hdrBuf;
unsigned int fileSize, fileChksum, hdrSize;
unsigned int chksum;

/* allocate a header buffer */

hdrSize = FILE_hdrSizeof();
hdrBuf = MBA_alloc(hdrSize);

/* read in the file header */

read(inFile, hdrBuf, hdrSize);

/* validate the file header */

status = FILE_hdrVerify(hdrBuf);
if(_msg_success(status) == 0)
{
 /* error handler */
}

/* get the file data length */

status = FILE_hdrGetLength(hdrBuf, &fileSize);
if(_msg_success(status) == 0)
{
 /* error handler */
}

/* allocate a file buffer */

fileBuf = MBA_alloc(fileSize);

/* read in the file data */

read(inFile, fileBuf, fileSize);

/* calculate file data checksum */

chksum = adler32(0, NULL, 0);
chksum = adler32(chksum, fileBuf, fileSize);

/* get the checksum stored in the file header */

status = FILE_hdrGetChksum(hdrBuf, &fileChksum);
if(_msg_success(status) == 0)
{
 /* error handler */
}
```

```
/* verify data checksum */

if(chksum != fileChksum)
{
  /* error handler */
}
```

The *FILE* header library supports the *MSG* status reporting system. The following small set of message codes is defined.

Table 3 - File Header MSG Codes

MSG Facility	MSG Code	Description	Parameter
FILE	FILE_SUCCESS	Success.	None.
	FILE _EHDRFORM	Header format error. An operation was attempted on a buffer that does not contain a valid file header.	The hex value of the file header flags word.
	FILE _EHDRPARM	A function call parameter value is out of range.	The name of the bad parameter.
	FILE _EHDRCKSM	The file header checksum validation failed.	The hex value of the calculated file header checksum.
	FILE_EHDRMEMA	The file header library failed to allocate internal scratch memory.	The number of bytes requested for the allocation.

The file header library signals all error messages at run time.

The FILE package also includes special builds of the file header library: *file_hdr_rtos* and *file_hdr_boot*. The user interface is identical for all variants. The *file_hdr_rtos* constituent is intended for linking with an RTOS executable image. The *file_hdr_boot* constituent is intended for linking with a primary boot code image. The variants contain less run time functionality than the application constituent *file_hdr*.

1.1 File Header Executables

A set of executables is provided in the *FILE* package for manipulating file headers attached to files in storage, particularly for host systems.

1.1.0 file_hdr_prefix

The *file_hdr_prefix* tool takes a file without a file header, and prefixes one to the file data, and write the pair to output. Numerous command line options allow the user to specify values for the file header members.

```
file_hdr_prefix [-c] [-s timestamp] [-t type] [-k key] [-n name] <in_file>
    <out_file>

    -c = Sets file compression flag in header.
    -s = The file timestamp as a decimal number.
    -t = The file type as a decimal number.
    -k = The file key as a decimal number.
    -n = The ASCII file name.  Must be <= 8 characters.
```

If optional arguments are not given, suitable but meaningless defaults are provided. The file data length and checksum values in the file header are automatically calculated based on the attributes of the input file.

1.1.1 file_hdr_show

The *file_hdr_prefix* tool takes a file with an attached file header and displays the header member values.

```
file_hdr_show <in_file>
```

The tool first runs a validation on the file header, so if one is not prefixed to the input file, an error message is printed.

2 File System Tools

The *FILE* package contains a collection of tools for managing target file systems. All target file systems are implemented using the VxWorks 5.5 DOSFS file system. Two types of underlying media are supported: RAM disks and TFFS disks. The RAM disks are simply the standard VxWorks implementation. The TFFS disks are implemented on top of EEPROM memory, available on the LAT SIB cPCI boards. All LAT VxWorks BSP's that support cPCI backplanes also include drivers to support the TFFS EEPROM disks.

2.0 File System Library

The constituent *file_sys* in package file contains functions that format and mount target file systems. The functions in the library are shown below.

```
FILE_sysRamCreate(void *addr, unsigned int size)

FILE_sysTffsMount(int drv)

FILE_sysTffsFormat(int drv, unsigned int offset)

FILE_sysTffsCheck(int drv)

FILE_sysTffsRepair(int drv)
```

The function *FILE_sysRamCreate()* performs a set of operations to create a standard VxWorks RAM disk containing a DOSFS file system. The function creates a new RAM disk device, formats a new DOSFS file system on it, and mounts the newly created file system. Note that the RAM disk partition is volatile; it must be re-created after every reboot and its contents are not preserved across reboots. The RAM disk file system is empty (no files or directories) after exit from the function. Users may specify a particular address for the partition with the *addr* parameter. If this parameter is set to *NULL*, then the memory is allocated from the system memory partition. The partition is given the device name */ram*, as described in Section 3.

The function *FILE_sysTffsMount()* mounts a previously formatted TFFS partition located within one of the SIB EEPROM banks. If the EEPROM bank is not properly formatted, this function will fail. The *drv* parameter specifies the TFFS drive number for the partition. The TFFS system numbers devices starting at '0'. When an SIB cPCI board is installed in the backplane, the lower EEPROM bank is registered as TFFS drive 0, while the upper EEPROM bank is registered as TFFS drive 1. The partition is given the device name either */ee0* or */ee1*, as described in Section 3.

The function *FILE_sysTffsFormat()* creates a new TFFS/DOSFS partition within one of the SIB EEPROM banks. The formatting is done in two stages. First, the TFFS formatting information is written to the EEPROM. The format parameters are set to allow an extra transfer block in the partition, which consumes some extra storage, but allows for complete failure of one of the other transfer blocks. Second, the standard DOSFS format information is written to the EEPROM bank. The partition must be mounted afterwards to be used. The *offset* parameter allows part of the EEPROM bank to be reserved for use outside of the file system. The *offset* value is the number of bytes from the base of the EEPROM bank at which the TFFS/DOSFS partition should begin. The offset value must be coordinated with any applications or tools which make use of the reserved region (see Section 2.3). This function should be used with care; it may destroy any previous formatting and content in the EEPROM bank. Note that this function can take an extremely long time to execute if the EEPROM bank is large.

The function *FILE_sysTffsCheck()* runs the VxWorks *chkdsk()* utility on one of the TFFS partitions. It will report an error if the file system metadata is inconsistent. The function *FILE_sysTffsRepair()* also runs the *chkdsk()* utility, but with the option for the utility to attempt to repair the file system metadata. Use this function with care, as the restoration process may result in the loss of user file data.

All of the file_sys constituent create and format functions automatically create the first 16 directories after the system level format is complete. After running a format functions, the user should be able to see in the device root directory subdirectories *d000* through *d015*.

The *file_sys* library supports the *MSG* status reporting system. The following small set of message codes is defined.

Table 4 - File System MSG Codes

MSG Facility	MSG Code	Description	Parameter
FILE	FILE_SUCCESS	Success.	None.
	FILE _ESYSFRMT	File system format error.	A string "TFFS" or "DOSFS" indicating whether the format operation failed at the TFFS or DOSFS layer.
	FILE _ESYSDEVC	File system device creation error.	A string "TFFS" or "DOSFS" indicating whether the device creation operation failed at the TFFS or DOSFS layer.

	FILE _ESYSMONT	File system mount error.	A string “TFFS” or “DOSFS” indicating whether the mount operation failed at the TFFS or DOSFS layer.
	FILE_ESYSCHK	The *chkdsk()* tool reported an error.	None.

Note that the create and mount functions are primarily for development and debugging. The LAT secondary boot code will handle these operations in the flight environment.

2.1 File System Executables

The constituent *file_tools* provides a set of VxWorks executables for working with the target TFFS/EEPROM file systems. These tools are fashioned as utilities which may be invoked from the VxWorks shell. It is expected that the functionality provided by these tools will be implemented by the command and telemetry system in the flight environment. In all cases, the *<drv>* argument specifies either the lower EEPROM bank (0) or the upper EEPROM bank (1).

2.2 tffs_show

The *tffs_show* tool displays the internal details of the TFFS formatting for a given TFFS/EEPROM partition.

```
tffs_show <drv>, <offset>
```

The erase unit header and block allocation map entries are shown for each erase unit in the partition. The *offset* parameter must match the offset value specified when the partition was formatted; otherwise, the display output may be erratic.

2.3 tffs_boot_partition

The *tffs_boot_partition* tool creates a bank header and boot partition at the start of an SIB EEPROM bank, as described in the primary boot code documentation.

```
tffs_boot_partition <drv>, <rtos_size>, <sbc_0_size>, <sbc_1_size>
```

The *xxx_size* parameters reserve the requested number of bytes for the file regions in the boot partition. If a TFFS partition is to follow the boot partition, the sizes must be carefully selected so that the boot partition and TFFS partition do not overlap. This tool should be used with care. It is normally used only when formatting an SIB EEPROM bank for the first time. If used after initial formatting, it is likely that the entire bank must be re-formatted. The bank header and boot partition are required if the EEPROM bank is to be used by the LAT primary and secondary boot codes.

2.4 tffs_boot_load

The *tffs_boot_load* tool copies a file into one of the boot partition file regions in an EEPROM bank.

```
tffs_boot_load <drv>, <rtos_file>, <sbc_0_file>, <sbc_1_file>
```

The EEPROM bank must have been previously formatted with a bank header and boot partition (see Section 2.3). The *xxx_file* parameters are file names of the files to load. These files must be accessible by one of the target file systems, or by a remote file system. Specifying *NULL* for one of the names skips the operation for the file region, allowing users to selectively load the different file regions. The files must be prefixed with a standard LAT file header.

2.5 tffs_boot_dump

The tffs_boot_dump tool copies a file out of one of the boot partition file regions in an EEPROM bank.

```
tffs_boot_dump <drv>, <rtos_file>, <sbc_0_file>, <sbc_1_file>
```

The EEPROM bank must have been previously formatted with a bank header and boot partition (see Section 2.3). The *xxx_file* parameters are file names of the files to receive the copied data. These files must be accessible by one of the target file systems, or by a remote file system. Specifying *NULL* for one of the names skips the operation for the file region, allowing users to selectively dump the different file regions. The files must be prefixed with a standard LAT file header.

Even if all file names are not given, the tool still attemps to read and parse not only the EEPROM bank header, but the file headers of all the boot partition file regions. If valid, the member values of these structures are displayed.

3 File ID and Path Tools

The *FILE* package contains a public header file "*FILE/FILE_defs.h*" which provide macro definitions for manipulating file ID values.

The file device name macros shown in Table 5 provide mappings between device ID numbers and target file system root path names. These names are standardized for LAT flight software systems.

Table 5 - File Device Macros

Device Number Macro	Device Name Macro	Root Path	Description
FILE_DEV_NUM_BOOT	FILE_DEV_NAME_BOOT	N/A	Special boot device designator.
FILE_DEV_NUM_RAM	FILE_DEV_NAME_RAM	/ram	RAM disk.
FILE_DEV_NUM_EE0	FILE_DEV_NAME_EE0	/ee0	TFFS drive 0 (SIB lower EEPROM bank)
FILE_DEV_NUM_EE1	FILE_DEV_NAME_EE1	/ee1	TFFS drive 1 (SIB upper EEPROM bank)
FILE_DEV_NUM_TMP	FILE_DEV_NAME_TMP	/tmp	Host temp file system.

Also included in the ID definition header are a set of translation macros for converting file ID values.

```
FILE_ID_TO_NUM(_id, _dev, _dir, _file)

FILE_NUM_TO_ID(_id, _dev, _dir, _file)

FILE_ID_TO_PATH(_str, _dev, _dir, _file)
```

The short example below shows some translations using these macros.

```
#include "FILE/FILE_defs.h"

char path[FILE_PATH_STR_SIZE];
unsigned int id, dev, dir, file;

/* create a file ID word from components */

FILE_NUM_TO_ID(&id, dev, dir, file)

/* extract components from a file ID word */

FILE_ID_TO_NUM(id, &dev, &dir, &file)

/* create file path /ee0/d002/f0000300 */

FILE_ID_TO_PATH(path, FILE_DEV_NAME_EE0, 2, 300)
```

Note that the *FILE_ID_TO_PATH()* macro requires that the user supply a device name. This probably means that a translation is needed from the device number. The macro *FILE_PATH_STR_SIZE* provides the maximum size in bytes of a LAT file path specification.

A set of functions providing the exact same capability and interface is presented in constituent *file_path*.

```
unsigned int FILE_pathIdToNum(unsigned int id, unsigned int *dev, unsigned int
      *dir, unsigned int *file);

unsigned int FILE_pathNumToId(unsigned int *id, unsigned int dev, unsigned int dir,
      unsigned int file);

unsigned int FILE_pathIdToPath(unsigned int id, char *str);
```

The function *FILE_pathIdToPath()*, unlike the macro *FILE_ID_TO_:PATH()*, performs the step of translating the device number into the device root path string. The functions in this constituent either return *FILE_SUCCESS* or *FILE_EPATPARM*, indicating an out of range function parameter.

The example below shows the typical usage of the *FILE_pathIdToPath()* function. The file ID value arrives in a telecommand, and the user needs to translate that to a file system path to open the file.

```
#include “FILE/FILE_path.h”
#include “MSG/MSG_pubdefs.h”

char path[FILE_PATH_STR_SIZE];
unsigned int myFileId;
unsigned int status;
int fd;

/* create file path string */

status = FILE_pathIdToPath(myFileId, path);
if(_msg_success(status) == 0)
{
  /* error handler */
}

/* open the file */

fd = open(path, O_RDONLY, 0);
if(fd == -1)
{
  /* error handler */
}
```

4 File Upload Tools

The *FILE* package contains a set of tools for managing the LAT flight software file upload process. The implementation matches the format and procedure expected by GLAST mission operations.

4.0 File Upload State Machine Library

The constituent *file_upl* implements the file upload state machine and telecommand parser as explained in Section 0.3.

Only one instance of the file upload state machine is allowed. This corresponds to the fact that only one upload may take place at once during mission operations. The library functions all take a *upl* parameter to indicate the global file upload state machine descriptor. This is an object of type *FILE_Upl*. Note that the file upload library functions provide no memory management capabilities; the functions simply act on buffer pointers provided by the caller. The function *FILE_uplGet()* may be called to retrieve a pointer to the global state machine descriptor.

The state machine first needs to be initialized with a call to *FILE_uplInit()*. In this function call, the user provides a pointer and size to the file upload assembly buffer. This effectively limits the maximum size of the file upload to the buffer size. The *file_upl* constituent uses this buffer to provide temporary storage of the file contents until the entire file data can be validated.

The *file_upl* constituent functions are listed below:

```
FILE_Upl* FILE_uplGet(void);

unsigned int FILE_uplInit(FILE_Upl *upl, void *fileBuf, unsigned int bufSize);

unsigned int FILE_uplReset(FILE_Upl *upl);

unsigned int FILE_uplInfo(FILE_Upl *upl, FILE_Upl_Info *info);

unsigned int FILE_uplPkt(FILE_Upl *upl, const void *pkt,
        FILE_Upl_Info *info);

unsigned int FILE_uplStart(FILE_Upl *upl, unsigned int size);

unsigned int FILE_uplData(FILE_Upl *upl, unsigned int offset, unsigned int
        size, const void *data);

unsigned int FILE_uplCancel(FILE_Upl *upl);

unsigned int FILE_uplCommit(FILE_Upl *upl, unsigned int id);
```

The *FILE_uplReset()* function returns the state machine to the **START** state at any time. It is equivalent to the file upload cancel telecommand.

The *FILE_uplInfo()* function, as well as others, fill in a user provided *FILE_Upl_Info* structure. This structure describes the current state of the file upload just after the last requested action was attempted. The *FILE_Upl_Info* structure is shown below.

```
typedef struct FILE_Upl_Info
{
 FILE_Upl_State state;
 unsigned int size_total;
 unsigned int size_current;
 unsigned int size_commit;
 unsigned int pkt_count;
 unsigned int offset_current;
 unsigned int error_code;
 unsigned int error_count;
 Void *buf_addr;
 unsigned int id_commit;
} FILE_Upl_Info;
```

The *state* member provides the current state of the file upload. The state values are shown below.

```
typedef enum _FILE_Upl_State
{
 FILE_UPL_STATE_START           = 0,
 FILE_UPL_STATE_LOAD            = 1,
 FILE_UPL_STATE_COMMIT          = 2,
 FILE_UPL_STATE_ERROR           = 3,
} FILE_Upl_State;
```

These values correspond with the state diagram in Figure 4.

The *size_total* member is the total number of bytes expected for the current file upload, and is set to '0' while in the **START** state. The *size_current* member is the total number of bytes received so far for the current file upload. Note that *size_current* may exceed *size_total* if retries are allowed on the file upload data packets. The *pkt_count* member is the count of File Upload Data telecommand packets processed by the state machine. It is set to '0' while in the **START** state. The *offset_current* member provides the offset value of the last successful File Upload Data telecommand. It is set to '0' on reset. The *error_code* member signals the most recent error condition encountered by the state machine. It is set to *FILE_SUCCESS* upon reset. The *error_count* member provides a count of the number of errors encountered since reset. The members *size_commit* and *id_commit* are only valid in the **COMMIT** state. The *size_commit* member provides an exact size in bytes, extracted from the uploaded file header, of the file data and header added together. This is the size that should be used in the actual commit write process. The *id_commit* member contains the file ID specifying the storage location for the file data.

The *FILE_uplPkt()* function performs most of the work of the library. It handles all the details of parsing the File Upload Start, Cancel, Commit, and Data telecommand packets (function codes 0, 1, 2, and 3). The user simply hands a pointer to the telecommand packet as a parameter. All updates are made to the file upload state machine, and any data in the telecommands is extracted into the file upload assembly buffer. The state of the file upload after attempting to process the packet is returned in the *info* structure.

The behavior of *FILE_uplPkt()* function when processing the File Upload Commit telecommand needs a bit of explaining. Because the actual commit write operation is specific to the environment (boot or application) the function cannot perform the full commit operation. Instead, the current contents of the file upload assembly buffer are validated. This involves validating the file header which should be prefixed at the head of the file data. Also, the file data checksum stored in the file header is compared against the checksum of the upload data. If the file upload data is good, the state is set to **COMMIT**, value *FILE_SUCCESS* is returned, and the *size_commit* member of the *info* structure is set to the total file size value. The user may then write the file data into storage, using the address in *info* member *buf_addr* to get access to the file upload data. At this time, the *id_commit* member of the info structure is also set. It contains the file ID value extracted from the telecommand packet parameters. It indicates the storage location for the file data.

The functions *FILE_uplStart()*, *FILE_uplData()*, *FILE_uplCancel()*, and *FILE_uplCommit()* provide an alternate interface to the file upload state machine. They perform the same actions as *FILE_uplPkt()*, but require the users to extract the telecommand parameters themselves before calling the functions. The parameters to these alternate functions are the same ones as found in the file upload telecommands.

The example below shows the basic usage of the *file_upl* functions.

```
#include "FILE/FILE_upl.h"
#include "FILE/FILE_upl_cmd.h"
#include "FILE/FILE_path.h"
#include "CCSDS/CCSDS_pkt.h"
#include "MSG/MSG_pubdefs.h"
#include "PBS/MBA.h"

unsigned int status;
unsigned int bufSize;
void *pkt, *fileBuf;
FILE_Upl *upl;
Unsigned short apid;
FILE_Upl_Info info;
char *path[FILE_PATH_STR_SIZE + 1];

/* allocate a file assemble buffer */
/* set maximum file upload size */

bufSize = MY_UPL_MAX_SIZE;
fileBuf = MBA_alloc(bufSize);

/* initialize the file upload state machine */

upl = FILE_uplGet();
status = FILE_uplInit(upl, fileBuf, bufSize)
if(_msg_success(status) == 0)
{
  /* error handler */
}

/* parse telecommand packets */

status = CCSDS_pktHdrGetFuncCode(pkt &funcCode)
if(_msg_success(status) == 0)
{
  /* error handler */
}

switch(funcCode)
{
  /* these function codes are handled completely by file_upl functions */

 case FILE_FC_FILE_LOAD_START:
  case FILE_FC_FILE_LOAD_DATA:
  case FILE_FC_FILE_LOAD_CANCEL:
  case FILE_FC_FILE_LOAD_COMMIT:

        status = FILE_uplPkt(upl, pkt, &info);
        if(_msg_success(status) == 0)
        {
           /* error handler */
        }

        /* check for COMMIT state to see if upload is complete */
```

```
            if(info.state == FILE_UPL_STATE_COMMIT)
            {
                  /* file is ready for write */
                  /* get storage location */

                  FILE_pathIdToPath(info.id_commit, path);

                  /* perform actual storage write operation */

                  fd = creat(path, O_WRONLY);
                  write(fd, info.buf_addr, info.size_commit);
                  close(fd);

                  /* reset state machine for new upload */

                  FILE_uplReset(upl);
            }
            break;
    }
```

The *FILE* upload library supports the *MSG* status reporting system. The following small set of message codes is defined.

Table 6 - File Upload MSG Codes

MSG Facility	MSG Code	Description	Parameter
FILE	FILE_SUCCESS	Success.	None.
	FILE _EUPLSTAT	A telecommand specified a state action that the state machine considers illegal.	The state of the file upload before the telecommand action was attempted.
	FILE _EUPLPARM	A function call parameter value is out of range.	The name of the bad parameter.
	FILE_EUPLAPID	A telecommand APID is not in the File Upload range.	The APID in the telecommand packet.
	FILE_EUPLFCOD	A telecommand function code is not in the File Upload range.	The function code in the telecommand packet.
	FILE _EUPLFSIZ	The file size specified in either the File Upload Start telecommand or in the file header is too large.	The out of range size value.

	FILE_EUPLPSIZ	A File Upload Data telecommand packet's offset and range place it outside the limit set by the File Upload Start telecommand.	The offset of the data.
	FILE_EUPLFHDR	The uploaded file header is invalid.	The value returned by *FILE_hdrVerify()*.
	FILE_EUPLFCHK	The calculated file data checksum does not match the value in the file header.	The calculated checksum value.
	FILE_IUPLSTAR	Information message announcing successful completion of a File Upload Start telecommand.	The size limit in bytes of the file to load.
	FILE_IUPLDATA	Information message announcing successful completion of a File Upload Data telecommand.	The offset of the data.
	FILE_IUPLRST	Information message announcing successful completion of a file upload state machine reset.	The state the file upload was in when reset.
	FILE_IUPLCOMM	Information message announcing successful completion of a *FILE_uplCommit()* function call.	The file ID value of the file to commit.

The file upload library signals all error messages at run time.

The *FILE* package also includes special builds of the file upload library *file_upl_boot*. The user interface is identical. The *file_upl_boot* constituent is intended for linking with a primary boot code image. It contains less run time functionality than the application constituent *file_upl*.

5 Unit Testing

The *FILE* package contains a unit test.

5.0 Unit Test Coverage

5.0.0 Constituent: file_hdr

The *FILE* package unit test performs two classes of unit test on the library *file_hdr*: tests for fault handling and tests for basic functionality. The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. The basic functionality tests ensure that all of the library functions can properly manipulate the LAT file header format members correctly. The functionality tests use some sample file headers to provide a reference.

5.0.1 Constituent: file_swap

The *FILE* package unit test also performs a basic functional test of the *file_swap* library. The sample file headers are checked by passing them through the swap functions. Depending on the platform running the test, the results are compared against the original file header data or a byte swapped version of the file header data.

5.0.2 Constituent: file_path

The *FILE* package unit test performs basic functional testing of the *file_path* library. The functional tests use sample file ID values and file paths to provide a reference. Sample file ID values and path strings are created.

5.0.3 Constituent: file_upl

The *FILE* package unit test performs basic functional testing of the *file_upl* library. The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. Also, illegal state transitions are tested to ensure that the state machine

always enters the **ERROR** state. Sample file upload telecommand packets are presented to the constituent functions, and fully valid file data sets are reconstructed from the packet fragments.

5.1 Running the Unit Test

The *FILE* package unit test may be run directly on UNIX hosts by typing *file_unit_test* at the command shell. If the proper modules are loaded, then the unit test may be run on VxWorks hosts by typing *file_unit_test* at the VxWorks shell prompt. The preferred method, however, is to use LTX. As part of the *FILE* package, a LTX script is provided which defines a test called *file_unit_test*.