RTEMS Filesystem Design Guide

Edition 4.9.6, for RTEMS 4.9.6

24 July 2011

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2009-08-14.15

The RTEMS Project is hosted at `http://www.rtems.com`. Any inquiries concerning RTEMS, its related support components, its documentation, or any custom services for RTEMS should be directed to the contacts listed on that site. A current list of RTEMS Support Providers is at `http://www.rtems.com/oarsupport`.

Table of Contents

Preface

This document describes the implementation of the RTEMS filesystem infrastructure. This infrastructure supports the following capabilities:

- Mountable file systems
- Hierarchical file system directory structure
- POSIX compliant set of routines for the manipulation of files and directories
- Individual file and directory support for the following:
    1. Permissions for read, write and execute
    2. User ID
    3. Group ID
    4. Access time
    5. Modification time
    6. Creation time
- Hard links to files and directories
- Symbolic links to files and directories

This has been implemented to provide the framework for a UNIX-like file system support. POSIX file and directory functions have been implemented that allow a standard method of accessing file, device and directory information within file systems. The file system concept that has been implemented allows for expansion and adaptation of the file system to a variety of existing and future data storage devices. To this end, file system mount and unmount capabilities have been included in this RTEMS framework.

This framework slightly alters the manner in which devices are handled under RTEMS from that of public release 4.0.0 and earlier. Devices that are defined under a given RTEMS configuration will now be registered as files in a mounted file system. Access to these device drivers and their associated devices may now be performed through the traditional file system open(), read(), write(), lseek(), fstat() and ioctl() functions in addition to the interface provided by the IO Manager in the RTEMS Classic API.

An In-Memory File System (IMFS) is included which provides full POSIX filesystem functionality yet is RAM based. The IMFS maintains a node structure for each file, device, and directory in each mounted instantiation of its file system. The node structure is used to manage ownership, access rights, access time, modification time, and creation time. A union of structures within the IMFS nodal structure provide for manipulation of file data, device selection, or directory content as required by the nodal type. Manipulation of these properties is accomplished through the POSIX set of file and directory functions. In addition to being useful in its own right, the IMFS serves as a full featured example filesystem.

The intended audience for this document is those persons implementing their own filesystem. Users of the filesystem may find information on the implementation useful. But the user interface to the filesystem is through the ISO/ANSI C Library and POSIX 1003.1b file and directory APIs.

1 Pathname Evaluation

This chapter describes the pathname evaluation process for the RTEMS Filesystem Infrastructure.

```
XXX Include graphic of the path evaluation process
```

1.1 Pathname Evaluation Handlers

There are two pathname evaluation routines. The handler patheval() is called to find, verify privlages on and return information on a node that exists. The handler evalformake() is called to find, verify permissions, and return information on a node that is to become a parent. Additionally, evalformake() returns a pointer to the start of the name of the new node to be created.

Pathname evaluation is specific to a filesystem. Each filesystem is required to provide both a patheval() and an evalformake() routine. Both of these routines gets a name to evaluate and a node indicating where to start the evaluation.

1.2 Crossing a Mount Point During Path Evaluation

If the filesystem supports the mount command, the evaluate routines must handle crossing the mountpoint. The evaluate routine should evaluate the name upto the first directory node where the new filesystem is mounted. The filesystem may process terminator characters prior to calling the evaluate routine for the new filesystem. A pointer to the portion of the name which has not been evaluated along with the root node of the new file system (gotten from the mount table entry) is passed to the correct mounted filesystem evaluate routine.

1.3 The rtems_filesystem_location_info_t Structure

The `rtems_filesystem_location_info_t` structure contains all information necessary for identification of a node.

The generic rtems filesystem code defines two global rtems_filesystem_location_info_t structures, the `rtems_filesystem_root` and the `rtems_filesystem_current`. Both are initially defined to be the root node of the base filesystem. Once the chdir command is correctly used the `rtems_filesystem_current` is set to the location specified by the command.

The filesystem generic code peeks at the first character in the name to be evaluated. If this character is a valid seperator, the `rtems_filesystem_root` is used as the node to start the evaluation with. Otherwise, the `rtems_filesystem_current` node is used as the node to start evaluating with. Therefore, a valid rtems_filesystem_location_info_t is given to the evaluate routine to start evaluation with. The evaluate routines are then responsible for making any changes necessary to this structure to correspond to the name being parsed.

```
struct rtems_filesystem_location_info_tt {
    void                                    *node_access;
    rtems_filesystem_file_handlers_r        *handlers;
    rtems_filesystem_operations_table       *ops;
    rtems_filesystem_mount_table_entry_t    *mt_entry;
};
```

node_access This element is filesystem specific. A filesystem can define and store any information necessary to identify a node at this location. This element is normally filled in by the filesystem's evaluate routine. For the filesystem's root node, the filesystem's initilization routine should fill this in, and it should remain valid until the instance of the filesystem is unmounted.

handlers This element is defined as a set of routines that may change within a given filesystem based upon node type. For example a directory and a memory file may have to completely different read routines. This element is set to an initialization state defined by the mount table, and may be set to the desired state by the evaluation routines.

ops This element is defined as a set of routines that remain static for the filesystem. This element identifies entry points into the filesystem to the generic code.

mt_entry This element identifies the mount table entry for this instance of the filesystem.

2 System Initialization

After the RTEMS initialization is performed, the application's initialization will be performed. Part of initialization is a call to rtems_filesystem_initialize(). This routine will mount the 'In Memory File System' as the base filesystem. Mounting the base filesystem consists of the following:

- Initialization of mount table chain control structure
- Allocation of a `jnode` structure that will server as the root node of the 'In Memory Filesystem'
- Initialization of the allocated `jnode` with the appropriate OPS, directory handlers and pathconf limits and options.
- Allocation of a memory region for filesystem specific global management variables
- Creation of first mount table entry for the base filesystem
- Initialization of the first mount table chain entry to indicate that the mount point is NULL and the mounted filesystem is the base file system

After the base filesystem has been mounted, the following operations are performed under its directory structure:

- Creation of the /dev directory
- Registration of devices under /dev directory

2.1 Base Filesystem

RTEMS initially mounts a RAM based file system known as the base file system. The root directory of this file system tree serves as the logical root of the directory hierarchy (Figure 3). Under the root directory a '/dev' directory is created under which all I/O device directories and files are registered as part of the file system hierarchy.

```
Figure of the tree structure goes here.
```

A RAM based file system draws its management resources from memory. File and directory nodes are simply allocated blocks of memory. Data associated with regular files is stored in collections of memory blocks. When the system is turned off or restarted all memory-based components of the file system are lost.

The base file system serves as a starting point for the mounting of file systems that are resident on semi-permanent storage media. Examples of such media include non- volatile memory, flash memory and IDE hard disk drives (Figure 3). File systems of other types will be mounted onto mount points within the base file system or other file systems that are subordinate to the base file system. The framework set up under the base file system will allow for these new file system types and the unique data and functionality that is required to manage the future file systems.

2.1.1 Base Filesystem Mounting

At present, the first file system to be mounted is the 'In Memory File System'. It is mounted using a standard MOUNT() command in which the mount point is NULL. This flags the mount as the first file system to be registered under the operating system and appropriate initialization of file system management information is performed (See figures 4 and 5). If a different file system type is desired as the base file system, alterations must be made to base_fs.c. This routine handles the mount of the base file system.

```
Figure of the mount table chain goes here.
```

Once the root of the base file system has been established and it has been recorded as the mount point of the base file system, devices are integrated into the base file system. For every device that is configured into the system (See ioman.c) a device registration process is performed. Device registration produces a unique dev_t handle that consists of a major and minor device number. In addition, the configuration information for each device contains a text string that represents the fully qualified pathname to that device's place in the base file system's hierarchy. A file system node is created for the device along the specified registration path.

```
Figure  of the Mount Table Processing goes here.
```

Note: Other file systems can be mounted but they are mounted onto points (directory mount points) in the base file system.

3 Mounting and Unmounting Filesystems

3.1 Mount Points

The following is the list of the characteristics of a mount point:

- The mount point must be a directory. It may have files and other directories under it. These files and directories will be hidden when the filesystem is mounted.
- The task must have read/write/execute permissions to the mount point or the mount attempt will be rejected.
- Only one filesystem can be mounted to a single mount point.
- The Root of the mountable filesystem will be referenced by the name of the mount point after the mount is complete.

3.2 Mount Table Chain

The mount table chain is a dynamic list of structures that describe mounted filesystems a specific points in the filesystem hierarchy. It is initialized to an empty state during the base filesystem initialization. The mount operation will add entries to the mount table chain. The un-mount operation will remove entries from the mount table chain.

Each entry in the mount table chain is of the following type:

```
struct rtems_filesystem_mount_table_entry_tt
{
   Chain_Node                             Node;
   rtems_filesystem_location_info_t       mt_point_node;
   rtems_filesystem_location_info_t       mt_fs_root;
   int                                    options;
   void                                  *fs_info;

   rtems_filesystem_limits_and_options_t  pathconf_limits_and_options;

  /*
   *  When someone adds a mounted filesystem on a real device,
   *  this will need to be used.
   *
   *  The best option long term for this is probably an
   *  open file descriptor.
   */
   char                                  *dev;
};
```

Node The Node is used to produce a linked list of mount table entry nodes.

mt_point_node The mt_point_node contains all information necessary to access the directory where a filesystem is mounted onto. This element may contain memory that is allocated during a path evaluation of the filesystem containing the mountpoint directory. The generic code allows this memory to be returned by unmount when the filesystem identified by mt_fs_root is unmounted.

mt_fs_root The mt_fs_root contains all information necessary to identify the root of the mounted filesystem. The user is never allowed access to this node by the generic code, but it is used to identify to the mounted filesystem where to start evaluation of pathnames at.

options XXX

fs_info The fs_info element is a location available for use by the mounted file system to identify unique things applicable to this instance of the file system. For example the IMFS uses this space to provide node identification that is unique for each instance (mounting) of the filesystem.

pathconf_limits_and_options
XXX

dev This character string represents the device where the filesystem will reside.

3.3 Adding entries to the chain during mount

When a filesystem is mounted, its presence and location in the file system hierarchy is recorded in a dynamic list structure known as a chain. A unique rtems_filesystem_mount_table_entry_tt structure is logged for each filesystem that is mounted. This includes the base filesystem.

3.4 Removing entries from the chain during unmount

When a filesystem is dismounted its entry in the mount table chain is extracted and the memory for this entry is freed.

4 System Call Development Notes

This set of routines represents the application's interface to files and directories under the RTEMS filesystem. All routines are compliant with POSIX standards if a specific interface has been established. The list below represents the routines that have been included as part of the application's interface.

1. access()
2. chdir()
3. chmod()
4. chown()
5. close()
6. closedir()
7. dup()
8. dup2()
9. fchmod()
10. fcntl()
11. fdatasync()
12. fpathconf()
13. fstat()
14. ioctl()
15. link()
16. lseek()
17. mkdir()
18. mkfifo()
19. mknod()
20. mount()
21. open()
22. opendir()
23. pathconf()
24. read()
25. readdir()
26. unmount()

The sections that follow provide developmental information concerning each of these functions.

4.1 access

File:

access.c

Processing:

This routine is layered on the stat() function. It acquires the current status information for the specified file and then determines if the caller has the ability to access the file for read, write or execute according to the mode argument to this function.

Development Comments:

This routine is layered on top of the stat() function. As long as the st_mode element in the returned structure follow the standard UNIX conventions, this function should support other filesystems without alteration.

4.2 chdir

File:

chdir.c

Processing:

This routine will determine if the pathname that we are attempting to make that current directory exists and is in fact a directory. If these conditions are met the global indication of the current directory (rtems_filesystem_current) is set to the rtems_filesystem_location_info_t structure that is returned by the rtems_filesystem_evaluate_path() routine.

Development Comments:

This routine is layered on the rtems_filesystem_evaluate_path() routine and the filesystem specific OP table function node_type().

The routine node_type() must be a routine provided for each filesystem since it must access the filesystems node information to determine which of the following types the node is:

- RTEMS_FILESYSTEM_DIRECTORY
- RTEMS_FILESYSTEM_DEVICE
- RTEMS_FILESYSTEM_HARD_LINK
- RTEMS_FILESYSTEM_MEMORY_FILE

This acknowledges that the form of the node management information can vary from one filesystem implementation to another.

RTEMS has a special global structure that maintains the current directory location. This global variable is of type rtems_filesystem_location_info_t and is called rtems_filesystem_current. This structure is not always valid. In order to determine if the structure is valid, you must first test the node_access element of this structure. If the pointer is NULL, then the structure does not contain a valid indication of what the current directory is.

4.3 chmod

File:

chmod.c

Processing:

This routine is layered on the open(), fchmod() and close() functions. As long as the standard interpretation of the mode_t value is maintained, this routine should not need modification to support other filesystems.

Development Comments:

The routine first determines if the selected file can be open with read/write access. This is required to allow modification of the mode associated with the selected path.

The fchmod() function is used to actually change the mode of the path using the integer file descriptor returned by the open() function.

After mode modification, the open file descriptor is closed.

4.4 chown

File:

chown.c

Processing:

This routine is layered on the rtems_filesystem_evaluate_path() and the file system specific chown() routine that is specified in the OPS table for the file system.

Development Comments:

rtems_filesystem_evaluate_path() is used to determine if the path specified actually exists. If it does a rtems_filesystem_location_info_t structure will be obtained that allows the shell function to locate the OPS table that is to be used for this filesystem.

It is possible that the chown() function that should be in the OPS table is not defined. A test for a non-NULL OPS table chown() entry is performed before the function is called.

If the chown() function is defined in the indicated OPS table, the function is called with the rtems_filesystem_location_info_t structure returned from the path evaluation routine, the desired owner, and group information.

4.5 close

File:

close.c

Processing:

This routine will allow for the closing of both network connections and file system devices. If the file descriptor is associated with a network device, the appropriate network function handler will be selected from a table of previously registered network functions (rtems_libio_handlers) and that function will be invoked.

If the file descriptor refers to an entry in the filesystem, the appropriate handler will be selected using information that has been placed in the file control block for the device (rtems_libio_t structure).

Development Comments:

rtems_file_descriptor_type examines some of the upper bits of the file descriptor index. If it finds that the upper bits are set in the file descriptor index, the device referenced is a network device.

Network device handlers are obtained from a special registration table (rtems_libio_handlers) that is set up during network initialization. The network handler invoked and the status of the network handler will be returned to the calling process.

If none of the upper bits are set in the file descriptor index, the file descriptor refers to an element of the RTEMS filesystem.

The following sequence will be performed for any filesystem file descriptor:

1. Use the rtems_libio_iop() function to obtain the rtems_libio_t structure for the file descriptor
2. Range check the file descriptor using rtems_libio_check_fd()
3. Determine if there is actually a function in the selected handler table that processes the close() operation for the filesystem and node type selected. This is generally done to avoid execution attempts on functions that have not been implemented.
4. If the function has been defined it is invoked with the file control block pointer as its argument.
5. The file control block that was associated with the open file descriptor is marked as free using rtems_libio_free().
6. The return code from the close handler is then passed back to the calling program.

4.6 closedir

File:

closedir.c

Processing:

The code was obtained from the BSD group. This routine must clean up the memory resources that are required to track an open directory. The code is layered on the close() function and standard memory free() functions. It should not require alterations to support other filesystems.

Development Comments:

The routine alters the file descriptor and the index into the DIR structure to make it an invalid file descriptor. Apparently the memory that is about to be freed may still be referenced before it is reallocated.

The dd_buf structure's memory is reallocated before the control structure that contains the pointer to the dd_buf region.

DIR control memory is reallocated.

The close() function is used to free the file descriptor index.

4.7 dup() Unimplemented

File:

dup.c

Processing:

Development Comments:

4.8 dup2() Unimplemented

File:

dup2.c

Processing:

Development Comments:

4.9 fchmod

File:

fchmod.c

Processing:

This routine will alter the permissions of a node in a filesystem. It is layered on the following functions and macros:

- rtems_file_descriptor_type()
- rtems_libio_iop()
- rtems_libio_check_fd()
- rtems_libio_check_permissions()
- fchmod() function that is referenced by the handler table in the file control block associated with this file descriptor

Development Comments:

The routine will test to see if the file descriptor index is associated with a network connection. If it is, an error is returned from this routine.

The file descriptor index is used to obtain the associated file control block.

The file descriptor value is range checked.

The file control block is examined to determine if it has write permissions to allow us to alter the mode of the file.

A test is made to determine if the handler table that is referenced in the file control block contains an entry for the fchmod() handler function. If it does not, an error is returned to the calling routine.

If the fchmod() handler function exists, it is called with the file control block and the desired mode as parameters.

4.10 fcntl()

File:

fcntl.c

Processing:

This routine currently only interacts with the file control block. If the structure of the file control block and the associated meanings do not change, the partial implementation of fcntl() should remain unaltered for other filesystem implementations.

Development Comments:

The only commands that have been implemented are the F_GETFD and F_SETFD. The commands manipulate the LIBIO_FLAGS_CLOSE_ON_EXEC bit in the `flags` element of the file control block associated with the file descriptor index.

The current implementation of the function performs the sequence of operations below:

1. Test to see if we are trying to operate on a file descriptor associated with a network connection
2. Obtain the file control block that is associated with the file descriptor index
3. Perform a range check on the file descriptor index.

4.11 fdatasync

File:

fdatasync.c

Processing:

This routine is a template in the in memory filesystem that will route us to the appropriate handler function to carry out the fdatasync() processing. In the in memory filesystem this function is not necessary. Its function in a disk based file system that employs a memory cache is to flush all memory based data buffers to disk. It is layered on the following functions and macros:

- rtems_file_descriptor_type()
- rtems_libio_iop()
- rtems_libio_check_fd()
- rtems_libio_check_permissions()
- fdatasync() function that is referenced by the handler table in the file control block associated with this file descriptor

Development Comments:

The routine will test to see if the file descriptor index is associated with a network connection. If it is, an error is returned from this routine.

The file descriptor index is used to obtain the associated file control block.

The file descriptor value is range checked.

The file control block is examined to determine if it has write permissions to the file.

A test is made to determine if the handler table that is referenced in the file control block contains an entry for the fdatasync() handler function. If it does not an error is returned to the calling routine.

If the fdatasync() handler function exists, it is called with the file control block as its parameter.

4.12 fpathconf

File:

fpathconf.c

Processing:

This routine is layered on the following functions and macros:

- rtems_file_descriptor_type()
- rtems_libio_iop()
- rtems_libio_check_fd()
- rtems_libio_check_permissions()

When a filesystem is mounted, a set of constants is specified for the filesystem. These constants are stored with the mount table entry for the filesystem. These constants appear in the POSIX standard and are listed below.

- PCLINKMAX
- PCMAXCANON
- PCMAXINPUT
- PCNAMEMAX
- PCPATHMAX
- PCPIPEBUF
- PCCHOWNRESTRICTED
- PCNOTRUNC
- PCVDISABLE
- PCASYNCIO
- PCPRIOIO
- PCSYNCIO

This routine will find the mount table information associated the file control block for the specified file descriptor parameter. The mount table entry structure contains a set of filesystem specific constants that can be accessed by individual identifiers.

Development Comments:

The routine will test to see if the file descriptor index is associated with a network connection. If it is, an error is returned from this routine.

The file descriptor index is used to obtain the associated file control block.

The file descriptor value is range checked.

The file control block is examined to determine if it has read permissions to the file.

Pathinfo in the file control block is used to locate the mount table entry for the filesystem associated with the file descriptor.

The mount table entry contains the pathconf_limits_and_options element. This element is a table of constants that is associated with the filesystem.

The name argument is used to reference the desired constant from the pathconf_limits_and_options table.

4.13 fstat

File:

fstat.c

Processing:

This routine will return information concerning a file or network connection. If the file descriptor is associated with a network connection, the current implementation of `fstat()` will return a mode set to `S_IFSOCK`. In a later version, this routine will map the status of a network connection to an external handler routine.

If the file descriptor is associated with a node under a filesystem, the fstat() routine will map to the fstat() function taken from the node handler table.

Development Comments:

This routine validates that the struct stat pointer is not NULL so that the return location is valid.

The struct stat is then initialized to all zeros.

rtems_file_descriptor_type() is then used to determine if the file descriptor is associated with a network connection. If it is, network status processing is performed. In the current implementation, the file descriptor type processing needs to be improved. It currently just drops into the normal processing for file system nodes.

If the file descriptor is associated with a node under a filesystem, the following steps are performed:

1. Obtain the file control block that is associated with the file descriptor index.
2. Range check the file descriptor index.
3. Test to see if there is a non-NULL function pointer in the handler table for the fstat() function. If there is, invoke the function with the file control block and the pointer to the stat structure.

4.14 ioctl

File:

ioctl.c

Processing:

Not defined in the POSIX 1003.1b standard but commonly supported in most UNIX and POSIX system. Ioctl() is a catchall for I/O operations. Routine is layered on external network handlers and filesystem specific handlers. The development of new filesystems should not alter the basic processing performed by this routine.

Development Comments:

The file descriptor is examined to determine if it is associated with a network device. If it is processing is mapped to an external network handler. The value returned by this handler is then returned to the calling program.

File descriptors that are associated with a filesystem undergo the following processing:

1. The file descriptor index is used to obtain the associated file control block.
2. The file descriptor value is range checked.
3. A test is made to determine if the handler table that is referenced in the file control block contains an entry for the ioctl() handler function. If it does not, an error is returned to the calling routine.
4. If the ioctl() handler function exists, it is called with the file control block, the command and buffer as its parameters.
5. The return code from this function is then sent to the calling routine.

4.15 link

File:

link.c

Processing:

This routine will establish a hard link to a file, directory or a device. The target of the hard link must be in the same filesystem as the new link being created. A link to an existing link is also permitted but the existing link is evaluated before the new link is made. This implies that links to links are reduced to links to files, directories or devices before they are made.

Development Comments:

Calling parameters: const char *existing const char *new

link() will determine if the target of the link actually exists using rtems_filesystem_evaluate_path()

rtems_filesystem_get_start_loc() is used to determine where to start the path evaluation of the new name. This macro examines the first characters of the name to see if the name of the new link starts with a rtems_filesystem_is_separator. If it does the search starts from the root of the RTEMS filesystem; otherwise the search will start from the current directory.

The OPS table evalformake() function for the parent's filesystem is used to locate the node that will be the parent of the new link. It will also locate the start of the new path's name. This name will be used to define a child under the parent directory.

If the parent is found, the routine will determine if the hard link that we are trying to create will cross a filesystem boundary. This is not permitted for hard-links.

If the hard-link does not cross a filesystem boundary, a check is performed to determine if the OPS table contains an entry for the link() function.

If a link() function is defined, the OPS table link() function will be called to establish the actual link within the filesystem.

The return code from the OPS table link() function is returned to the calling program.

4.16 lseek

File:

lseek.c

Processing:

This routine is layered on both external handlers and filesystem / node type specific handlers. This routine should allow for the support of new filesystems without modification.

Development Comments:

This routine will determine if the file descriptor is associated with a network device. If it is lseek will map to an external network handler. The handler will be called with the file descriptor, offset and whence as its calling parameters. The return code from the external handler will be returned to the calling routine.

If the file descriptor is not associated with a network connection, it is associated with a node in a filesystem. The following steps will be performed for filesystem nodes:

1. The file descriptor is used to obtain the file control block for the node.
2. The file descriptor is range checked.
3. The offset element of the file control block is altered as indicated by the offset and whence calling parameters
4. The handler table in the file control block is examined to determine if it contains an entry for the lseek() function. If it does not an error is returned to the calling program.
5. The lseek() function from the designated handler table is called with the file control block, offset and whence as calling arguments
6. The return code from the lseek() handler function is returned to the calling program

4.17 mkdir

File:

mkdir.c

Processing:

This routine attempts to create a directory node under the filesystem. The routine is layered the mknod() function.

Development Comments:

See mknod() for developmental comments.

4.18 mkfifo

File:

mkfifo.c

Processing:

This routine attempts to create a FIFO node under the filesystem. The routine is layered the mknod() function.

Development Comments:

See mknod() for developmental comments

4.19 mknod

File:

mknod.c

Processing:

This function will allow for the creation of the following types of nodes under the filesystem:

- directories
- regular files
- character devices
- block devices
- fifos

At the present time, an attempt to create a FIFO will result in an ENOTSUP error to the calling function. This routine is layered the filesystem specific routines evalformake and mknod. The introduction of a new filesystem must include its own evalformake and mknod function to support the generic mknod() function. Under this condition the generic mknod() function should accommodate other filesystem types without alteration.

Development Comments:

Test for nodal types - I thought that this test should look like the following code:

```
if ( (mode & S_IFDIR) = = S_IFDIR) ||
     (mode & S_IFREG) = = S_IFREG) ||
     (mode & S_IFCHR) = = S_IFCHR) ||
     (mode & S_IFBLK) = = S_IFBLK) ||
     (mode & S_IFIFO) = = S_IFIFO))
     Set_errno_and_return_minus_one (EINVAL);
```

Where:

- S_IFREG (0100000) - Creation of a regular file
- S_IFCHR (0020000) - Creation of a character device
- S_IFBLK (0060000) - Creation of a block device
- S_IFIFO (0010000) - Creation of a FIFO

Determine if the pathname that we are trying to create starts at the root directory or is relative to the current directory using the rtems_filesystem_get_start_loc() function.

Determine if the pathname leads to a valid directory that can be accessed for the creation of a node.

If the pathname is a valid location to create a node, verify that a filesystem specific mknod() function exists.

If the mknod() function exists, call the filesystem specific mknod() function. Pass the name, mode, device type and the location information associated with the directory under which the node will be created.

4.20 mount

File:

mount.c

Arguments (Not a standard POSIX call):

rtems_filesystem_mount_table_entry_t **mt_entry,

If the mount operation is successful, this pointer to a pointer will be set to reference the mount table chain entry that has been allocated for this file system mount.

rtems_filesystem_operations_table *fs_ops,

This is a pointer to a table of functions that are associated with the file system that we are about to mount. This is the mechanism to selected file system type without keeping a dynamic database of all possible file system types that are valid for the mount operation. Using this method, it is only necessary to configure the filesystems that we wish to use into the RTEMS build. Unused filesystems types will not be drawn into the build.

char *fsoptions,

This argument points to a string that selects mounting for read only access or read/write access. Valid states are "RO" and "RW"

char *device,

This argument is reserved for the name of a device that will be used to access the filesystem information. Current filesystem implementations are memory based and do not require a device to access filesystem information.

char *mount_point

This is a pathname to a directory in a currently mounted filesystem that allows read, write and execute permissions. If successful, the node found by evaluating this name, is stored in the mt_entry.

Processing:

This routine will handle the mounting of a filesystem on a mount point. If the operation is successful, a pointer to the mount table chain entry associated with the mounted filesystem will be returned to the calling function. The specifics about the processing required at the mount point and within the filesystem being mounted is isolated in the filesystem specific mount() and fsmount_me() functions. This allows the generic mount() function to remain unaltered even if new filesystem types are introduced.

Development Comments:

This routine will use get_file_system_options() to determine if the mount options are valid ("RO" or "RW").

It confirms that a filesystem ops-table has been selected.

Space is allocated for a mount table entry and selective elements of the temporary mount table entry are initialized.

If a mount point is specified: The mount point is examined to determine that it is a directory and also has the appropriate permissions to allow a filesystem to be mounted.

The current mount table chain is searched to determine that there is not another filesystem mounted at the mount point we are trying to mount onto.

If a mount function is defined in the ops table for the filesystem containing the mount point, it is called at this time.

If no mount point is specified: Processing if performed to set up the mount table chain entry as the base filesystem.

If the fsmount_me() function is specified for ops-table of the filesystem being mounted, that function is called to initialize for the new filesystem.

On successful completion, the temporary mount table entry will be placed on the mount table chain to record the presence of the mounted filesystem.

4.21 open

File:

open.c

Processing:

This routine is layered on both RTEMS calls and filesystem specific implementations of the open() function. These functional interfaces should not change for new filesystems and therefore this code should be stable as new file systems are introduced.

Development Comments:

This routine will allocate a file control block for the file or device that we are about to open.

It will then test to see if the pathname exists. If it does a rtems_filesystem_location_info_t data structure will be filled out. This structure contains information that associates node information, filesystem specific functions and mount table chain information with the pathname.

If the create option has been it will attempt to create a node for a regular file along the specified path. If a file already exists along this path, an error will be generated; otherwise, a node will be allocated for the file under the filesystem that contains the pathname. When a new node is created, it is also evaluated so that an appropriate rtems_filesystem_location_info_t data structure can be filled out for the newly created node.

If the file exists or the new file was created successfully, the file control block structure will be initialized with handler table information, node information and the rtems_filesystem_location_info_t data structure that describes the node and filesystem data in detail.

If an open() function exists in the filesystem specific handlers table for the node that we are trying to open, it will be called at this time.

If any error is detected in the process, cleanup is performed. It consists of freeing the file control block structure that was allocated at the beginning of the generic open() routine.

On a successful open(), the index into the file descriptor table will be calculated and returned to the calling routine.

4.22 opendir

File:

opendir.c

Processing:

This routine will attempt to open a directory for read access. It will setup a DIR control structure that will be used to access directory information. This routine is layered on the generic open() routine and filesystem specific directory processing routines.

Development Comments:

The BSD group provided this routine.

4.23 pathconf

File:

pathconf.c

Processing:

This routine will obtain the value of one of the path configuration parameters and return it to the calling routine. It is layered on the generic open() and fpathconf() functions. These interfaces should not change with the addition of new filesystem types.

Development Comments:

This routine will try to open the file indicated by path.

If successful, the file descriptor will be used to access the pathconf value specified by `name` using the fpathconf() function.

The file that was accessed is then closed.

4.24 read

File:

deviceio.c

Processing:

This routine is layered on a set of RTEMS calls and filesystem specific read operations. The functions are layered in such a way as to isolate them from change as new filesystems are introduced.

Development Comments:

This routine will examine the type of file descriptor it is sent.

If the file descriptor is associated with a network device, the read function will be mapped to a special network handler. The return code from the network handler will then be sent as the return code from generic read() function.

For file descriptors that are associated with the filesystem the following sequence will be performed:

1. Obtain the file control block associated with the file descriptor
2. Range check the file descriptor
3. Determine that the buffer pointer is not invalid
4. Check that the count is not zero
5. Check the file control block to see if we have permissions to read
6. If there is a read function in the handler table, invoke the handler table read() function
7. Use the return code from the handler table read function(number of bytes read) to increment the offset element of the file control block
8. Return the number of bytes read to the calling program

4.25 readdir

File:

readdir.c

Processing:

This routine was acquired from the BSD group. It has not been altered from its original form.

Development Comments:

The routine calls a customized getdents() function that is provided by the user. This routine provides the filesystem specific aspects of reading a directory.

It is layered on the read() function in the directory handler table. This function has been mapped to the Imfs_dir_read() function.

4.26 unmount

File:

unmount.c

Processing:

This routine will attempt to dismount a mounted filesystem and then free all resources that were allocated for the management of that filesystem.

Development Comments:

- This routine will determine if there are any filesystems currently mounted under the filesystem that we are trying to dismount. This would prevent the dismount of the filesystem.
- It will test to see if the current directory is in the filesystem that we are attempting to dismount. This would prevent the dismount of the filesystem.
- It will scan all the currently open file descriptors to determine is there is an open file descriptor to a file in the filesystem that we are attempting to unmount().

If the above preconditions are met then the following sequence is performed:

1. Call the filesystem specific unmount() function for the filesystem that contains the mount point. This routine should indicate that the mount point no longer has a filesystem mounted below it.
2. Call the filesystem specific fsunmount_me() function for the mounted filesystem that we are trying to unmount(). This routine should clean up any resources that are no longer needed for the management of the file system being un-mounted.
3. Extract the mount table entry for the filesystem that was just dismounted from the mount table chain.
4. Free the memory associated with the extracted mount table entry.

4.27 eval

File:

XXX

Processing:

XXX

Development Comments:

XXX

4.28 getdentsc

File:

XXX

Processing:

XXX

Development Comments:

XXX

5 Filesystem Implementation Requirements

This chapter details the behavioral requirements that all filesystem implementations must adhere to.

5.1 General

The RTEMS filesystem framework was intended to be compliant with the POSIX Files and Directories interface standard. The following filesystem characteristics resulted in a functional switching layer.

```
Figure of the Filesystem Functional Layering goes here.
This figure includes networking and disk caching layering.
```

1. Application programs are presented with a standard set of POSIX compliant functions that allow them to interface with the files, devices and directories in the filesystem. The interfaces to these routines do not reflect the type of subordinate filesystem implementation in which the file will be found.
2. The filesystem framework developed under RTEMS allows for mounting filesystem of different types under the base filesystem.
3. The mechanics of locating file information may be quite different between filesystem types.
4. The process of locating a file may require crossing filesystem boundaries.
5. The transitions between filesystem and the processing required to access information in different filesystem is not visible at the level of the POSIX function call.
6. The POSIX interface standard provides file access by character pathname to the file in some functions and through an integer file descriptor in other functions.
7. The nature of the integer file descriptor and its associated processing is operating system and filesystem specific.
8. Directory and device information must be processed with some of the same routines that apply to files.
9. The form and content of directory and device information differs greatly from that of a regular file.
10. Files, directories and devices represent elements (nodes) of a tree hierarchy.
11. The rules for processing each of the node types that exist under the filesystem are node specific but are still not reflected in the POSIX interface routines.

```
Figure of the Filesystem Functional Layering goes here.
This figure focuses on the Base Filesystem and IMFS.
```

```
Figure of the IMFS Memfile control blocks
```

5.2 File and Directory Removal Constraints

The following POSIX constraints must be honored by all filesystems.

- If a node is a directory with children it cannot be removed.

- The root node of any filesystem, whether the base filesystem or a mounted filesystem, cannot be removed.
- A node that is a directory that is acting as the mount point of a file system cannot be removed.
- On filesystems supporting hard links, a link count is maintained. Prior to node removal, the node's link count is decremented by one. The link count must be less than one to allow for removal of the node.

5.3 API Layering

5.3.1 Mapping of Generic System Calls to Filesystem Specific Functions

The list of generic system calls includes the routines open(), read(), write(), close(), etc..

The Files and Directories section of the POSIX Application Programs Interface specifies a set of functions with calling arguments that are used to gain access to the information in a filesystem. To the application program, these functions allow access to information in any mounted filesystem without explicit knowledge of the filesystem type or the filesystem mount configuration. The following are functions that are provided to the application:

1. access()
2. chdir()
3. chmod()
4. chown()
5. close()
6. closedir()
7. fchmod()
8. fcntl()
9. fdatasync()
10. fpathconf()
11. fstat()
12. fsync()
13. ftruncate()
14. link()
15. lseek()
16. mkdir()
17. mknod()
18. mount()
19. open()
20. opendir()
21. pathconf()
22. read()

23. readdir()
24. rewinddir()
25. rmdir()
26. rmnod()
27. scandir()
28. seekdir()
29. stat()
30. telldir()
31. umask()
32. unlink()
33. unmount()
34. utime()
35. write()

The filesystem's type as well as the node type within the filesystem determine the nature of the processing that must be performed for each of the functions above. The RTEMS filesystem provides a framework that allows new filesystem to be developed and integrated without alteration to the basic framework.

To provide the functional switching that is required, each of the POSIX file and directory functions have been implemented as a shell function. The shell function adheres to the POSIX interface standard. Within this functional shell, filesystem and node type information is accessed which is then used to invoke the appropriate filesystem and node type specific routine to process the POSIX function call.

5.3.2 File/Device/Directory function access via file control block - rtems_libio_t structure

The POSIX open() function returns an integer file descriptor that is used as a reference to file control block information for a specific file. The file control block contains information that is used to locate node, file system, mount table and functional handler information. The diagram in Figure 8 depicts the relationship between and among the following components.

1. File Descriptor Table

   This is an internal RTEMS structure that tracks all currently defined file descriptors in the system. The index that is returned by the file open() operation references a slot in this table. The slot contains a pointer to the file descriptor table entry for this file. The rtems_libio_t structure represents the file control block.

2. Allocation of entry in the File Descriptor Table

   Access to the file descriptor table is controlled through a semaphore that is implemented using the rtems_libio_allocate() function. This routine will grab a semaphore and then scan the file control blocks to determine which slot is free for use. The first free slot is marked as used and the index to this slot is returned as the file descriptor for the open() request. After the alterations have been made to the file control block table, the semaphore is released to allow further operations on the table.

3. Maximum number of entries in the file descriptor table is configurable through the src/exec/sapi/headers/confdefs.h file. If the CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS constant is defined its value will represent the maximum number of file descriptors that are allowed. If CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS is not specified a default value of 20 will be used as the maximum number of file descriptors allowed.
4. File control block - rtems_libio_t structure

   ```
   struct rtems_libio_tt {
     rtems_driver_name_t              *driver;
     off_t                             size;
     off_t                             offset;
     unsigned32                        flags;
     rtems_filesystem_location_info_t  pathinfo;
     Objects_Id                        sem;
     unsigned32                        data0;
     void                              data1;
     void                              file_info;
     rtems_filesystem_file_handlers_r  handlers;
   };
   ```

   A file control block can exist for regular files, devices and directories. The following fields are important for regular file and directory access:

   - Size - For a file this represents the number of bytes currently stored in a file. For a directory this field is not filled in.
   - Offset - For a file this is the byte file position index relative to the start of the file. For a directory this is the byte offset into a sequence of dirent structures.
   - Pathinfo - This is a structure that provides a pointer to node information, OPS table functions, Handler functions and the mount table entry associated with this node.
   - file_info - A pointer to node information that is used by Handler functions
   - handlers - A pointer to a table of handler functions that operate on a file, device or directory through a file descriptor index

5.3.3 File/Directory function access via rtems_filesystem_location_info_t structure

The rtems_filesystem_location_info_tt structure below provides sufficient information to process nodes under a mounted filesystem.

```
struct rtems_filesystem_location_info_tt {
    void                                    *node_access;
    rtems_filesystem_file_handlers_r        *handlers;
    rtems_filesystem_operations_table       *ops;
    rtems_filesystem_mount_table_entry_t    *mt_entry;
};
```

It contains a void pointer to filesystem specific nodal structure, pointers to the OPS table for the filesystem that contains the node, the node type specific handlers for the node and a reference pointer to the mount table entry associated with the filesystem containing the node

5.4 Operation Tables

Filesystem specific operations are invoked indirectly. The set of routines that implement the filesystem are configured into two tables. The Filesystem Handler Table has routines that are specific to a filesystem but remain constant regardless of the actual file type. The File Handler Table has routines that are both filesystem and file type specific.

5.4.1 Filesystem Handler Table Functions

OPS table functions are defined in a `rtems_filesystem_operations_table` structure. It defines functions that are specific to a given filesystem. One table exists for each filesystem that is supported in the RTEMS configuration. The structure definition appears below and is followed by general developmental information on each of the functions contained in this function management structure.

```
typedef struct {
  rtems_filesystem_evalpath_t        evalpath;
  rtems_filesystem_evalmake_t        evalformake;
  rtems_filesystem_link_t            link;
  rtems_filesystem_unlink_t          unlink;
  rtems_filesystem_node_type_t       node_type;
  rtems_filesystem_mknod_t           mknod;
  rtems_filesystem_rmnod_t           rmnod;
  rtems_filesystem_chown_t           chown;
  rtems_filesystem_freenode_t        freenod;
  rtems_filesystem_mount_t           mount;
  rtems_filesystem_fsmount_me_t      fsmount_me;
  rtems_filesystem_unmount_t         unmount;
  rtems_filesystem_fsunmount_me_t    fsunmount_me;
  rtems_filesystem_utime_t           utime;
  rtems_filesystem_evaluate_link_t   eval_link;
  rtems_filesystem_symlink_t         symlink;
} rtems_filesystem_operations_table;
```

5.4.1.1 evalpath Handler

Corresponding Structure Element:

evalpath

Arguments:

```
const char                          *pathname,      /* IN     */
int                                  flags,         /* IN     */
rtems_filesystem_location_info_t   *pathloc        /* IN/OUT */
```

Description:

This routine is responsible for evaluating the pathname passed in based upon the flags and the valid `rthems_filesystem_location_info_t`. Additionally, it must make any changes to pathloc necessary to identify the pathname node. This should include calling the evalpath for a mounted filesystem, if the given filesystem supports the mount command.

This routine returns a 0 if the evaluation was successful. Otherwise, it returns a -1 and sets errno to the correct error.

This routine is required and should NOT be set to NULL.

5.4.1.2 evalformake Handler

Corresponding Structure Element:

evalformake

Arguments:

```
const char                          *path,       /* IN */
rtems_filesystem_location_info_t *pathloc,    /* IN/OUT */
const char                         **name       /* OUT    */
```

Description:

This method is given a path to evaluate and a valid start location. It is responsible for finding the parent node for a requested make command, setting pathloc information to identify the parent node, and setting the name pointer to the first character of the name of the new node. Additionally, if the filesystem supports the mount command, this method should call the evalformake routine for the mounted filesystem.

This routine returns a 0 if the evaluation was successful. Otherwise, it returns a -1 and sets errno to the correct error.

This routine is required and should NOT be set to NULL. However, if the filesystem does not support user creation of a new node, it may set errno to ENOSYS and return -1.

5.4.1.3 link Handler

Corresponding Structure Element:

link

Arguments:

```
rtems_filesystem_location_info_t    *to_loc,      /* IN */
rtems_filesystem_location_info_t    *parent_loc,  /* IN */
const char                          *token        /* IN */
```

Description:

This routine is used to create a hard-link.

It will first examine the st_nlink count of the node that we are trying to. If the link count exceeds LINK_MAX an error will be returned.

The name of the link will be normalized to remove extraneous separators from the end of the name.

This routine is not required and may be set to NULL.

5.4.1.4 unlink Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.1.5 node_type Handler

Corresponding Structure Element:

node_type()

Arguments:

```
rtems_filesystem_location_info_t    *pathloc        /* IN */
```

Description:

XXX

5.4.1.6 mknod Handler

Corresponding Structure Element:

mknod()

Arguments:

```
const char                          *token,        /* IN */
mode_t                               mode,         /* IN */
dev_t                                dev,          /* IN */
rtems_filesystem_location_info_t    *pathloc       /* IN/OUT */
```

Description:

XXX

5.4.1.7 rmnod Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.1.8 chown Handler

Corresponding Structure Element:

chown()

Arguments:

```
rtems_filesystem_location_info_t    *pathloc        /* IN */
uid_t                                owner          /* IN */
gid_t                                group          /* IN */
```

Description:

XXX

5.4.1.9 freenod Handler

Corresponding Structure Element:

freenod()

Arguments:

```
rtems_filesystem_location_info_t      *pathloc       /* IN */
```

Description:

This routine is used by the generic code to allow memory to be allocated during the evaluate routines, and set free when the generic code is finished accessing a node. If the evaluate routines allocate memory to identify a node this routine should be utilized to free that memory.

This routine is not required and may be set to NULL.

5.4.1.10 mount Handler

Corresponding Structure Element:

mount()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

Description:

XXX

5.4.1.11 fsmount_me Handler

Corresponding Structure Element:

XXX

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

Description:

This function is provided with a filesystem to take care of the internal filesystem management details associated with mounting that filesystem under the RTEMS environment.

It is not responsible for the mounting details associated the filesystem containing the mount point.

The rtems_filesystem_mount_table_entry_t structure contains the key elements below:

rtems_filesystem_location_info_t *mt_point_node,

This structure contains information about the mount point. This allows us to find the ops-table and the handling functions associated with the filesystem containing the mount point.

rtems_filesystem_location_info_t *fs_root_node,

This structure contains information about the root node in the file system to be mounted. It allows us to find the ops-table and the handling functions associated with the filesystem to be mounted.

rtems_filesystem_options_t options,

Read only or read/write access

void *fs_info,

This points to an allocated block of memory the will be used to hold any filesystem specific information of a global nature. This allocated region if important because it allows us to mount the same filesystem type more than once under the RTEMS system. Each instance of the mounted filesystem has its own set of global management information that is separate from the global management information associated with the other instances of the mounted filesystem type.

rtems_filesystem_limits_and_options_t pathconf_info,

The table contains the following set of values associated with the mounted filesystem:

- link_max
- max_canon
- max_input
- name_max
- path_max
- pipe_buf
- posix_async_io
- posix_chown_restrictions
- posix_no_trunc
- posix_prio_io

- posix_sync_io
- posix_vdisable

These values are accessed with the pathconf() and the fpathconf () functions.

const char *dev

The is intended to contain a string that identifies the device that contains the filesystem information. The filesystems that are currently implemented are memory based and don't require a device specification.

If the mt_point_node.node_access is NULL then we are mounting the base file system.

The routine will create a directory node for the root of the IMFS file system.

The node will have read, write and execute permissions for owner, group and others.

The node's name will be a null string.

A filesystem information structure(fs_info) will be allocated and initialized for the IMFS filesystem. The fs_info pointer in the mount table entry will be set to point the filesystem information structure.

The pathconf_info element of the mount table will be set to the appropriate table of path configuration constants (LIMITS_AND_OPTIONS).

The fs_root_node structure will be filled in with the following:

- pointer to the allocated root node of the filesystem
- directory handlers for a directory node under the IMFS filesystem
- OPS table functions for the IMFS

A 0 will be returned to the calling routine if the process succeeded, otherwise a 1 will be returned.

5.4.1.12 unmount Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.1.13 fsunmount_me Handler

Corresponding Structure Element:

imfs_fsunmount_me()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

Description:

XXX

5.4.1.14 utime Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.1.15 eval_link Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.1.16 symlink Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

5.4.2 File Handler Table Functions

Handler table functions are defined in a `rtems_filesystem_file_handlers_r` structure. It defines functions that are specific to a node type in a given filesystem. One table exists for each of the filesystem's node types. The structure definition appears below. It is followed by general developmental information on each of the functions associated with regular files contained in this function management structure.

```
typedef struct {
  rtems_filesystem_open_t           open;
  rtems_filesystem_close_t          close;
```

```
    rtems_filesystem_read_t          read;
    rtems_filesystem_write_t         write;
    rtems_filesystem_ioctl_t         ioctl;
    rtems_filesystem_lseek_t         lseek;
    rtems_filesystem_fstat_t         fstat;
    rtems_filesystem_fchmod_t        fchmod;
    rtems_filesystem_ftruncate_t     ftruncate;
    rtems_filesystem_fpathconf_t     fpathconf;
    rtems_filesystem_fsync_t         fsync;
    rtems_filesystem_fdatasync_t     fdatasync;
    rtems_filesystem_fcntl_t         fcntl;
  } rtems_filesystem_file_handlers_r;
```

5.4.2.1 open Handler

Corresponding Structure Element:

open()

Arguments:

```
rtems_libio_t   *iop,
const char      *pathname,
unsigned32       flag,
unsigned32       mode
```

Description:

XXX

5.4.2.2 close Handler

Corresponding Structure Element:

close()

Arguments:

```
rtems_libio_t     *iop
```

Description:

XXX

NOTES:

XXX

5.4.2.3 read Handler

Corresponding Structure Element:

read()

Arguments:

```
rtems_libio_t     *iop,
void              *buffer,
unsigned32         count
```

Description:

XXX

NOTES:

XXX

5.4.2.4 write Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

5.4.2.5 ioctl Handler

Corresponding Structure Element:

XXX

Arguments:

```
rtems_libio_t     *iop,
unsigned32       command,
void              *buffer
```

Description:

XXX

NOTES:

XXX

5.4.2.6 lseek Handler

Corresponding Structure Element:

lseek()

Arguments:

```
rtems_libio_t     *iop,
off_t              offset,
int                whence
```

Description:

XXX

NOTES:

XXX

5.4.2.7 fstat Handler

Corresponding Structure Element:

fstat()

Arguments:

```
rtems_filesystem_location_info_t   *loc,
struct stat                        *buf
```

Description:

The following information is extracted from the filesystem specific node and placed in the `stat` structure:

- st_mode
- st_nlink
- st_ino
- st_uid
- st_gid
- st_atime
- st_mtime
- st_ctime

NOTES:

Both the `stat()` and `lstat()` services are implemented directly using the `fstat()` handler. The difference in behavior is determined by how the path is evaluated prior to this handler being called on a particular file entity.

The `fstat()` system call is implemented directly on top of this filesystem handler.

5.4.2.8 fchmod Handler

Corresponding Structure Element:

fchmod()

Arguments:

```
rtems_libio_t     *iop
mode_t              mode
```

Description:

XXX

NOTES:

XXX

5.4.2.9 ftruncate Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

5.4.2.10 fpathconf Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

5.4.2.11 fsync Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

5.4.2.12 fdatasync Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

5.4.2.13 fcntl Handler

Corresponding Structure Element:

XXX

Arguments:

XXX

Description:

XXX

NOTES:

XXX

6 In-Memory Filesystem

This chapter describes the In-Memory FileSystem (IMFS). The IMFS is a full featured POSIX filesystem that keeps all information in memory.

6.1 IMFS Per Node Data Structure

Each regular file, device, hard link, and directory is represented by a data structure called a `jnode`. The `jnode` is formally represented by the structure:

```
struct IMFS_jnode_tt {
  Chain_Node          Node;             /* for chaining them together */
  IMFS_jnode_t       *Parent;           /* Parent node */
  char                name[NAME_MAX+1]; /* "basename" */
  mode_t              st_mode;          /* File mode */
  nlink_t             st_nlink;         /* Link count */
  ino_t               st_ino;           /* inode */

  uid_t               st_uid;           /* User ID of owner */
  gid_t               st_gid;           /* Group ID of owner */
  time_t              st_atime;         /* Time of last access */
  time_t              st_mtime;         /* Time of last modification */
  time_t              st_ctime;         /* Time of last status change */
  IMFS_jnode_types_t  type;             /* Type of this entry */
  IMFS_typs_union     info;
};
```

The key elements of this structure are listed below together with a brief explanation of their role in the filesystem.

Node exists to allow the entire `jnode` structure to be included in a chain.

Parent is a pointer to another `jnode` structure that is the logical parent of the node in which it appears. This field may be NULL if the file associated with this node is deleted but there are open file descriptors on this file or there are still hard links to this node.

name is the name of this node within the filesystem hierarchical tree. Example: If the fully qualified pathname to the `jnode` was `/a/b/c`, the `jnode` name field would contain the null terminated string `"c"`.

st_mode is the standard Unix access permissions for the file or directory.

st_nlink is the number of hard links to this file. When a `jnode` is first created its link count is set to 1. A `jnode` and its associated resources cannot be deleted unless its link count is less than 1.

st_ino is a unique node identification number

st_uid is the user ID of the file's owner

st_gid is the group ID of the file's owner

st_atime is the time of the last access to this file

st_mtime is the time of the last modification of this file

st_ctime is the time of the last status change to the file

type is the indication of node type must be one of the following states:

- IMFS_DIRECTORY
- IMFS_MEMORY_FILE
- IMFS_HARD_LINK
- IMFS_SYM_LINK
- IMFS_DEVICE

info is this contains a structure that is unique to file type (See IMFS_typs_union in imfs.h).

- IMFS_DIRECTORY

  An IMFS directory contains a dynamic chain structure that records all files and directories that are subordinate to the directory node.

- IMFS_MEMORY_FILE

  Under the in memory filesystem regular files hold data. Data is dynamically allocated to the file in 128 byte chunks of memory. The individual chunks of memory are tracked by arrays of pointers that record the address of the allocated chunk of memory. Single, double, and triple indirection pointers are used to record the locations of all segments of the file. The memory organization of an IMFS file are discussed elsewhere in this manual.

- IMFS_HARD_LINK

  The IMFS filesystem supports the concept of hard links to other nodes in the IMFS filesystem. These hard links are actual pointers to other nodes in the same filesystem. This type of link cannot cross-filesystem boundaries.

- IMFS_SYM_LINK

  The IMFS filesystem supports the concept of symbolic links to other nodes in any filesystem. A symbolic link consists of a pointer to a character string that represents the pathname to the target node. This type of link can cross-filesystem boundaries. Just as with most versions of UNIX supporting symbolic links, a symbolic link can point to a non-existent file.

- IMFS_DEVICE

  All RTEMS devices now appear as files under the in memory filesystem. On system initialization, all devices are registered as nodes under the file system.

6.2 Miscellaneous IMFS Information

6.3 Memory associated with the IMFS

A memory based filesystem draws its resources for files and directories from the memory resources of the system. When it is time to un-mount the filesystem, the memory resources that supported filesystem are set free. In order to free these resources, a recursive walk of the filesystems tree structure will be performed. As the leaf nodes under the filesystem are encountered their resources are freed. When directories are made empty by this process, their resources are freed.

6.3.1 Node removal constraints for the IMFS

The IMFS conforms to the general filesystem requirements for node removal. See Section 5.2 [File and Directory Removal Constraints], page 27.

6.3.2 IMFS General Housekeeping Notes

The following is a list of odd housekeeping notes for the IMFS.

- If the global variable rtems_filesystem_current refers to the node that we are trying to remove, the node_access element of this structure must be set to NULL to invalidate it.
- If the node was of IMFS_MEMORY_FILE type, free the memory associated with the memory file before freeing the node. Use the IMFS_memfile_remove() function.

6.4 IMFS Operation Tables

6.4.1 IMFS Filesystem Handler Table Functions

OPS table functions are defined in a rtems_filesystem_operations_table structure. It defines functions that are specific to a given filesystem. One table exists for each filesystem that is supported in the RTEMS configuration. The structure definition appears below and is followed by general developmental information on each of the functions contained in this function management structure.

```
rtems_filesystem_operations_table  IMFS_ops = {
  IMFS_eval_path,
  IMFS_evaluate_for_make,
  IMFS_link,
  IMFS_unlink,
  IMFS_node_type,
  IMFS_mknod,
  IMFS_rmnod,
  IMFS_chown,
  IMFS_freenodinfo,
  IMFS_mount,
  IMFS_initialize,
  IMFS_unmount,
  IMFS_fsunmount,
  IMFS_utime,
```

```
  IMFS_evaluate_link,
  IMFS_symlink,
  IMFS_readlink
};
```

6.4.1.1 IMFS_evalpath()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.1.2 IMFS_evalformake()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.1.3 IMFS_link()

Corresponding Structure Element:

link

Arguments:

```
rtems_filesystem_location_info_t    *to_loc,      /* IN */
rtems_filesystem_location_info_t    *parent_loc,  /* IN */
const char                          *token        /* IN */
```

File:

imfs_link.c

Description:

This routine is used in the IMFS filesystem to create a hard-link.

It will first examine the st_nlink count of the node that we are trying to. If the link count exceeds LINK_MAX an error will be returned.

The name of the link will be normalized to remove extraneous separators from the end of the name.

IMFS_create_node will be used to create a filesystem node that will have the following characteristics:

- parent that was determined in the link() function in file link.c
- Type will be set to IMFS_HARD_LINK
- name will be set to the normalized name
- mode of the hard-link will be set to the mode of the target node

If there was trouble allocating memory for the new node an error will be returned.

The st_nlink count of the target node will be incremented to reflect the new link.

The time fields of the link will be set to reflect the creation time of the hard-link.

6.4.1.4 IMFS_unlink()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.1.5 IMFS_node_type()

Corresponding Structure Element:

IMFS_node_type()

Arguments:

```
rtems_filesystem_location_info_t    *pathloc        /* IN */
```

File:

imfs_ntype.c

Description:

This routine will locate the IMFS_jnode_t structure that holds ownership information for the selected node in the filesystem.

This structure is pointed to by pathloc->node_access.

The IMFS_jnode_t type element indicates one of the node types listed below:

- RTEMS_FILESYSTEM_DIRECTORY
- RTEMS_FILESYSTEM_DEVICE
- RTEMS_FILESYSTEM_HARD_LINK
- RTEMS_FILESYSTEM_MEMORY_FILE

6.4.1.6 IMFS_mknod()

Corresponding Structure Element:

IMFS_mknod()

Arguments:

```
const char                          *token,        /* IN */
mode_t                               mode,         /* IN */
dev_t                                dev,          /* IN */
rtems_filesystem_location_info_t    *pathloc       /* IN/OUT */
```

File:

imfs_mknod.c

Description:

This routine will examine the mode argument to determine is we are trying to create a directory, regular file and a device node. The creation of other node types is not permitted and will cause an assert.

Memory space will be allocated for a `jnode` and the node will be set up according to the nodal type that was specified. The IMFS_create_node() function performs the allocation and setup of the node.

The only problem that is currently reported is the lack of memory when we attempt to allocate space for the `jnode` (ENOMEN).

6.4.1.7 IMFS_rmnod()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.1.8 IMFS_chown()

Corresponding Structure Element:

IMFS_chown()

Arguments:

```
rtems_filesystem_location_info_t    *pathloc        /* IN */
uid_t                                owner          /* IN */
gid_t                                group          /* IN */
```

File:

imfs_chown.c

Description:

This routine will locate the IMFS_jnode_t structure that holds ownership information for the selected node in the filesystem.

This structure is pointed to by pathloc->node_access.

The st_uid and st_gid fields of the node are then modified. Since this is a memory based filesystem, no further action is required to alter the ownership of the IMFS_jnode_t structure.

6.4.1.9 IMFS_freenod()

Corresponding Structure Element:

IMFS_freenod()

Arguments:

```
rtems_filesystem_location_info_t      *pathloc       /* IN */
```

File:

imfs_free.c

Description:

This method is a private function to the IMFS. It is called by IMFS routines to free nodes that have been allocated. Examples of where this routine may be called from are unlink and rmnod.

Note: This routine should not be confused with the filesystem callback freenod. The IMFS allocates memory until the node no longer exists.

6.4.1.10 IMFS_freenodinfo()

Corresponding Structure Element:

IMFS_freenodinfo()

Arguments:

```
rtems_filesystem_location_info_t      *pathloc       /* IN */
```

File:

imfs_free.c

Description:

The In-Memory File System does not need to allocate memory during the evaluate routines. Therefore, this routine simply routines PASS.

6.4.1.11 IMFS_mount()

Corresponding Structure Element:

IMFS_mount()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

File:

imfs_mount.c

Description:

This routine provides the filesystem specific processing required to mount a filesystem for the system that contains the mount point. It will determine if the point that we are trying to mount onto is a node of IMFS_DIRECTORY type.

If it is the node's info element is altered so that the info.directory.mt_fs element points to the mount table chain entry that is associated with the mounted filesystem at this point. The info.directory.mt_fs element can be examined to determine if a filesystem is mounted at a directory. If it is NULL, the directory does not serve as a mount point. A non-NULL entry indicates that the directory does serve as a mount point and the value of info.directory.mt_fs can be used to locate the mount table chain entry that describes the filesystem mounted at this point.

6.4.1.12 IMFS_fsmount_me()

Corresponding Structure Element:

IMFS_initialize()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

File:

imfs_init.c

Description:

This function is provided with a filesystem to take care of the internal filesystem management details associated with mounting that filesystem under the RTEMS environment.

It is not responsible for the mounting details associated the filesystem containing the mount point.

The rtems_filesystem_mount_table_entry_t structure contains the key elements below:

rtems_filesystem_location_info_t *mt_point_node,

This structure contains information about the mount point. This allows us to find the ops-table and the handling functions associated with the filesystem containing the mount point.

rtems_filesystem_location_info_t *fs_root_node,

This structure contains information about the root node in the file system to be mounted. It allows us to find the ops-table and the handling functions associated with the filesystem to be mounted.

rtems_filesystem_options_t options,

Read only or read/write access

void *fs_info,

This points to an allocated block of memory the will be used to hold any filesystem specific information of a global nature. This allocated region if important because it allows us to mount the same filesystem type more than once under the RTEMS system. Each instance of the mounted filesystem has its own set of global management information that is separate from the global management information associated with the other instances of the mounted filesystem type.

rtems_filesystem_limits_and_options_t pathconf_info,

The table contains the following set of values associated with the mounted filesystem:

- link_max
- max_canon
- max_input
- name_max
- path_max
- pipe_buf
- posix_async_io
- posix_chown_restrictions
- posix_no_trunc
- posix_prio_io
- posix_sync_io
- posix_vdisable

These values are accessed with the pathconf() and the fpathconf () functions.

const char *dev

The is intended to contain a string that identifies the device that contains the filesystem information. The filesystems that are currently implemented are memory based and don't require a device specification.

If the mt_point_node.node_access is NULL then we are mounting the base file system.

The routine will create a directory node for the root of the IMFS file system.

The node will have read, write and execute permissions for owner, group and others.

The node's name will be a null string.

A filesystem information structure(fs_info) will be allocated and initialized for the IMFS filesystem. The fs_info pointer in the mount table entry will be set to point the filesystem information structure.

The pathconf_info element of the mount table will be set to the appropriate table of path configuration constants (IMFS_LIMITS_AND_OPTIONS).

The fs_root_node structure will be filled in with the following:

- pointer to the allocated root node of the filesystem
- directory handlers for a directory node under the IMFS filesystem
- OPS table functions for the IMFS

A 0 will be returned to the calling routine if the process succeeded, otherwise a 1 will be returned.

6.4.1.13 IMFS_unmount()

Corresponding Structure Element:

IMFS_unmount()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

File:

imfs_unmount.c

Description:

This routine allows the IMFS to unmount a filesystem that has been mounted onto a IMFS directory.

The mount entry mount point node access is verified to be a mounted directory. It's mt_fs is set to NULL. This identifies to future calles into the IMFS that this directory node is no longer a mount point. Additionally, it will allow any directories that were hidden by the mounted system to again become visible.

6.4.1.14 IMFS_fsunmount()

Corresponding Structure Element:

imfs_fsunmount()

Arguments:

```
rtems_filesystem_mount_table_entry_t   *mt_entry
```

File:

imfs_init.c

Description:

This method unmounts this instance of the IMFS file system. It is the counterpart to the IMFS_initialize routine. It is called by the generic code under the fsunmount_me callback.

All method loops finding the first encountered node with no children and removing the node from the tree, thus returning allocated resources. This is done until all allocated nodes are returned.

6.4.1.15 IMFS_utime()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.1.16 IMFS_eval_link()

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.2 Regular File Handler Table Functions

Handler table functions are defined in a rtems_filesystem_file_handlers_r structure. It defines functions that are specific to a node type in a given filesystem. One table exists for each of the filesystem's node types. The structure definition appears below. It is followed by general developmental information on each of the functions associated with regular files contained in this function management structure.

```
rtems_filesystem_file_handlers_r IMFS_memfile_handlers = {
  memfile_open,
  memfile_close,
  memfile_read,
  memfile_write,
  memfile_ioctl,
  memfile_lseek,
  IMFS_stat,
  IMFS_fchmod,
  memfile_ftruncate,
  NULL,                /* fpathconf */
  NULL,                /* fsync */
  IMFS_fdatasync,
  IMFS_fcntl
};
```

6.4.2.1 memfile_open() for Regular Files

Corresponding Structure Element:

memfile_open()

Arguments:

```
rtems_libio_t   *iop,
const char      *pathname,
unsigned32       flag,
unsigned32       mode
```

File:

memfile.c

Description:

Currently this function is a shell. No meaningful processing is performed and a success code is always returned.

6.4.2.2 memfile_close() for Regular Files

Corresponding Structure Element:

memfile_close()

Arguments:

```
rtems_libio_t     *iop
```

File:

memfile.c

Description:

This routine is a dummy for regular files under the base filesystem. It performs a capture of the IMFS_jnode_t pointer from the file control block and then immediately returns a success status.

6.4.2.3 memfile_read() for Regular Files

Corresponding Structure Element:

memfile_read()

Arguments:

```
rtems_libio_t     *iop,
void              *buffer,
unsigned32         count
```

File:

memfile.c

Description:

This routine will determine the `jnode` that is associated with this file.

It will then call IMFS_memfile_read() with the `jnode`, file position index, buffer and transfer count as arguments.

IMFS_memfile_read() will do the following:

- Verify that the `jnode` is associated with a memory file
- Verify that the destination of the read is valid
- Adjust the length of the read if it is too long
- Acquire data from the memory blocks associated with the file
- Update the access time for the data in the file

6.4.2.4 memfile_write() for Regular Files

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.2.5 memfile_ioctl() for Regular Files

Corresponding Structure Element:

XXX

Arguments:

```
rtems_libio_t     *iop,
unsigned32       command,
void              *buffer
```

File:

memfile.c

Description:

The current code is a placeholder for future development. The routine returns a successful completion status.

6.4.2.6 memfile_lseek() for Regular Files

Corresponding Structure Element:

Memfile_lseek()

Arguments:

```
rtems_libio_t     *iop,
off_t              offset,
int                whence
```

File:

memfile.c

Description:

This routine make sure that the memory based file is sufficiently large to allow for the new file position index.

The IMFS_memfile_extend() function is used to evaluate the current size of the memory file and allocate additional memory blocks if required by the new file position index. A success code is always returned from this routine.

6.4.2.7 IMFS_stat() for Regular Files

Corresponding Structure Element:

IMFS_stat()

Arguments:

```
rtems_filesystem_location_info_t   *loc,
struct stat                        *buf
```

File:

imfs_stat.c

Description:

This routine actually performs status processing for both devices and regular files.

The IMFS_jnode_t structure is referenced to determine the type of node under the filesystem.

If the node is associated with a device, node information is extracted and transformed to set the st_dev element of the stat structure.

If the node is a regular file, the size of the regular file is extracted from the node.

This routine rejects other node types.

The following information is extracted from the node and placed in the stat structure:

- st_mode
- st_nlink
- st_ino
- st_uid
- st_gid
- st_atime
- st_mtime
- st_ctime

6.4.2.8 IMFS_fchmod() for Regular Files

Corresponding Structure Element:

IMFS_fchmod()

Arguments:

```
rtems_libio_t     *iop
mode_t              mode
```

File:

imfs_fchmod.c

Description:

This routine will obtain the pointer to the IMFS_jnode_t structure from the information currently in the file control block.

Based on configuration the routine will acquire the user ID from a call to getuid() or from the IMFS_jnode_t structure.

It then checks to see if we have the ownership rights to alter the mode of the file. If the caller does not, an error code is returned.

An additional test is performed to verify that the caller is not trying to alter the nature of the node. If the caller is attempting to alter more than the permissions associated with user group and other, an error is returned.

If all the preconditions are met, the user, group and other fields are set based on the mode calling parameter.

6.4.2.9 memfile_ftruncate() for Regular Files

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.2.10 No pathconf() for Regular Files

Corresponding Structure Element:

NULL

Arguments:

Not Implemented

File:

Not Implemented

Description:

Not Implemented

6.4.2.11 No fsync() for Regular Files

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.2.12 IMFS_fdatasync() for Regular Files

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.3 Directory Handler Table Functions

Handler table functions are defined in a rtems_filesystem_file_handlers_r structure. It defines functions that are specific to a node type in a given filesystem. One table exists for each of the filesystem's node types. The structure definition appears below. It is followed by general developmental information on each of the functions associated with directories contained in this function management structure.

```
rtems_filesystem_file_handlers_r IMFS_directory_handlers = {
  IMFS_dir_open,
  IMFS_dir_close,
  IMFS_dir_read,
  NULL,             /* write */
  NULL,             /* ioctl */
  IMFS_dir_lseek,
  IMFS_dir_fstat,
  IMFS_fchmod,
  NULL,             /* ftruncate */
  NULL,             /* fpathconf */
  NULL,             /* fsync */
  IMFS_fdatasync,
  IMFS_fcntl
};
```

6.4.3.1 IMFS_dir_open() for Directories

Corresponding Structure Element:

imfs_dir_open()

Arguments:

```
rtems_libio_t  *iop,
const char     *pathname,
unsigned32      flag,
unsigned32      mode
```

File:

imfs_directory.c

Description:

This routine will look into the file control block to find the `jnode` that is associated with the directory.

The routine will verify that the node is a directory. If its not a directory an error code will be returned.

If it is a directory, the offset in the file control block will be set to 0. This allows us to start reading at the beginning of the directory.

6.4.3.2 IMFS_dir_close() for Directories

Corresponding Structure Element:

imfs_dir_close()

Arguments:

```
rtems_libio_t     *iop
```

File:

imfs_directory.c

Description:

This routine is a dummy for directories under the base filesystem. It immediately returns a success status.

6.4.3.3 IMFS_dir_read() for Directories

Corresponding Structure Element:

imfs_dir_read

Arguments:

```
rtems_libio_t  *iop,
void           *buffer,
unsigned32      count
```

File:

imfs_directory.c

Description:

This routine will read a fixed number of directory entries from the current directory offset. The number of directory bytes read will be returned from this routine.

6.4.3.4 No write() for Directories

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.3.5 No ioctl() for Directories

Corresponding Structure Element:

ioctl

Arguments:

File:

Not supported

Description:

XXX

6.4.3.6 IMFS_dir_lseek() for Directories

Corresponding Structure Element:

imfs_dir_lseek()

Arguments:

```
rtems_libio_t      *iop,
off_t               offset,
int                 whence
```

File:

imfs_directory.c

Description:

This routine alters the offset in the file control block.

No test is performed on the number of children under the current open directory. The imfs_dir_read() function protects against reads beyond the current size to the directory by returning a 0 bytes transfered to the calling programs whenever the file position index exceeds the last entry in the open directory.

6.4.3.7 IMFS_dir_fstat() for Directories

Corresponding Structure Element:

imfs_dir_fstat()

Arguments:

```
rtems_filesystem_location_info_t   *loc,
struct stat                        *buf
```

File:

imfs_directory.c

Description:

The node access information in the rtems_filesystem_location_info_t structure is used to locate the appropriate IMFS_jnode_t structure. The following information is taken from the IMFS_jnode_t structure and placed in the stat structure:

- st_ino
- st_mode
- st_nlink
- st_uid
- st_gid
- st_atime
- st_mtime
- st_ctime

The st_size field is obtained by running through the chain of directory entries and summing the sizes of the dirent structures associated with each of the children of the directory.

6.4.3.8 IMFS_fchmod() for Directories

Corresponding Structure Element:

IMFS_fchmod()

Arguments:

```
rtems_libio_t     *iop
mode_t             mode
```

File:

imfs_fchmod.c

Description:

This routine will obtain the pointer to the IMFS_jnode_t structure from the information currently in the file control block.

Based on configuration the routine will acquire the user ID from a call to getuid() or from the IMFS_jnode_t structure.

It then checks to see if we have the ownership rights to alter the mode of the file. If the caller does not, an error code is returned.

An additional test is performed to verify that the caller is not trying to alter the nature of the node. If the caller is attempting to alter more than the permissions associated with user group and other, an error is returned.

If all the preconditions are met, the user, group and other fields are set based on the mode calling parameter.

6.4.3.9 No ftruncate() for Directories

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.3.10 No fpathconf() for Directories

Corresponding Structure Element:

fpathconf

Arguments:

Not Implemented

File:

Not Implemented

Description:

Not Implemented

6.4.3.11 No fsync() for Directories

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.3.12 IMFS_fdatasync() for Directories

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.4 Device Handler Table Functions

Handler table functions are defined in a rtems_filesystem_file_handlers_r structure. It defines functions that are specific to a node type in a given filesystem. One table exists for each of the filesystem's node types. The structure definition appears below. It is followed by general developmental information on each of the functions associated with devices contained in this function management structure.

```
typedef struct {
  rtems_filesystem_open_t           open;
  rtems_filesystem_close_t          close;
  rtems_filesystem_read_t           read;
  rtems_filesystem_write_t          write;
  rtems_filesystem_ioctl_t          ioctl;
  rtems_filesystem_lseek_t          lseek;
  rtems_filesystem_fstat_t          fstat;
  rtems_filesystem_fchmod_t         fchmod;
  rtems_filesystem_ftruncate_t      ftruncate;
  rtems_filesystem_fpathconf_t      fpathconf;
```

```
    rtems_filesystem_fsync_t          fsync;
    rtems_filesystem_fdatasync_t      fdatasync;
  } rtems_filesystem_file_handlers_r;
```

6.4.4.1 device_open() for Devices

Corresponding Structure Element:

device_open()

Arguments:

```
rtems_libio_t     *iop,
const char        *pathname,
unsigned32         flag,
unsigned32         mode
```

File:

deviceio.c

Description:

This routine will use the file control block to locate the node structure for the device.

It will extract the major and minor device numbers from the `jnode`.

The major and minor device numbers will be used to make a rtems_io_open() function call to open the device driver. An argument list is sent to the driver that contains the file control block, flags and mode information.

6.4.4.2 device_close() for Devices

Corresponding Structure Element:

device_close()

Arguments:

```
rtems_libio_t     *iop
```

File:

deviceio.c

Description:

This routine extracts the major and minor device driver numbers from the IMFS_jnode_t that is referenced in the file control block.

It also forms an argument list that contains the file control block.

A rtems_io_close() function call is made to close the device specified by the major and minor device numbers.

6.4.4.3 device_read() for Devices

Corresponding Structure Element:

device_read()

Arguments:

```
rtems_libio_t     *iop,
void              *buffer,
unsigned32         count
```

File:

deviceio.c

Description:

This routine will extract the major and minor numbers for the device from the - jnode- associated with the file descriptor.

A rtems_io_read() call will be made to the device driver associated with the file descriptor. The major and minor device number will be sent as arguments as well as an argument list consisting of:

- file control block
- file position index
- buffer pointer where the data read is to be placed
- count indicating the number of bytes that the program wishes to read from the device
- flags from the file control block

On return from the rtems_io_read() the number of bytes that were actually read will be returned to the calling program.

6.4.4.4 device_write() for Devices

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.4.5 device_ioctl() for Devices

Corresponding Structure Element:

ioctl

Arguments:

```
rtems_libio_t     *iop,
unsigned32         command,
void              *buffer
```

File:

deviceio.c

Description:

This handler will obtain status information about a device.

The form of status is device dependent.

The rtems_io_control() function uses the major and minor number of the device to obtain the status information.

rtems_io_control() requires an rtems_libio_ioctl_args_t argument list which contains the file control block, device specific command and a buffer pointer to return the device status information.

The device specific command should indicate the nature of the information that is desired from the device.

After the rtems_io_control() is processed, the buffer should contain the requested device information.

If the device information is not obtained properly a -1 will be returned to the calling program, otherwise the ioctl_return value is returned.

6.4.4.6 device_lseek() for Devices

Corresponding Structure Element:

device_lseek()

Arguments:

```
rtems_libio_t     *iop,
off_t              offset,
int                whence
```

File:

deviceio.c

Description:

At the present time this is a placeholder function. It always returns a successful status.

6.4.4.7 IMFS_stat() for Devices

Corresponding Structure Element:

IMFS_stat()

Arguments:

```
rtems_filesystem_location_info_t   *loc,
struct stat                        *buf
```

File:

imfs_stat.c

Description:

This routine actually performs status processing for both devices and regular files.

The IMFS_jnode_t structure is referenced to determine the type of node under the filesystem.

If the node is associated with a device, node information is extracted and transformed to set the st_dev element of the stat structure.

If the node is a regular file, the size of the regular file is extracted from the node.

This routine rejects other node types.

The following information is extracted from the node and placed in the stat structure:

- st_mode
- st_nlink
- st_ino
- st_uid
- st_gid
- st_atime
- st_mtime
- st_ctime

6.4.4.8 IMFS_fchmod() for Devices

Corresponding Structure Element:

IMFS_fchmod()

Arguments:

```
rtems_libio_t     *iop
mode_t             mode
```

File:

imfs_fchmod.c

Description:

This routine will obtain the pointer to the IMFS_jnode_t structure from the information currently in the file control block.

Based on configuration the routine will acquire the user ID from a call to getuid() or from the IMFS_jnode_t structure.

It then checks to see if we have the ownership rights to alter the mode of the file. If the caller does not, an error code is returned.

An additional test is performed to verify that the caller is not trying to alter the nature of the node. If the caller is attempting to alter more than the permissions associated with user group and other, an error is returned.

If all the preconditions are met, the user, group and other fields are set based on the mode calling parameter.

6.4.4.9 No ftruncate() for Devices

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.4.10 No fpathconf() for Devices

Corresponding Structure Element:

fpathconf

Arguments:

Not Implemented

File:

Not Implemented

Description:

Not Implemented

6.4.4.11 No fsync() for Devices

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

6.4.4.12 No fdatasync() for Devices

Not Implemented

Corresponding Structure Element:

XXX

Arguments:

XXX

File:

XXX

Description:

XXX

7 Miniature In-Memory Filesystem

This chapter describes the Miniature In-Memory FileSystem (miniIMFS). The miniIMFS is a reduced feature version of the IMFS designed to provide minimal functionality and have a low memory footprint.

This chapter should be written after the IMFS chapter is completed and describe the implementation of the mini-IMFS.

8 Trivial FTP Client Filesystem

This chapter describes the Trivial FTP (TFTP) Client Filesystem.

This chapter should be written after the IMFS chapter is completed and describe the implementation of the TFTP.

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.