Flash File System Integration into RTEMS Operating System

Alan Mick

This presentation contains no ITAR restricted material

The Johns Hopkins University
APPLIED PHYSICS LABORATORY

APL Flash File System

- **An Extension of Existing Open Source Flash File System**
  - An open source file system designed for NAND Flash Devices
    - Write-Once, Read-Many, Erase by Block
    - Optimum performance in writing or appending files
    - File rewriting is supported, but is inefficient
  - Used in the Android operating system
  - Logging or Journaling File System
- **APLFFS Extensions**
  - Eliminated dynamic memory allocation
  - Enhanced support for large disks by allowing multiple pages per chunk

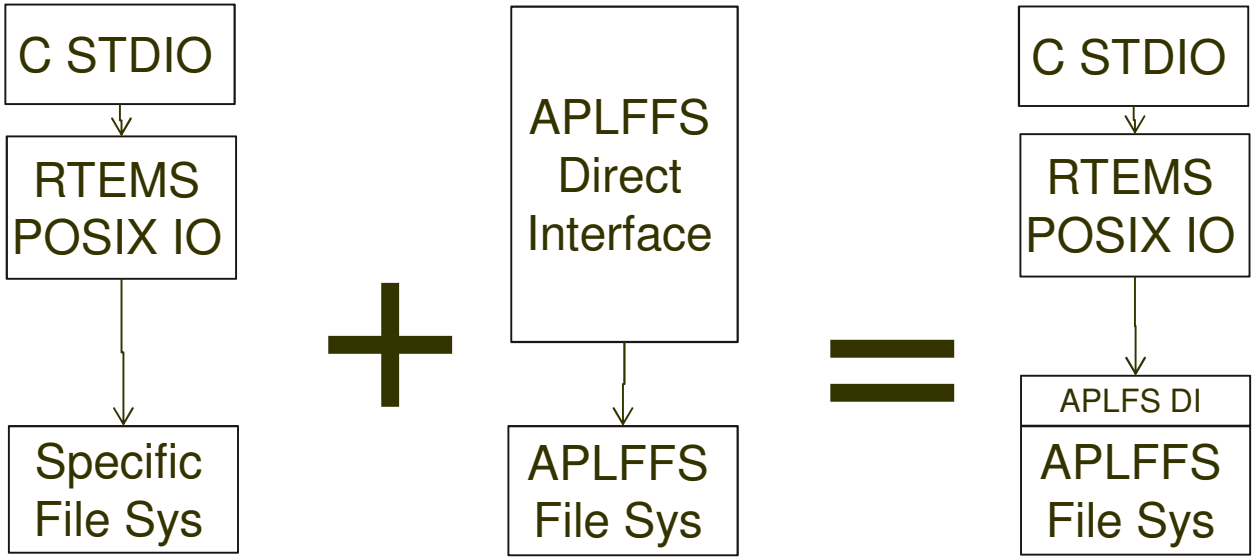
APL Flash File System on RTEMS for Solar Probe

- **Why RTEMS?**
  - The Solar Probe target processor is Aeroflex Gaisler LEON3
  - RTEMS is given priority over other operating systems by the vendor
- **APL FFS is OS-independent**
  - A Direct interface is available for standalone-usage
  - Source is provided for integration into a Linux kernel
  - Calls to the direct interface are inserted into the appropriate file system driver template for the target operating system.

RTEMS

- **RTEMS is an open-source real-time operating system with space heritage.**
- **Primary operating system supported for the LEON3 by Aeroflex Gaisler.**
- **RTEMS 4.10 Gaisler SPARC Leon distribution**
- **RTEMS 4.11 File system support has been updated**
  - Simplification of path evaluations
  - Support for removable media such as USB sticks
  - Details are provided at: http://git.rtems.org/rtems/commit/?id=3b7c123c8d910eb60ab3b38dec6224e2de9847c9

Basic Approach



- **The APLFFS has a "direct interface" which allows it to be used in a "bare C" environment.**
- **This interface is very similar to the POSIX IO interface with some features of the C stdio interface.**
- **The Direct Interface was adapted to conform to the conventions used by RTEMS and duplicate functionality provided by RTEMS was eliminated.**
- **However, we will not focus specifically on the APLFFS; we will focus on the general process of integrating *any* custom file system into RTEMS.**

APL

File System Overview

APL

RTEMS Root File System Configuration

- RTEMS is supplied with a basic in-memory file system which serves as the root for mounting additional file systems and devices.
- By default, RTEMS is configured to use a very simple in-memory root file system, “minIMFS”
- “minIMFS” is very basic and does not support mount points, so it must be replaced with “IMFS” a more robust in-memory file system.
- To effect the replacement, the following defines should be include in the build procedure:
  - `#define CONFIGURE_FILESYSTEM_IMFS`
  - `#define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM`
- Location of IMFS (and other FS) code: cpukit/libfs/src

RTEMS Init Task

- The RTEMS “Init” task is started up upon system initialization. It is akin to the “main” function in a C application program.
- The “Init” task performs general configuration and housekeeping functions and starts up additional system tasks. It may exit once those other tasks have been started.
- As part of the initialization, file systems are registered. After registration they become known to the operating system and devices formated for that file system may be mounted under the root IMFS.
- Registration provides the file system name and a pointer to the file systems mount function. A call to initialize the new file system (obtain semaphores, initialize basic data structures) may also be performed at this time.

```
/*
 * Register the aplffs file system with RTEMS
 *
 * First we register the aplffs with RTEMS. This inserts the
 * aplffs mount me handler into the file system heap with the
 * appropriate identifying tag.
 */
errno = 0;
status = rtems_filesystem_register( "aplffs", aplffs_fsmount_me_h);
if (status != 0)
{
    printf("Registration of APLFFS failed: %x\n", errno);
}
else
{
    printf( "aplffs file system  has been registered\n");
}
```

FS Name

FS Mount Function

APL

Mounting a Device – Create the Mount Point

- **Once the custom file system has been registered and initialized, a device may be associated with it by mounting it.**
  - This does not have to take place during or in the RTEMS Init task.
  - The device is optional, for instance, an EEPROM file system might be located in a well-known address space rather than on an external device. A means of providing “extra information,” for instance, and address range, during the mount is provided.
- **Create a directory that will serve as the file system’s mount point:**
  - `status = mkdir("/recorder01/" 0777);`
  - Note that the path ends in “/”. It is rumored that this is necessary. I can neither confirm nor deny, but it works.
  - While the directory used to mount the file system may contain files, those files will become inaccessible after the mount operation. Unmounting will restore accessibility.

Mounting a Device - Mount

```
status = mount(“/dev/SSR01”, "/recorder01/",
"aplffs", RTEMS_FILESYSTEM_READ_WRITE, NULL);
```

- **Arguments:**
  - The device: “`/dev/SSR01`” (Optional)
  - The mount point: "`/recorder01/`"
  - The file system name, as registered: "`aplffs`"
  - Options: `RTEMS_FILESYSTEM_READ_WRITE`
  - Pointer to file system specific data: `NULL` (Optional)
- **After mounting, the device may be accessed using the normal file system operations: OPEN, CLOSE, READ, WRITE, etc.**
- **The device may be dismounted:**
  - `status = unmount( “/recorder01/” );`

File System Implementation

File System Level Handlers

APL

File System Handlers

- **An RTEMS file system consists of a collection of well defined functions, called file system handlers. These are grouped into several classes:**
  - File System Level Operations, for instance, mount
  - File Operations, for instance, open, close, read, write
  - Directory Operations, once again, open, close, read, write
- **Each group of operations has a corresponding table that stores pointers to the functions. The table may then be used by the higher level operating system code to call those functions.**
- **There is also a table of limits, sizes and options, for instance, the maximum file name length.**

File System Level Handlers

- **Required file system level handlers include:**
  - "Mount Me" and "Unmount Me"
  - "Evaluate Path" and "Evaluate Path for Make"
  - "Make Node" and "Unlink"
  - Miscellaneous Internal Functions: "Node Type" and "Free Node"
- **Others, such as those for hard and symbolic links are optional.**
- **The file system handlers are the more fundamental and characteristic of RTEMS file IO. Therefore, they will be the primary focus of the presentation.**

Mount & Mount Me

- Two sets of mount handlers: “mount” / “unmount” and “mount me” / “unmount me.”
- Mount is called when the RTEMS mount is directed to create a file system with one of your file system’s directories as the root node.
- Mount Me is called when the RTEMS mount is directed to create an instance of your file system with some *other* file system’s directory as the root node.
- Allowing other file systems to be mounted within your directory tree is not required, so the “mount” handler is optional.
- Allowing another file system to mount your file system within its directory tree is how your file system is added to the IMFS root, so the “mount me” handler is required!

Mount Me Interface

```
int aplffs_fsmount_me_h
    (
    rtems_filesystem_mount_table_entry_t *mt_entry,     /* IN */
    const void                           *data          /* IN */
    )
```

- **The data argument is the file system specific data pointer from the RTEMS mount call.**
- **The mt_entry argument contains information required by the custom file system to instantiate the file system under the mount point:**
  - **`mt_entry->dev`**: The device that is being mounted ("/dev/SSR01" in our example)
  - **`mt_entry->fs_info:`** Information the file system needs to remember at the root. Might be read from the device being mounted. For instance, a pointer to the FAT.
  - **`mt_entry->mt_fs_root.node_access:`** Information the file system needs to remember to perform operations on a specific node (file or directory).
  - **`mt_entry->mt_fs_root.node_access_2:`** More information for node access.
  - **`mt_entry->pathconf_limits_and_options:`** The file system's limits and options table.
  - **`mt_entry->mt_fs_root.ops:`** The file system level handler table.
  - **`mt_entry->mt_fs_root.handlers:`** The node specific handler table.

APL

Mount Me Processing

- **Using the device name (mt_entry->dev) open the device and access any information needed to create the file system data structures specific to it.**
  - For instance, you might need to read the file allocation table from a specific device location.
  - APLFFS scans the entire device and builds a balanced tree structure containing data objects for each directory and file. This is rooted in a general structure representing the device.
  - Initalize the mt_entry->fs_info based on this information. APLFFS sets this to a pointer to the “device structure” initialized during the scan.
- **Initialize the mt_fs_root structure based on the information specific to the root node.**
  - APLFFS initializes node_access to the address of the in-memory object that represents the root directory of the file system tree. APLFFS does not use node_access_2.
  - pathconf_limits_and_options is a pointer to the file system’s limits and options table.
  - ops is a pointer the file system’s file level handler table.
  - handlers is a pointer to the appropriate node-specific handler table. Since the file system root node is a directory (not an ordinary file) the handler table is the directory operations table.

Path Evaluation and Evaluation for Make

- The path evaluation handler evaluates a pathname within the domain of the file system, and updates the file system location info data structure so that further operations may be performed on the node (file or directory).
- The evaluation for make handler is similar, but the named node does not yet exist. The information returned is for the parent node (necessarily a directory) and the node's name is extracted from the full path.

Eval Path Processing

- **“pathname” is initialized to the portion of the path within the file system that is to be evaluated.**
  - In our example, an open on “/recorder01/instrument02/comp.lut” would be passed in as “instrument02/comp.lut”. A file in the root directory would be “file.ext” and the root directory itself would be an empty string “”.
- **The input value of the “pathloc” structure is initialized to the values set in mt_entry->mt_fs_root during the mount operation.**
  - pathloc->node_access and node_access_2 are pointers to the object(s) (or other information) that represent the file system’s root node.
  - pathloc->mt_entry is a pointer to the file systems mount table entry.
  - pathloc->mt_entry->fs_info is a pointer to the general file system information structure initialized during the mount operation.
  - pathloc->mt_entry->dev is the device name of the associated device.
- **The pathloc structure is initialized to the information provided for the root by the mount operation. The file system information structures from pathloc are used to parse the pathname in order to locate the named node. The pathloc structure is then updated to represent it.**
  - The device may have to be accessed in order to locate the node. The device name or the fs_info structure might be used to do this.
  - node_access (and, if used, node_access_2) will be updated with pointers to the node’s objects (or other information) for subsequent use.
  - Since pathloc is initialized to the root node and the root node is by definition a directory, the pathloc->handlers may have to be updated if the evaluated node is a file.
  - pathloc->ops and pathloc->mt_entry should not be updated.
- **IMPORTANT !!! If the path does not evaluate to an existing node, BE SURE to correctly set `errno = ENOENT`.**
  - The open function uses the errno value ENOENT (Error, No Entity) to indicate that the node should be created. If the proper errno is not set, the open function will not create the node.

Eval Path Processing Complications

- The path to be evaluated may include the relative directory symbol “..” which should be handled correctly. It may also include multiple concatenated name separators (“////”) that should be treated as one (“/”).
- If the path to be evaluated includes a symbolic or hard link, the link has to be evaluated in order to find the real, honest-to-goodness, actual node. Links are not supported in the APLFFS, so you should reference existing RTEMS file system code to learn more.
- If the path to be evaluated includes the mount point of another file system that is rooted in your file system, you must call the evaluation handler for the mounted file system. Mount points for file systems is not supported in the APLFFS, so you should reference existing RTEMS file system code to learn more.
- The flags argument is an indicator of the caller’s permissions. They should be evaluated against the evaluated node’s authorizations. Since the APLFFS does not support restrictions on file access based on permissions, you should reference existing RTEMS file system code to learn more.
- pathname and pathnamelen may be empty / 0. This indicates that the node to be evaluated is the file system’s root node. In this case, pathloc is already correctly initialized and need not be updated.

Eval for Make Interface

```
int aplffs_eval4make_h
(
    const char      *pathname,    /* IN */
   rtems_filesystem_location_info_t *pathloc,/* IN/OUT */
   const char      **name        /* OUT    */
)
```

- **Note differences from Path Evaluation:**
  - No path name length
  - New return argument, name. Returns the name of the node to be created, extracted from the path.

Eval for Make Processing

- **Processing is similar to the path eval handler, except that the end node is assumed not to exist, and parsing stops when the parent (directory) node is located.**
- **The pathloc structure is updated for the parent (directory) node.**
- **A pointer to the (non-existent) node name is returned in the name argument.**
  - No memory is allocated, this is a pointer into the pathname string argument.

Eval for Make Processing Notes

- Since the new, non-existent node must have a name, pathname cannot be empty / null. Set errno to EINVAL (Error, Invalid Input).
- Since the parent node must be a directory, if it is not errno is set to EINVAL (Error, Invalid Input).

Make Node Handler

```
int aplffs_mknod_h
(
 const char  *path,        /* IN */
 mode_t       mode,        /* IN */
 dev_t        dev,         /* IN */
 rtems_filesystem_location_info_t  *pathloc /*IN/OUT*/
)
```

- **The pathloc argument contains node access information for the node which will contain the new node to be created. This must be a directory node.**
- **The path argument contains the node's name, not the full path. This is the returned name argument value from the eval for make handler.**
- **The dev argument may be used to access the device as necessary.**
- **The mode argument specifies what type of node is to be created, for instance, directory or regular data file in the "IFMT" field.**
  - (mode & S_IFMT) may have a value of S_IFDIR (directory); S_IFREG (regular data file); S_IFCHR (character oriented file); S_IFBLK (block oriented device); S_IFLNK (symbolic link); S_IFSOCK (socket) or S_IFFIO (FIFO or pipe).
- **Create the requested node type in the specified directory, and then update the pathloc argument (node_access, node_access_2 and handlers) for the new node.**

APL

Unlink Handler

```
int aplffs_unlink_h
    (
    rtems_filesystem_location_info_t  *parent_pathloc, /* IN
*/
    rtems_filesystem_location_info_t  *pathloc         /* IN
*/
    )
```

- The parent_pathloc argument is the location information for the parent node, necessarily a directory node.
- The pathloc argument is the location information for the node to be deleted.
- Use pathloc->node_access and node_access_2 along with parent_pathloc->node_access and node_access_2 to delete the node.
- In the case that pathloc describes a directory node, all nodes in the subtree should be deleted.

Node Type Handler

```
rtems_filesystem_node_types_t aplffs_node_type_h
(
   rtems_filesystem_location_info_t  *pathloc  /* in */
)
```

- The pathloc argument contains access information for the node in question.
- Use pathloc->node_access and node_access_2 to determine and return the type of node – directory, regular file, etc.
- Type values are defined in libio.h as the enum rtems_filesystem_node_types_t.

Free Node Handler

```
int aplffs_freenod_h
(
   rtems_filesystem_location_info_t  *pathloc  /* in */
)
```

- During the path evaluation and path evaluation for make processing, it may be necessary to allocate and maintain memory until the file operation is complete. After the operation is complete, this handler is called to allow for freeing any allocated memory.
- The pathloc argument contains access information for the node that was evaluated and operated on.
- Use pathloc->node_access, node_access_2, and pathloc->mt_entry->fs_info to free any memory that was tempora.

Summary: Open / Create

- **The following calls are made when the system open call is made to create a file:**
  - The system OPEN procedure parses the path it is given to the mount point of the file system. It then uses the handlers found at the mount point to complete the processing.
  - Path Evaluation Handler is called with the (shortened) path to the file and the root node of the file system. The handler parses the path and finds that the file does not exist. It returns NOENT.
  - If NOENT is returned and the CREATE option was specified, the OPEN procedure calls Path Evaluation for Make. The handler parses the path and returns the directory to create the node in and the new node's name.
  - The OPEN procedure now calls the Make Node handler to actually create the node.
  - The OPEN procedure once again calls the Path Evaluation handler, just to make sure everything went right. Since the node now exists, the node's pathloc information is returned.
  - The OPEN procedure now calls the file system's Open File handler. This handler is our next topic of discussion.
  - Upon successful completion of the Open File handler, a file handle is returned for the application process to use in subsequent calls.

File System Implementation

File Operation Handlers

APL

File Operation Handlers

- File Operations include open, close, read, write, lseek, ftrucate, fsync and fdatasync.
- Other (optional) operations include ioctl, fstat, chmod, pathconf, fcntl and rmnode.
- We introduce a new data structure to represent an open file. This structure is in libio.h

```
struct rtems_libio_tt {
  rtems_driver_name_t               *driver;
  off_t                              size;      /* size of file */
  off_t                              offset;    /* current offset into file */
  uint32_t                           flags;
  rtems_filesystem_location_info_t   pathinfo;
  rtems_id                           sem;
  uint32_t                           data0;     /* private to "driver" */
  void                              *data1;     /* ... */
};
```

- The file id returned by the system OPEN call is a pointer to this structure, which might be termed a “file descriptor”

APL

Open Handler

```
int aplffs_file_open
    (rtems_libio_t *iop,
     const char *pathname,
     uint32_t flag,
     uint32_t mode)
```

- **The iop argument contains the pathloc (now called pathinfo) structure filled in during the system OPEN call.**
- **The flag and mode arguments are those passed to the system OPEN call.**
- **The iop->pathinfo->node_access and node_access_2 can be used to access the file system node information.**
- **The iop->sem (semaphore) can be used to serialize access between several callers.**
- **The open handler sets the iop->size value to the size of the file and iop->offset to zero, the beginning of the file.**
- **Upon return, the system OPEN call returns an index to the iop structure, which is the "handle" the caller uses to perform file io operations.**

APL

Read and Write Handlers

```
ssize_t aplffs_file_read
    (rtems_libio_t *iop,
     void          *buffer,
     size_t         count)
```

```
ssize_t aplffs_file_write
    (rtems_libio_t *iop,
     const void *buffer,
     size_t count)
```

- **The iop argument provides access to the specific node, indicates the offset from which to read or write, provides a semaphore for access control.**
- **The data is transferred from buffer to the file or from the file to buffer. The count argument provides the size of the transfer.**
- **The number of bytes actually transferred is returned.**

APL

Read and Write Handler Distinctions

- **READ: The system caller will update iop->offset based on the return value – don't do it in your handler!**
- **WRITE: The custom file system handler must update both iop->size and offset. This makes sense, since only the handler can tell if a write affected the file size.**

Other File Handlers

- Other file handlers receive the iop argument and any additional information that pertains to the operation.
- The implementations are straightforward, given the node access information in the fileinfo structure.

Directory Handlers

- If the nodes being accessed is a directory (for instance, for a ls style directory listing) the nodes handler table will be the directory handlers.
- We have not yet implemented the directory handlers, so we do not have any special insight to share at this time.

Questions?

APL

File System Level Operations

```
/*************************************************************************
 * The aplffs_operations_table contains function pointers to the
 * file system level methods for the APL Flash File System.
 *************************************************************************/
struct _rtems_filesystem_operations_table  aplffs_operations_table =
{
    aplffs_eval_path_h,     /* evalpath file system handler */
    aplffs_eval4make_h,     /* evalformake file system handler */
    NULL,                   /* link file system handler */
    aplffs_unlink_h,        /* unlink file system handler */
    aplffs_node_type_h,     /* node_type file system handler */
    aplffs_mknod_h,         /* mknod file system handler */
    NULL,                   /* chown file system handler */
    aplffs_freenod_h,       /* freenod file system handler */
    NULL,                   /* mount file system handler */
    aplffs_fsmount_me_h,    /* mount_me file system handler */
    NULL,                   /* unmount file system handler */
    aplffs_fsunmount_me_h,  /* fsunmount_me file system handler */
    NULL,                   /* utime file system handler */
    NULL,                   /* eval_link file system handler */
    NULL,                   /* symlink file system handler */
    NULL,                   /* readlink file system handler */
    NULL,                   /* rename file system handler */
    NULL                    /* statvfs file system handler */
};
```

APL

File Operations

```
/*************************************************************************
 * The aplffs_file_handlers table contains function pointers to the
 * file handler methods. These are the appropriate operators for
 * nodes of type file.
 *************************************************************************/
struct _rtems_filesystem_file_handlers_r aplffs_file_handlers = {
     aplffs_file_open,       /* open file handler */
     aplffs_file_close,      /* close file handler */
     aplffs_file_read,       /* read file handler */
     aplffs_file_write,      /* write file handler */
     NULL,                   /* ioctl file handler */
     aplffs_file_lseek,      /* lseek file handler */
     NULL,                   /* fstat file handler */
     NULL,                   /* fchmod file handler */
     aplffs_file_ftruncate,  /* ftruncate file handler */
     NULL,                   /* fpathconf file handler */
     aplffs_file_fsync,      /* fsync file handler */
     aplffs_file_fdatasync,  /* fdatasync file handler */
     NULL,                   /* fcntl file handler */
     NULL                    /* rmnode file handler */
};
```

APL

Directory Operations

```
/*************************************************************************
 * The aplffs_dir_handlers table contains function pointers to the
 * directory handler methods. These are the appropriate methods for
 * nodes of type directory.
 *************************************************************************/
struct _rtems_filesystem_file_handlers_r aplffs_dir_handlers =
{
     aplffs_dir_open,    /* open directory handler */
     aplffs_dir_close,   /* close directory handler */
     aplffs_dir_read,    /* read directory handler */
     NULL,  /* write directory handler */
     NULL, /* ioctl directory handler */
     NULL, /* lseek directory handler */
     NULL, /* fstat directory handler */
     NULL, /* fchmod directory handler */
     NULL, /* ftruncate directory handler */
     NULL, /* fpathconf directory handler */
     NULL, /* fsync directory handler */
     NULL, /* fdatasync directory handler */
     NULL, /* fcntl directory handler */
     NULL  /* rmnode directory hadler */
};
```

APL

Mount Table Entry

```
struct rtems_filesystem_mount_table_entry_tt
{
  rtems_chain_node                    Node; /* Next entry in linked
list */
  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;

  const char      *target; /* The mount point of the file system. */
  const char      *type;   /* The type / name of the file system. */
  char            *dev;    /* The device to mount */
};
```

- **Location: cpukit/libcsupport/include/rtems/libio.h**

APL

File System Location Information

```
struct rtems_filesystem_location_info_tt
{
   void                                        *node_access;
   void                                        *node_access_2;
   const rtems_filesystem_file_handlers_r      *handlers;
   const rtems_filesystem_operations_table     *ops;
   rtems_filesystem_mount_table_entry_t        *mt_entry;
};
```

- **Location: cpukit/include/rtems/fs.h**

Path Evaluation Interface

```
int aplffs_eval_path_h
    (
       /*IN The path to be evaluated */
       const char *pathname,

       /*IN The length of the path string */
       size_t pathnamelen,

       /*IN The permission flags    */
       int    flags,

       /* IN/OUT The evaluated location structure */
       rtems_filesystem_location_info_t *pathloc
    )
```