CSC 714 : Final Project Report

Gayathri TK gtambar@ncsu.edu

Jayush Luniya jrluniya@ncsu.edu

RTFS : Real Time File System

Project Url

http://www4.ncsu.edu/~jrluniya/rt/

Abstract

Incorporating a web server to an embedded device provides a powerful mechanism allowing users to monitor and control embedded applications using any standard browser. Web enabling devices provides a new method of interfacing to devices that requires essentially no target side programming and works with universally available, standard client software. The web server uses a file system to store embedded web pages in RAM, flash, or on disk. File systems provide capabilities for changing web pages dynamically and maintaining dynamic objects. Pages can be protected with password security to restrict both read and write access. Hence the need for a real-time file system on an embedded device.

The goal of this project is to implement a Real Time File System (RTFS) for Renesas M16C board. RTFS is a RAM-based file system which provides real time guarantees on file access times. The Renesas M16C board typically acts as a base station for a cluster of sensor nodes and aggregates data to exchange across clusters. By web-enabling the device, the data can also be available to other base stations and high-end machines in a heterogeneous environment.

Description

Real time file system need to address different set of design issues not handled by normal file systems.

- RTFS should be a primitive file system with low metadata overhead due to resource constraints on the embedded device
- Typical embedded devices do not have hard disks and hence the file system should be a memory based file system
- RTFS should support dynamic creation and deletion of files, directories, and links with full read and write capability. Not a static ROM-image file system
- RTFS should be able to provide real time bounds for file access operations.
- RTFS should have a thin hardware dependant layer to maximize ease of porting to new architecture. Also RTFS should be minimally dependent on the underlying RTOS.

We have implemented RTFS as a RAM based file system with very little file system overhead. The initial work in this project was to come up with very simplistic data structures to be used for maintaining the file system metadata. We then decided the APIs that need to be implemented and tested them by building a mini shell.

The next step was to port the file system to Renesas M16C board and test and benchmark the file system operations. This report explains in detail the portability issues and technique used for benchmarking the file system operations.

Portability issues

Once we implemented the file system on a normal Linux machine, we worked on porting our code to Renesas M16C board. The board comes with SDK which has a very primitive C compiler. The following were some of the issues we faced while we were porting our file system to Renesas M16C architecture:

- We had to statically allocate all the memory needed for our file system as malloc() was failing
- Function activation frame size is only 255 bytes. Hence cannot allocate more than 255 bytes for local variables inside any function.
- There was absolutely no memory protection. We found our data structures were getting corrupted because of collision between the call stack and global data area.
- Need to explicitly cast pointers as FAR pointers while assigning a pointer in global area to local variable or when passing them as function arguments

Testing the file system

In order to test our file system on the architecture, we decided to simulate the shell environment on the M16C processor using input and output buffers. The input command buffer contains all commands, like mkdir, touch, read etc. that need to be executed. The test code reads the commands one by one and calls the corresponding shell routine which internally calls the file system api to execute the command. for eg. shell command *mkdir* calls the file system api *f_mkdir* to actually create the directory in the file system. For *append* command, which requires contents of the file to be retrieved from stdin, we implemented *rtfs_getc()* method to return characters from buffer instead of standard input stream.

The output is written to a buffer by calling *rtfs_printf()* instead of normal *printf()*. After each command is executed, the contents of the buffer can be viewed by using KD30, a debugging tool available with Renesas board SDK.

Providing Real Time Guarantees to RTFS

In order to provide real time guarantees to our file system, we had to put a bound on the following parameters:

<u>Directory related parameters</u>

- Maximum number of directories in the file system : MAX_DIR_ENTRIES 32
- Maximum number of subdirectories inside a directory : MAX_SUB_DIRS 5
- Maximum depth of a directory in the hierarchy : MAX_DIR_DEPTH 5
- Maximum characters in directory name : MAX_DIR_NAME_SIZE 8

<u>File related parameters</u>

- Maximum no of files in a directory : MAX_FILE_ENTRIES 4
- Maximum no of data blocks in a file : NUM_BLKS_PER_FILE 4
- Maximum file size (NUM_BLKS_PER_FILE*DATA_BLK_SIZE) : MAX_FILE_SIZE
- Maximum characters in file name : MAX_FILE_NAME_SIZE 8

<u>File system metadata parameters</u>

- Maximum number of file descriptors : MAX_FILE_DESC 32
- Maximum number of data blocks in the file system : MAX_DATA_BLKS 32

- Data block size : DATA_BLK_SIZE 128

RTFS Benchmarking

Unlike most general-purpose applications, embedded applications often have to meet various stringent constraints, such as time, space, and power. Constraints on time are commonly formulated as *worst-case* (WC) constraints. If these timing constraints are not met, even only occasionally in a hard real-time system, then the system may not be considered functional. The *worst-case execution time* (WCET) must be calculated to determine if a timing constraint will always be met.

For our file system to be used as a real time file system, we should be able to provide average and worst case scenarios for all of the file system operations.

Renesas M16C Timers

Renesas SDK does not provide functions like gettimeofday(), which are used to measure the time taken for a certain module in normal applications. Hence we had to use timers provided in architecture to calculate time bounds for file system operations. We had used Timer-mode timer for this purpose.

The MCU has 11 timers. The timers are separated into two categories by functionality, 5 Timer A's and 6 Timer B's. All 11 timers can operate in Timer Mode. We had used Timer A0 in timer mode. In Timer Mode, the counter register counts down using the selected clock source until the counter underflows (0000 to FFFFh). At this point, the timer interrupt request bit is set and the contents of the reload register are loaded back into the counter and countdown continues.

We had implemented a timer with 1 millisecond granularity. We used two counters in the timer:

- count - counts the number of seconds since the last reset
- time_cnt - counts the number of milliseconds since the last reset

Since each of the file operations were taking less than 1 millisecond, we measured time taken for a series of similar operations and averaged them out to get the average and worst case execution times.

Test cases used for benchmarking

For each of the operations, we had to come up with scenarios to accurately calculate the average and worst case access times. The test cases used are listed below:

- **mkdir, rmdir**

Average case: Created/deleted 2 directories under root, at level 1, 6 directories in level 2, 5 directories in level 3, 2 directories in level 4 and 5 directories in level 5, which is MAX_DIR_DEPTH

Worst case: Since we have maintaining the directory hierarchy as shown in the figure below, the

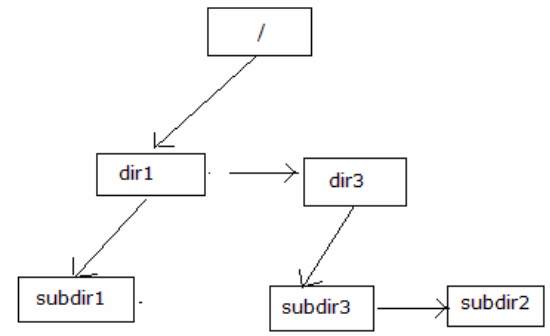

Figure 1: Directory Hierarchy

worst case access time would be access the last subdirectory at level = MAX_DIR_DEPTH. If MAX_DIR_DEPTH = 2 and MAX_SUB_DIRS=2, then in Fig 1, access creating/removing subdir2 would give the worst case access time.

- **creat, close, open, remove**

  Average case: Created 4 files under directories in level 1, 4 files under directories in level 2, 4 files in level 3, 1 file in level 4 and 6 files under directories in level 5 (MAX_DIR_DEPTH)

  Worst case: Accessing any file in a directory which is at level = MAX_DIR_DEPTH and is at the end of the subdirectory chain will lead to worst case access time for that file.

- **write, read**

  Average case: Wrote/read one data block of data into/from each of the files created above

  Worst case: There should not be any significant difference between worst and average case access time for read and write operations as they access files using file descriptors, which is basically a pointer to the File control block (FCB). This can clearly be observed from the experimental results. We measured the time taken to write to all blocks allocated to a file and read all data blocks of a file created in a directory at the highest depth.

Benchmark Results

The following table shows our experimental results.

Operation	Average-case Access Times			Worst-case Access Times		
	# Access	Total time taken	Time/access	# Access	Total time taken	Time/access
		(in millisecs)	(in millisecs)		(in millisecs)	(in millisecs)
mkdir	20	7	0.35	20	12	0.6
rmdir	20	7	0.35	20	12	0.6
creat	20	11	0.55	8	6	0.75
open	20	10	0.5	8	6	0.75
remove	20	13	0.65	8	7	0.875
read	25	13	0.52	20	12	0.6
write	25	13	0.52	20	11	0.55

Future Work

- **Reducing Average and Worst case bounds**

  While we were interested in building a very simplistic file system, we did not have a chance to look or evaluate our code to see if we could optimize our file system to improve average or worst case access time. One possible future work could be to optimize our file system code to improve the real time bounds provided above.

- **Porting the file system to Flash Memory – Persistent file system**

  As we had mentioned before, RTFS is a RAM-based, which makes it a temporary. Data can be stored only until power is available. If data must be available across power failures, there is a need to build a persistent file system, such as a flash file system. Flash file systems have

their own limitations, which were discussed in detail in our prior report. Renesas M16C board comes with 384K flash memory, and hence porting RTFS to flash memory would be a very interesting future work of this project.

Individual Contributions

Since this project milestone involved working on Renesas M16C board, we chose to work together in reading about the architecture, coming up with test scenarios, taking benchmark results and in writing the final project report.

References

[1] FAT Filesystem http://users.iafrica.com/c/cq/cquirke/fat.htm

[2] Transaction-Safe FAT File System http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcemain4/html/cmcontransaction-safefatfilesystem.asp

[3] Flash Filesystems for Embedded Linux Systems http://www.linuxjournal.com/node/4678/print

[4] TargetFFS: An Embedded Flash File System http://www.blunkmicro.com/ffs.htm

[5] Renesas M16C Architecture http://www.renesasinteractive.com

[6] Algorithms and Data Structures for Flash Memories http://www.cs.tau.ac.il/~stoledo/Pubs/flash-survey.pdf

[7] A Transactional Flash File System for Microcontrollers http://www.cs.tau.ac.il/~stoledo/Pubs/usenix2005.pdf

[8] ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes http://www.cs.colorado.edu/~rhan/Papers/sensys_elf_external.pdf