RAM Based File System
for
HTTP daemon on Renesas M16 board

Cuong Phu Nguyen
cpnguyen

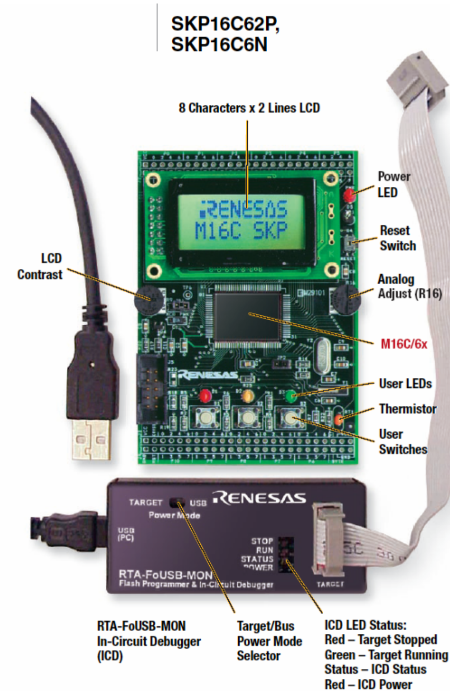
Kyung Chul Lee
kclee

CSC 714 – Real Time Computer Systems

Dr. Mueller

November 30, 2005

1. Introduction



The Renesas M16 offers a robust platform of 16-bit CISC featuring high ROM code efficiency, extensive EMI/EMS noise immunity, ultra-low power consumption, high-speed processing in actual applications, and numerous and varied integrated peripherals. Extensive device scalability from low- to high-end MCU series, featuring a single architecture as well as compatible pin assignments and peripheral functions, provides support for a vast range of application fields. The Renesas M16 is equipped with two 126kB RAMs, a 10/100 BaseT port, a RS232 port, a LCD, three LEDs, etc. It comes with the Renesas software development tools which include a complete software development tool chain including, HEW (IDE, GUI), NC30WA (C-compiler, assembler, and linker), KD30 (Debugger), and FoUSB (Flash-over-USB Programmer). A real-time, source-level debug environment is implemented using the KD30 debugging software with the RTA-FoUSB-MON Flash Programmer/In-Circuit Debugger (ICD). The Flash-over-USBTM(FoUSB) Programmer software, with the ICD, allows in-system programming. The ICD and firmware provide a convenient USB (Universal Serial Bus) interface between the board and the host PC. This interface reduces resource requirements on the processor.

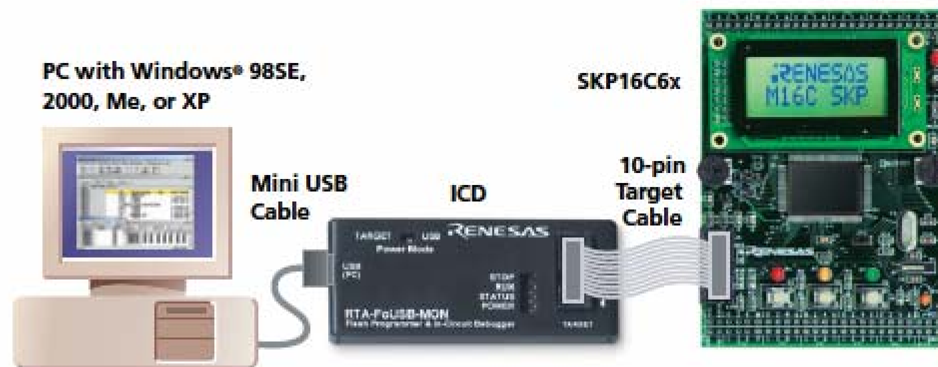
Our project is aiming to implement a filesystem on Renesas to be used by a HTTP daemon. The existing network stack is implemented as a single dispatcher, which bases on the input packet types to invoke the appropriate protocol entities. HTTP daemon is on of such protocols running on top of TCP layer. For each new connection to HTTP daemon, data can be read or written. In order to operate, the HTTP daemon uses a set of filesystem APIs to store, retrieve and modify user data. As a goal of the project, the HTTP daemon needs to handle sensors data, collected by other group's projects. The sensor data will be sent to the HTTP daemon via 10/100 BaseT port or via 801.11b/g interface. As data need to go in and out fast enough, we need to implement a RAM filesystem and provide the HTTP daemon with a set of APIs. As the RAM capacity of the board is limited, the filesystem needs to be compact enough. Hence the lengths of file name and the number of directory levels should be limited.

2. Approach.

We will adopt a step-by-step approach to achieve our goal and break down our project into four different stages. However, as we approaching the problem, there may be more sub stages to be added.

Stage 1 - Get familiar with the Renesas board and environment.

There board comes with an integrated development environment for Windows. This IDE allow us to create and maintain the project, compile the code, and download the code from PC through the USB port. The debugging capability is also provided. There are open source software modules, such as BSP, drivers, and sample applications. These open source programs need to be explored in order to get familiar with the board hardware. Ideas about specific implementation may come from these source codes too.



1. **M16C StaterKit Plus Software Install**

   We installed software from the CD-ROM came with M16C. It asked for the serial number and we could find under the actual board after we detach it from the ether board.

2. **USB Driver Installation**

   We also had to install the driver for the RTA-FoUSB-MON In-Circuit Debugger (ICD) before we could use it. First, we verified that the Target Power Mode switch is in the USB position so that the power to the board will be supplied from the either board not from USB.
   Next, we connected one end of the mini USB cable into the ICD and the other end into our PC's USB port. The red "Power" LED on the ICD lit up and the yellow "Status" LED started blink.

Stage 2 - Design a RAM file system and a set of API.

We may take a look at existing filesystem like Extended File System, FAT, etc. to design our own RAM file system. Based on the requirement and the limitation of the RAM

space, we may not implement the fancy features like long file names, extended number of file entries, too many directory levels, etc. The file system may have the following functional blocks:

File/Directory entry blocks.
This block will be searched first in order to seek for a file. The total number of entries will be fixed. Each entry contains the file name, file size and a pointer to file location.

File data block.
This block contains the actual data for a file. The advantage of the RAM file system is that it is more flexible than other sect based file system. On the other hand, we need to take into account memory allocation in our code.

The set of APIs to maintain and manipulate the filesystem, as well as to provide services for the HTTP daemon. Since the task structure is simple, the API may not need to deal with the re-entrence issue: only one function is called at a time. One of the top priority requirement to the APIs is the execution time need to be short. There won't be any blocking waits.

We had developed APIs as following:

```
/* definition for return code */
#define FS_SUCCESS_C        0
#define FS_GEN_ERROR_C      1
#define FS_MEM_ERROR_C      2
#define FS_NAME_ERROR_C     3
#define FS_UNKNOWN_ERROR_C  4


/* file access mode */
#define FS_READ_ONLY                  0
#define FS_WRITE_ONLY                 0
#define FS_READ_WRITE                 0

/*
CREATE A FILE

- path: the absolute path name of a file to be created
- size: the final size of a file.  The size is fixed and can not be chaged.

fs_create() returns 0 for success or 1 otherwise
*/

INT16 fs_create(FAR char* path, INT16 size);



/*
OPEN A FILE

- path: the asolute path name of a file.
- mode: one of the following FS_READ, FS_WRITE, or FS_READ_WRITE

open() return the file descriptor or -1 if an error occured.
*/

fs_fileHandle_t * open(FAR char* path, INT16 mode);
```

```
/*
CLOSE A FILE

- fd: file descriptor

close() returns nothing
*/

void close(fs_fileHandle_t * fd);



/*
READ FROM A FILE DESCRIPTOR

- fd: file descriptor
- buf: data to be stored into
- count: number of bytes to read

read() attempts to read up to count bytes from file descriptor fd into the buffer
starting at buf.
If count is zero, read() returns zero and has no other results.
*/

INT16 read(fs_fileHandle_t * fd, FAR char *buf, INT16 count);


/*
WRITE TO A FILE DESCRIPTOR

- fd: file descriptor
- buf: data to be stored from
- count: number of bytes to write

write() writes up to count bytes to the file referenced by the file descriptor fd
from the buffer starting at buf.
*/

INT16 write(fs_fileHandle_t * fd, FAR char *buf, INT16 count);


/*
REPOSITION READ/WRITE FILE OFFSET

- fd: file descriptor
- buf: data to be stored from
- count: one of the following
       FS_SEEK_SET: The offset is set to offset bytes.
       FS_SEEK_CUR: The offset is set to its current location plus offset bytes.
       FS_SEEK_END: The offset is set to the size of the file plus offset bytes.

seek() repositions the offset of the file descriptor to the argument offset
according to the directive whence.
*/

INT16 seek(INT16 fd, INT16 offset, INT16 whence);


/*
MAKE A NEW DIRECTORY

- path: the absolute path name of a directory to be created
- max_size: directory size

mkdir() return 1 for sucess or 0 otherwise
*/

INT16 mkdir(FAR char *path);
```

```
/*
READ DIRECTORY LIST

- path: the absolute path name of a directory to be read

readdir() list of strings containing filenames
*/


FAR char** readdir(FAR char *path);

/*
FORMAT FILE SYSTEM

fs_format() return 0 for sucess or 1 otherwise (general error)
*/
INT16 fs_format();

/*
PRINT INFORMATION ABOUT THE FILE SYSTEM

fs_print_all() prints all the file system to the screen for debugging
*/
#ifdef SIMULATION
void fs_print_all();
#endif
```

The file system specification as following:

Directory Entries

<table>
<tr><td>1 byte</td><td>The maximum number of files</td></tr>
<tr><td>1 byte</td><td>The number of existing files</td></tr>
<tr><td>Multiple of 20 bytes</td><td>File Entries
<table>
<tr><td>11 bytes</td><td>Filename (maximum 10 characters)</td></tr>
<tr><td>1 bytes</td><td>File type
- 0 for file
- 1 for directory</td></tr>
<tr><td>4 byte</td><td>File size</td></tr>
<tr><td>4 byte</td><td>File Location</td></tr>
</table></td></tr>
</table>

File Descriptor

<table>
<tr><td>1 byte</td><td>Type
- 0 for file not in use
- 1 for file currently being read
- 2 for file currently being written</td></tr>
</table>

	- 3 for file currently being read and written
4 byte	Operation cursor byte offset from the begging of the file
4 byte	File Location (page index)
4 byte	File Size (# pages)

Page Counter

4 byte	Available Free Space

We are currently implementing this file system on the Linux environment first in order to prove the concept of the algorithm.

Stage 3 - Improvement for the HTTP daemon

As we are implementing the project, we originally planned to make improvements on the code of the HTTP daemon. To some extent, we can event improve the protocol stack if needed. However, this stage was not performed due to insufficient time.

Stage 4 - Test module.

We wrote a testing script which creates directories and files and performs read and writes on a Solaris machine to verify the correctness of the program.

```
Beginning testing the file system...
number of actually created directory: 10
max directory creation time is 3214480556
min directory creation time is 3214480556
average directory creation time is 208003448
directory tests are passed succesfully
number of actually created files: 20
max file creation time is 3214480492
min file creation time is 3214480492
average file creation time is 208003384
file creation tests are passed succesfully
number of actually written files: 20
max file writing time is 3214480492
min file writing time is 3214480492
average file writing time is 208003384
file writing tests are passed succesfully
data read from file /entry_1
 0
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
```

```
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112
113 114 115 116 117 118 119 120 121 122 123 124 125 126 127  0

...

number of actually read files: 20
max file reading time is 3214480492
min file reading time is 3214480492
average file reading time is 208003384
file reading tests are passed succesfully
-----------------------
Entering directory /
-----------------------
Entering directory entry_1
Leaving directory entry_1
-----------------------
-----------------------
Entering directory entry_2
Leaving directory entry_2
-----------------------
-----------------------
Entering directory entry_3
Leaving directory entry_3
-----------------------
-----------------------
Entering directory entry_4
Leaving directory entry_4
-----------------------
-----------------------
Entering directory entry_5
Leaving directory entry_5
-----------------------
-----------------------
Entering directory entry_6
Leaving directory entry_6
-----------------------
-----------------------
Entering directory entry_7
Leaving directory entry_7
-----------------------
-----------------------
Entering directory entry_8
Leaving directory entry_8
-----------------------
-----------------------
Entering directory entry_9
Leaving directory entry_9
-----------------------
-----------------------
Entering directory entry_10
Leaving directory entry_10
-----------------------
Leaving directory /
-----------------------
Done testing the file system!!!
```

References

[1] The File system, 1996-1999 David A Rusling
http://www.tldp.org/LDP/tlk/fs/filesystem.html

[2] HEW (Highperformance Embedded Workshop)
http://www2.eu.renesas.com/products/mpumcu/tool/hew/documents.html

[3] HTTP - Hypertext Transfer Protocol *http://www.w3.org/Protocols/*

[4] M16C Family, Renesas
http://america.renesas.com/fmwk.jsp?cnt=m16c_family_landing.jsp&fp=/products/mpumcu/m16c_family/

[5] SFS: Simple Filesystem *http://www.eros-os.org/devel/ObRef/standard/SFS.html*

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