NetBurner Embedded Flash File System

Hardware and Software Guide

Document Number: 350080-003
Revision 1.2, Oct 17, 2007

TABLE OF CONTENTS

1 Introduction

1.1 How to Use This Guide

The guide covers the NetBurner implementation of the HCC Embedded Flash File System (EFFS) and NetBurner flash card hardware interface. The EFFS is just one part of the NetBurner suite of tools and software. You may need to reference other documents for configuration and hardware specifics for your particular NetBurner platform.

Topics Covered:

- Hardware interface design for MMC/SD and Compact Flash cards
- Basic EFFS example to mount a card, then read and write files
- FTP example to view, upload and download files
- HTTP example to illustrate how to incorporate files from the flash card into the web server
- Advanced programming topics

1.2 Source Code for Example Programs

Source code for all the examples is installed with the NNDK in the directory "\nburn\examples\". The latest manual revision, can be downloaded from http://www.netburner.com in the "support/technical_documents" section.

1.3 Supported Hardware Platforms

The EFFS is supported on the following NetBurner platforms:

- PK70
- Mod5234
- Mod5270
- Mod5272
- Mod5282
- SD/MMC Flash Cards: NNDK module development board revision 1.07 and higher for the above modules. You may be able to modify a revision 1.06 development board to be compatible with the SD wiring interface described below.
- Compact Flash Cards: NNDK module development boards revision 1.03 and higher will work with the NetBurner Flash Card Interface adapter. This is the same adapter used for the compact flash style WiFi cards.

The module development board has a SD/MMC card connector located between the module connectors. To use a compact flash card you will need to purchase a compact flash interface card that connects to J14 and J17. NetBurner WiFi compact flash interface cards will work for flash cards as well.

1.4 Hardware Setup

This document assumes you have installed the NetBurner Development Tools and are running on a NetBurner Network Development Kit platform. The examples will use the serial debug interface to display status messages from the running application. Once you are familiar with the EFFS, you should be able to run the same examples on your custom harware implementation.

NetBurner Development Board showing SD/MMC card interface (network module has been removed).

NetBurner Development Board showing Compact Flash card interface

1.5 Software Installation

NetBurner software and tools run on Windows NT 4.0, 2000, and XP. Insert the CD into the CD-ROM drive. The Autorun feature should automatically start the install. If it does not, run “setup.exe” from the CD-ROM.

1.6 Debug Port

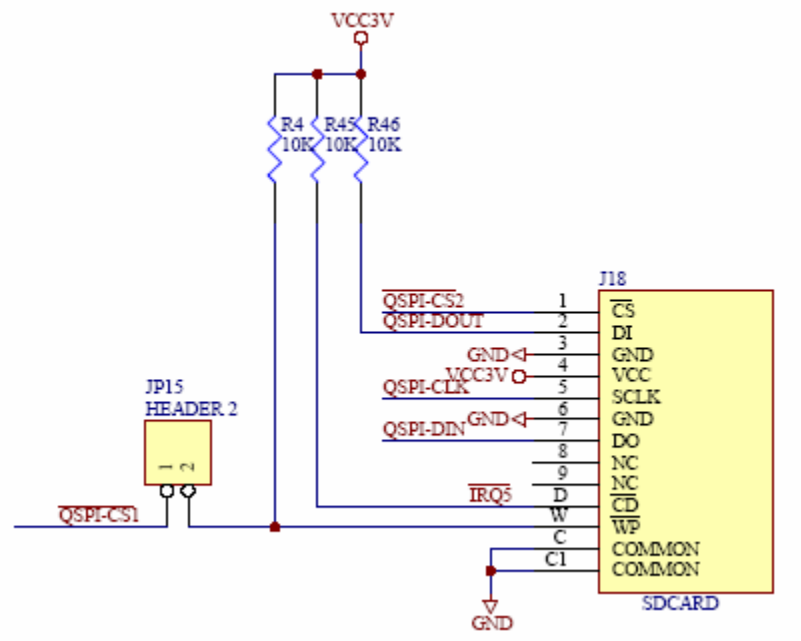
Throughout this guide, we will refer to the “debug port”. The debug port is one of the RS-232 ports that can be used to interact with your NetBurner device in the example programs. By default stdout, stdin and stderr are mapped to the debug port, so when you use functions like printf( ), scanf( ), gets( ), etc. they read and write to the debug port. All of this is configurable. You can also disable the debug port and use the port as a general purpose UART, or you can reassign the stdio file descriptors to use other serial or network interfaces.

2 Flash Card Hardware Interface Design

2.1 MMC/SD Hardware Interface

2.1.1 Schematic Representation

The SD/MMC flash card interface is installed on NetBurner Network Development Kits (NNDK) with development board revision 1.06 and higher. The schematic representation is shown below:



2.1.2 Signal Description

SD/MMC Pin	SD/MMC Signal Name	Modxxxx Signal Connection
1	/CS, Chip Select	J2-35 with 10k pull-up *
2	DI, Data In	J2-28 - QSPI DOUT
3	GND	Ground
4	Vcc	Vcc 3.3V
5	CLK	J2-25 - QSPI CLK
6	GND	Ground
7	DO, Data Out	J2-27 - QSPI DIN
8	NC, No Connection	No Connection
9	NC, No Connection	No Connection
D	CD, Card Detect	J2-47 - with 10k pull-up *
W	WP, Write Protect	J2-40 - with 10k pull-up *
Com	Common	Ground
Com	Common	Ground

* These signals can be any available GPIO pins on the NetBurner module, with a corresponding software change in mmc_mcf.cpp to specify the signal name.

2.1.3 SD/MMC Connector Part Number

The SD/MMC flash card connector used on the NetBurner development board is available from Mouser Electronics, part number 688-SCDA1A0901.

2.1.4 SD/MMC Card Compatibility

The SD/MMC must support native SPI mode transfers, which is common for most SD/MMC cards. Up to 2GB is supported for standard SD/MMC and up to 32GB is supported for SDHC.

2.1.5 Exclusive Use of the QSPI

The EFFS normally requires exclusive use of the QSPI interface. The standard software and drivers operate in this mode. There are also performance reasons for exclusive use. While it may be possible to share the QSPI with other peripherals, it is not supported in the development tool suite.

2.1.6 SD/MMC Interface Control Pins

You can modify the default pin selections for the interface control signals in \nburn\<platform>\system\mmc_mcf.cpp file. The relavent functions in this file are shown below. Note that these values must be used on the NetBurner development board since the board is wired in this configuration.

```
/*-------------------------------------------------------------------
 These following functions are used to configure which module
 GPIO pins are used for the SD/MMC flash card interface.
 The commented-out pins class calls at the top of the functions
 provide the same functionality as the sim register interface but will
 result in slightly lower performance due to added overhead.
 -------------------------------------------------------------------*/

/*-------------------------------------------------------------------
 Configure pins to be used for SD/MMC interface control to GPIO
 -------------------------------------------------------------------*/
void MMC_BaseInit()
{
  // J2[25].function(PINJ2_25_SPI_CLK);
  // J2[27].function(PINJ2_27_SPI_DIN);
  // J2[28].function(PINJ2_28_SPI_DOUT);
  // J2[35].function( PIN_GPIO );
  // J2[47].function( PIN_GPIO );
  // J2[40].function( PIN_GPIO );

#if defined FAT_CONF_5270

#ifdef PK70
   sim.gpio.pddr_qspi &= ~0x10;
   sim.gpio.par_qspi |= 0x1F;
   sim.gpio.par_qspi &= ~0xC0;
   sim.gpio.par_ad &= ~0x01;
```

```
   sim.eport.epddr &= ~0x30;
   sim.eport.eppar &= ~0x0F00;
   sim.gpio.pddr_datal |= 0x10;
#elif defined MOD5234
   sim.gpio.pddr_qspi &= ~0x10;
   sim.gpio.par_qspi |= 0x1F;
   sim.gpio.par_qspi &= ~0xC0;
   sim.gpio.pddr_etpu |= 0x01;
   sim.gpio.par_etpu &= ~0x01;
   sim.eport.epier &= ~0x20;
   sim.eport.epddr &= ~0x20;
   sim.eport.eppar &= ~0x0C00;
#else  //MOD5270 platform
   sim.gpio.pddr_qspi &= ~0x10;
   sim.gpio.par_qspi |= 0x1F;
   sim.gpio.par_qspi &= ~0xC0;
   sim.eport.epier &= ~0x20;
   sim.eport.epddr &= ~0x20;
   sim.eport.eppar &= ~0x0C00;
   sim.gpio.pddr_timer |= ~0x40;
   sim.gpio.par_timer &= ~0x3000;
#endif

#elif defined FAT_CONF_5282
    sim.gpio.ddrqs &= ~0x10;
    sim.gpio.ddrqs |= 0x20;
    sim.gpio.pqspar |= 0x07;
    sim.gpio.pqspar &= ~0x30;
    sim.eport.epddr &= ~0x20;
    sim.eport.eppar &= ~0x0C00;

#elif defined FAT_CONF_5272
    sim.paddr &= ~0x0800;
    sim.pacnt &= ~0x00C00000;
    sim.icr4 &= ~0x0F000000;
#endif
}

/*-------------------------------------------------------------------
  Get Card Detect state
  RETURN: 0 - card is removed
          1 - card present
 -------------------------------------------------------------------*/
int get_cd( void )
{
//   int rv = ( ( ( int )J2[47] ) == 0 );

#if defined FAT_CONF_5270
   int rv = !( sim.eport.eppdr & 0x20 );

#elif defined FAT_CONF_5282
   int rv = !( sim.eport.eppdr & 0x20 );

#elif defined FAT_CONF_5272
   int rv = ( ( sim.isr & 0x40 ) == 0 );
#endif

   return rv;
}
```

```
/*-----------------------------------------------------------------------
  Get Write Protect state
  RETURN: 0 - not protected
          1 - write protected
-----------------------------------------------------------------------*/
int get_wp( void )
{
//int rv = ( ( ( int )J2[40] ) == 1 );

#if defined FAT_CONF_5270
#ifdef PK70
   int rv = ( sim.eport.eppdr & 0x10 );
#else
   int rv = ( sim.gpio.ppdsdr_qspi & 0x10 );
#endif

#elif defined FAT_CONF_5282
   int rv = ( sim.gpio.portqsp & 0x10 );

#elif defined FAT_CONF_5272
   int rv = ( sim.padat & 0x0800 );
#endif

   return rv;
}

/*-----------------------------------------------------------------------
 Set SPI chip select low
 -----------------------------------------------------------------------*/
void spi_cs_lo( void )
{
  // J2[35] = 0;

#if defined FAT_CONF_5270
#ifdef PK70
   sim.gpio.pclrr_datal &= ~0x10;
#elif defined MOD5234
   sim.gpio.pclrr_etpu &= ~0x01;
#else
   sim.gpio.pclrr_timer = ~0x20;
#endif

#elif defined FAT_CONF_5282
   sim.gpio.clrqs = ~0x20;

#elif defined FAT_CONF_5272
   sim.pdcnt |= 0x30;
#endif
}

/*-----------------------------------------------------------------------
 Set SPI chip select high
 -----------------------------------------------------------------------*/
void spi_cs_hi( void )
{
// J2[35] = 1;

#if defined FAT_CONF_5270
#ifdef PK70
   sim.gpio.pclrr_datal |= 0x10;
#elif defined MOD5234
   sim.gpio.pclrr_etpu |= 0x01;
#else
```

```
   sim.gpio.ppdsdr_timer = 0x20;
#endif

#elif defined FAT_CONF_5282
   sim.gpio.portqsp = 0x20;

#elif defined FAT_CONF_5272
   sim.pdcnt &= ~(0x0030);   // QSPI_CS2 Hiz
#endif
}
```

2.1.7 Interrupt driven QSPI

You can enable the SD/MMC QSPI interface to work with the NetBurner interrupt driven QSPI driver instead of the default polling mode driver. This will result in slightly lower SD/MMC performace but overall higher system performance. This is useful if you find that your network performance or user tasks run poorly during file system accesses. A modification must be made in the file "\nburn\<platform>\system\mmc_mcf.cpp", followed by a rebuild of the platform system directory. Near to top of this file the following line should be uncommented:
//#define SD_IRQ_QSPI

2.2 Compact Flash Hardware Interface

While the SD/MMC interface is very simple, the Compact Flash Interface (CFI) implementation will require someone with experience in digital hardware design. The CFI uses the processor address bus, data bus, chip selects, bus control signals, and interrupts. The CFI is implemented by connecting directly to module signals, but also requires external logic. The external logic is implemented in a CPLD on NetBurner CF adapter boards.

The explanation and design considerations for each of these signals is beyond the scope of this document. However, experienced digital designers should be able to implement the interface based on the board and CPLD schematics included with the NetBurner CF adapter board. For development purposes, the NetBurner CF adapter board plugs into the module development board to facilitate software development.

3 Basic File System Operation

Typical file system operation will involve mounting a drive, opening and closing files, and reading and writing files. The following is a list of the most common function calls used to perform these operations. For a complete list of functions refer to the HCC Embedded Flash File System Implementation Guide included with your development kit documentation as a pdf file.

3.1 Common EFFS Function Calls

Create/delete working directory for current task priority:

```
int f_enterFS( void );
void f_releaseFS(void );
```

Mount/dismount a flash card:

```
int f_mountfat(MMC_DRV_NUM, mmc_initfunc, F_MMC_DRIVE0);
int f_delvolume(int drivenum)
```

Open/Close a file

```
F_FILE *f_open(const char *filename, const char *mode);
int         f_close(F_FILE *filehandle)
```

Read, write, and related functions:

```
int    f_getfreespace(int drivenum, F_SPACE *pspace)
long f_write(const void *buf, long size,long size_st, F_FILE *filehandle)
long f_read( void *buf, long size,long size_st, F_FILE *filehandle)
long f_seek(F_FILE *filehandle,long offset, long whence)
int    f_eof(F_FILE *filehandle)
int    f_rewind(F_FILE *filehandle)
int    f_delete(const char *filename)
```

Directory functions:

```
int f_findfirst(const char *filename, F_FIND *find)
int f_findnext(F_FIND *find)
int f_chdir(const char *dirname)
int f_mkdir(const char *dirname)
```

File time functions

```
int f_settimedate(const char *filename, unsigned short ctime, unsigned short cdate)
int f_gettimedate(const char *filename, unsigned short *pctime, unsigned short *pcdate)
```

3.2 File Time and Date Stamps

The EFFS supports file time and date stamps. There are a number of ways to obtain the current world time for an embedded system, including a Network Time Server (NTP), Real-time clock (RTC), and setting it manually. For simplicity the first two examples concentrate on file system calls. The third example includes methods to set the time and date through all the aforementioned methods. If no time or date is set, the file time stamp will be January 1, 1980.

3.3 File System Utils

All of the NetBurner EFFS examples include a helpful utility file called FileSystemUtils.cpp. This file provides an easy use interface for initializing, getting status, testing, reading and writing to a CF or SD/MMC card. To select between the types of cards, edit the header file cardtype.h. This file also demonstrates many of the commonly used EFFS function calls. The examples which include FileSystemUtils.cpp can be found here: "…\nburn\examples\EFFS".

3.4 Basic EFFS Example Program

The following is a very basic example that uses the file system to read, write and detect errors. It can be used for either SD/MMC cards, or compact flash cards by modifying the header file cardtype.h. The source for the NetBurner EFFS FTP example has the default installation location: "\Nburn\examples\EFFS\EFFS-BASIC"

4 FTP File System Operation

The source for the NetBurner EFFS FTP example has the default installation location: "\Nburn\examples\EFFS\EFFS-FTP". The NetBurner FTP server can be combined with the EFFS file system to enable FTP upload and download capability to the external flash card. To accomplish this we need to make the following changes:

1. Add the FTP #include directives.
2. Add a working directory to the EFFS for the FTP task priority with f_enterFS();
3. Add a function call in main.cpp to start the FTP Server.
4. Delete the function calls in the previous example to disconnect the flash drive.
5. Add the required file system functionality to the FTP callback functions. This has been done in the file named ftp_f.cpp and ftp_f.h. For details on how to implement the FTP callback functions, please view the included ftp_f.cpp file.

4.1 Start the FTP Server

The main.cpp is identical to the previous Basic example with the following additions:

Add the FTP include directives:

```
#include <ftpd.h>
#include "ftp_f.h"
```

Add a working directory to the EFFS for the FTP task priority:

```
OSChangePrio( FTP_PRIO );
f_enterFS();
OSChangePrio( MAIN_PRIO );
```

Add the function call to start the FTP Server in UserMain( ):

```
// Start FTP server with task priority higher than UserMain(),
// FTP_PRIO is the NNDK default value for FTP task priority and
// can be found in constants.h

int status = FTPDStart( 21, FTP_PRIO );
```

4.2 Running the FTP Example

Once you have the example downloaded and running, you can use Internet Explorer to drag and drop files to the flash card. Note: at the time this application note was written, browsors such as firefox did not support the drag and drop FTP feature. Once Internet Explorer is open, use the FTP URL: ftp://<ip address>. For example, ftp://10.1.1.2. You can also use other FTP client programs.

Note that in order for NTP to work, you must have access to a NTP server either on your LAN or through an Internet connection. You need to make sure you have a valid IP address, mask, and gateway address. The gateway address is required for Internet communication. You can set it through the IPSetup program.

5 HTTP and FTP File System Operation

The source for the NetBurner EFFS FTP example has a default installation location of: "\Nburn\examples\EFFS\EFFS-HTTP". This example combines the HTTP and FTP operation so that you can FTP files to the flash card and have them displayed by the web server. When HTML files exist on both the module flash memory and external flash card flash memory, the web server will look at the external flash card first, then look to internal flash. For example, if you have an index.htm in internal flash and not on the flash card, the index.htm file from internal flash will be displayed. If you have an index.htm file located in both internal flash and in the external flash card, then the index.htm from the external flash card will be loaded. The EFFS-HTTP example illustrates this case.

To implement the web server support the following changes need to be made to the FTP example program:

1. Add a working directory to the EFFS for the HTTP task priority with f_enterFS();
2. Add a function call to start the web server in UserMain( )
3. Add the web server functions to select which files are sent to the web browser when a GET request is received. This has been done in the web.cpp file.
4. Add the ability to set the system time through NTP, RTC, or manually. The source code for these functions is located in effs_time.cpp and effs_time.h.

5.1 Start the Web Server and Set the System Time

The main.cpp is identical to the previous example with the following additions:

Add a working directory to the EFFS for the HTTP task priority:

```
OSChangePrio( HTTP_PRIO );
f_enterFS();
OSChangePrio( MAIN_PRIO );
```

Add a function call to start the web server in UserMain() :

```
StartHTTP();
```

5.2 Add Web Page Processing

In addition to the file web.cpp, the following file must be included in main.cpp: `#include "http_f.h"`

The function MyDoGet( ) handles the GET request from a web browser and selects the appropriate file. This file contains many debug statements that are displayed through the serial debug port. After looking at this code you can monitor the port during runtime and see how the decisions are made.

5.3 Running the EFFS HTTP Demo

1. Delete any .htm or .html files on your flash card. You can do this easily with the Internet Explorer FTP client capability if you wish.
2. Compile the EFFS HTTP demo program, download it to your NetBurner device, and verify it is running correctly by monitoring the status messages from the serial debug port and MTTTY.

3. There will be a wealth of information available through the serial debug port, telling you what the application is doing at each step, and interactively when a web browser requests information from the web server.
4. Open a web browser and connect to the NetBurner web browser. You should see the index.htm web page from internal flash memory.
5. Use Internet Explorer and connect, via FTP, to the NetBurner device. Drag and drop the index.htm file in the project source code directory (not the project html directory) so that it is uploaded to the flash card.
6. Refresh the web browser page. You should now be viewing the index.htm from the flash card.

Note that in order for NTP to work, you must have access to a NTP server either on your LAN or through an Internet connection. You need to make sure you have a valid IP address, mask, and gateway address. The gateway address is required for Internet communication. You can set it through the IPSetup program.

5.4 Dynamic Web Page Content

HTML dynamic web page content is implemented with the FUNCTIONCALL HTML tag. In order for this method to work properly all functions called with the FUNCTIONCALL tag must be linked at the time the application is built. What this comes down to is that dynamic HTML pages must be implemented in the flash memory on the NetBurner module – you cannot create these types of dynamic HTML pages and run them from an external flash card. However, the dynamic web page can use and link to files on the flash card, so if you need additional memory for images, they can be stored on the flash card and used by an internal web page. Since links can be created dynamically, all the files do not need to be known at compile time. This does not affect dynamic web pages created in Java, since they do not use the HTML function call tag.

5.5 Long File Names

The EFFS supports the 8.3 format by default, and optionally supports long file names. The long filename is optional because of the increase in system resources required to do long filenames. In particular the stack sizes of applications which call the file system must be increased and the amount of checking required is increased. The maximum long filename space required by the standard is 260 bytes. As a consequence each time a long filename is processed large areas of memory must be available.

To switch from the 8.3 format to long file names:

1. Edit \nburn\include\constants.h and increase the user task stack size to a minimum of 8096.
   ```
   #define USER_TASK_STK_SIZE (8096)
   ```
2. Run the batch file: "\nburn\pcbin\longfilenames.bat". This will switch the EFFS library to the long file name version.
3. Rebuild the system library. From DevC++, select "Build -> Rebuild All".

On December 3rd 2003 Microsoft announced that it would exercise its patent rights relating to certain elements of how long filenames are implemented in FAT file systems. As a consequence it is up to the user to contact Microsoft to get the required licenses should they use the long filename option. It is also possible that some flash card manufacturers may obtain rights to use long file names on their media. You will need to check with your flash card vendor to determine if this is the case.

6 Using the EFFS-STD File System with On-chip Flash

6.1 EFFS-FAT Vs. EFFS-STD

The previoius sections of this guide have focused on the EFFS-FAT file system, which is a FAT32 file system used for external flash cards. The EFFS-STD file system uses the on-chip flash memory of your NetBurner device. It has a different set of libraries and functions than the EFFS-FAT file system. In most cases the differences between the EFFS-STD and EFFS-FAT are simply a change in functioncall prefix from f_ to fs_. For example, the EFFS-FAT function call f_open( ) is fs_open( ) for EFFS-STD.

6.2 Overview

When using the on-chip flash you need to be aware that the flash will be shared between your application and the file system. You will need to specify the amount of space to be used by the file system, while making sure you leave enough room for your application. The amount of flash used by your application is displayed each time you compile. You want to use the compressed number, not the uncompressed. You certainly want to leave enough additional space so your application can grow.

The flash memory chip will be divided into sectors, typically 4k or 64k bytes in size. The EFFS-STD file system requires that you allocate a number of these sectors to be used by the file system. Please review the data sheet for the flash memory used on your NetBurner device so you are familiar with the architecture. The configuration settings are dependant on the specific flash chip you are using.

Warning: incorrect settings can erase the NetBurner boot monitor!

6.3 Example for a 2MB Flash Chip

The source code for the example covered in this document is located in the c:\nburn\examples\EFFS\EFFS-STD-AM29LV160B directory. The example platform used is a Mod5234-100CR. There is also an example using a SB70 with 512k of flash memory located in c:\nburn\examples\parallax.

6.3.1 Software

The following example is for the AMD/Spansion 2MB flash chip, part number AM29LV160DB-90EF / S29AL016D-90TFI-020. This flash chip is used in the Mod5234, Mod5272, SB72EX and CB34EX. The example will do the following:

- Configure the EFFS-STD file system to match the sector size of the flash chip (AM29LV160B.C)
- Initialize the file system (fs_main.c)
- Display the current size of the file system and perform a read/write test.

6.3.2 Project Settings

To create a project that uses the EFFS-STD file system, you will need to make the following changes in NBEclipse:

- Modify the COMPCODEFLAGS to match the new memory space of the **application**. The remaining space will be used for the file system. The format of the COMPCODEFLAGS setting is: COMPCODEFLAGS <start address> <end address>.

- Add the EFFS-STD library to the linker path: c:\nburn\lib\StdFFile.a. The example will use NBEclipse. If you are using the command line tools, you will need to add the following line to your makefile: LDEXTRA=/nburn/lib/StdFFile.a.

The comments in the AM29LV160B.c file in the following section describe how to make the NBEclipse changes.

6.3.3 Flash Memory Addresses

The flash memory on your NetBurner device is used for the Boot Monitor, System Parameter Storage, User Parameter Storage, Application, and now the EFFS-STD file system. A table of memory sizes for NetBurner platforms at the time of this writing is shown below. The example column illustrates one possible configuration. You can modify the parameters to suit your requirements. The COMPCODE flag starting address specifies the starting memory location of your **application**. The end address specifies the end location of the **application**. You should not modify the starting address! The Boot Monitor, System Parameters and User Parameters occupy the space between the start of flash memory address and the start of the application memory address. You will only need to modify the end address to represent the amount of memory allocated for the flash file system.

Platform	**Total Size in Bytes**	**Start Address**	**End Address**	**Configuration Definition Example**

Mod5270
Mod5282
SB70
SB72
	512K	0xFFC00000	0xFFC40000	**Example for 256k flash file system**
				Application must begin at 0xFFC08000
				
				COMPCODEFLAGS = 0xFFC08000 0xFFC40000
				
				#define FLASH_SIZE (512*1024) // total flash size
				#define FS_SIZE (256*1024) // size of filesystem

Mod5234
Mod5272
SB72EX
CB34EX
	2MB	0xFFC00000	0x FFE00000	**Example for 512K flash file system**
				Application must begin at 0xFFC08000
				
				COMPCODEFLAGS = 0xFFC08000 0xFFD80000
				
				#define FLASH_SIZE (2*1024*1024) // total flash size
				#define FS_SIZE (512*1024) // size of filesystem

PK70	4MB	0xFF800000	0xFFC00000	**Example for 2MB flash file system**
				Application must begin at 0xFF830000
				
				COMPCODEFLAGS = 0xFF830000 0xFFA00000
				
				#define FLASH_SIZE (4*1024*1024) // total flash size
				#define FS_SIZE (2*1024*1024) // size of filesystem

6.3.4 Configuration File for the 2MB Flash

The configuration section of the file AM29LV160B.c is shown below. The file system will the specified number of 64k physical flash sectors for file storage and management.

```
/*-------------------------------------------------------------------
 * EFFS-STD configuration file for Spansion AM29LV160B Flash Chip.
 * This file is part of an example that allocats 512K of flash space
 * to the file system, and the rest to the application.
 *-------------------------------------------------------------------*/

#include "AM29LV160B.h"
#include "basictypes.h"
#include "bsp.h"


// Start of Flash memory base address
#define FS_FLASHBASE (0xffc00000)

/*
 * BLOCKSIZE
 * This defines the size of the blocks to be used in the file storage area.
 * This must be an erasable unit of the flash chip. All blocks in the file
 * storage area must be the same size. This maybe different from the DESCSIZE
 * where the flash chip has different size erasable units available.
 *
 * SECTORSIZE
 * This defines the sector size. Each block is divided into a number of sectors.
 * This number is the smallest usable unit in the system and thus represents the
 * minimum file storage area. For best usage of the flash blocks the sector size
 * should always be a power of 2. For more information see sector section below.
 *
 * SECTORPERBLOCK
 * This defines the number of sectors in a block. It must always be true that:
 * SECTORPERBLOCK = BLOCKSIZE/SECTORSIZE
 *
 *
 * The memory map below is for a Mod5234 with a 2MB bottom boot block flash.
 * This example will allocate 1.5MB for the application space, and 512KB for
 * the file system. There are a total of 31 64K blocks on the flash chip.
 *
 *                             Address
 *                            ----------
 *    ----------------------  FFE0 0000 (End of flash space)
 *    |  File System Data  |
 *    |  512K              |
 *    |  64K x 6 Blocks    |
 *    |--------------------|  FFDA 0000 (Start of File System Data)
 *    |  DESC BLOCK 0/1    |
 *    |  64K x 2 Blocks    |
 *    |--------------------|  FFD8 0000 (Start of File System)
 *    |                    |
 *    |   Application      |
 *    |   1.5MB            |
 *    |   64K x 23 Blocks  |
 *    |                    |
 *    |--------------------|
 *    |   32K Block        |
 *    |--------------------| FFC0 8000  (Start of application space)
 *    |  8K User Params    |
 *    |--------------------|
 *    |  8K System Params  |
```

```
 *    |--------------------|
 *    |  16K Monitor       |
 *    |--------------------| FFC0 0000 (Start of Flash space)
 *
 *
 * CHANGES TO COMPCODE FLAGS
 * In NBEclipse, or your command line makefile, change the following line
 * so the application will only occupy the specified application space.
 * The first parameter is the start of application space, and the second
 * is the address just below the file system space.
 *
 *    COMPCODEFLAGS = 0xFFC08000 0xFFD80000
 *
 * If using NBEclipse:
 * - Right-click on the project and select "Properties"
 * - Select "NetBurner" in the left side of the dialog box
 * - Verify the Platform is set to Mod5234, then check the "Use Custom Platform Settings"
checkbox
 * - Modify the "Compcode Memory Range" to the above values
 *
 *
 * If using NBEclipse, you will also need to tell the linker to include
 * the c:\nburn\lib\StdFFile.a library. To do this right-click on your
 * project, select properties, GNU Linker, then add the library.
 *
 */


/* WARNING: These settings are for AM29LV160DB / S29AL016D bottom boot block flash
 * components used on the Mod5234.
 */
#define BLOCKSIZE  ( 64*1024 )    // Use only the 64k sectors
#define SECTORSIZE ( 16*1024 )    // 4 sectors per BLOCK
#define SECTORPERBLOCK (BLOCKSIZE/SECTORSIZE)

/*
 * Specify the total amount of flash memory in the system, and the amount
 * allocated to be used by the file system (the rest is used by the
 * application.
 */
#define FLASH_SIZE ( 2*1024*1024 )   // size of total flash in the system, 2MB
#define FS_SIZE ( 512*1024 )         // amount allocated to filesystem, 512KB
#define FIRST_ADDR (FLASH_SIZE - FS_SIZE) // first file system address to use in the
flash
#define BLOCKSTART 2   // first block where file system data starts
                       // (first 2 blocks are DESCRIPTORS)

/*
 * Descriptor Blocks:
 * These blocks contain critical information about the file system, block allocation,
 * wear information and file/directory information. At least two descriptor blocks
 * must be included in the system, which can be erased independently. An optional
 * descriptor write cache may be configured which improves the performance of the
 * file system. Please refer to the EFFS-STD implementation guide for additional
 * information.
 */
#define DESCSIZE ( 8*1024 )        // size of one descriptor
#define DESCBLOCKSTART 0           // position of first descriptor
#define DESCBLOCKEND 1             // position of last descriptor
#define DESCCACHE 2048
```