

FlashFile v2.10

**A File System Library
&
Communications Protocol**

For

Atmel AVR Microcontrollers

and

**Secure Digital &
Multimedia Cards**

or

Compact Flash Cards

**Progressive Resources, LLC
4105 Vincennes Road
Indianapolis, IN 46268**

Progressive Resources LLC

Flash File

Table of Contents

Updates in version 2.10	4
Updates in version 2.0	4
Description	4
Overall Scheme	4
Implementation	5
<i>Secure Digital and Multimedia Cards</i>	5
<i>Compact Flash Cards</i>	6
File System	7
Memory Considerations	8
<i>CodeVision AVR Heap Setup</i>	8
<i>ImageCraft AVR Heap Setup</i>	8
<i>IAR EWAVR Heap Setup</i>	9
V2.10 Update	10
Timing Considerations	10
Disk Space Management	11
Directory Support	12
Real-Time Clock	12
OPTIONS.H Header File	13
<i>Control Block</i>	13
<i>Standard Library <Includes></i>	14
<i>FlashFile "Includes"</i>	14
FlashFile Function Calls	16
uchar initialize_media (void)	16
sint fquickformat (void)	17
void GetVolID (void)	17
void read_directory (void)	17
sint fget_file_info (uchar *NAME, ulong *F_SIZE, uchar *F_CREATE, uchar *F_MODIFY, uchar *F_ATTRIBUTE, uint *F_CLUS_START)	18
sint fget_file_infoc (uchar flash *NAME, ulong *F_SIZE, uchar *F_CREATE, uchar *F_MODIFY, uchar *F_ATTRIBUTE, uint *F_CLUS_START)	18
sint chdir (schar *F_PATH)	18
sint chdirc (schar flash *F_PATH)	19
FILE * fopen (uchar *NAME, uchar MODE)	19
FILE * fopenc (uchar flash *NAMEC, uchar MODEC)	19
sint mkdir (schar *F_PATH)	20
*FILE fcreate (uchar *NAME, uchar MODE)	20
*FILE fcreatec (uchar flash *NAMEC, uchar MODE)	21
sint rmdir (schar *F_PATH)	21
sint remove (uchar *NAME)	21
sint removec (uchar flash *NAMEC)	21
sint rename (uchar *NAME_OLD, uchar *NAME_NEW)	21
sint fclose (FILE *rp)	22

Progressive Resources LLC

Flash File

<i>sint</i> <i>fflush</i> (<i>FILE *rp</i>)	22
<i>sint</i> <i>ffreemem</i> (<i>FILE *rp</i>).....	22
<i>slong</i> <i>ftell</i> (<i>FILE *rp</i>).....	22
<i>sint</i> <i>fseek</i> (<i>FILE *rp</i> , <i>ulong off_set</i> , <i>uchar mode</i>)	23
<i>sint</i> <i>fgetc</i> (<i>FILE *rp</i>)	23
<i>sint</i> <i>fread</i> (<i>void *rd_buff</i> , <i>uint dat_size</i> , <i>uint num_items</i> , <i>FILE *rp</i>)	23
<i>schar</i> <i>*fgets</i> (<i>schar *buffer</i> , <i>sint n</i> , <i>FILE *rp</i>).....	23
<i>sint</i> <i>ungetc</i> (<i>uchar file_data</i> , <i>FILE *rp</i>)	24
<i>sint</i> <i>fputc</i> (<i>uchar file_data</i> , <i>FILE *rp</i>).....	24
<i>sint</i> <i>fwrite</i> (<i>void *wr_buff</i> , <i>uint dat_size</i> , <i>uint num_items</i> , <i>FILE *rp</i>).....	24
<i>sint</i> <i>fputs</i> (<i>uchar *file_data</i> , <i>FILE *rp</i>)	24
<i>sint</i> <i>fputcsc</i> (<i>uchar flash *file_data</i> , <i>FILE *rp</i>)	25
<i>void</i> <i>fprintf</i> (<i>FILE *rp</i> , <i>uchar flash *pstr</i> , ...)	25
<i>sint</i> <i>feof</i> (<i>FILE *rp</i>).....	25
Example FlashFile Application	25
<i>Including the FlashFile</i>	26
<i>Importing the Data</i>	26
<i>Full Example Code Listing</i>	28
Software License	32
Operating License	32
Liability Disclaimer	32

Progressive Resources LLC

Flash File

Progressive Resources LLC Flash File System

Updates in version 2.10

Version 2.10 represents a shift in addressing scheme and code efficiency. The full revision history with bug fixes can be found at the top of the “file_sys.c” file. Be sure to read the documentation thoroughly especially concerning the follow points:

1. The directory listings addressing scheme for the Flash File has changed to use pointers to the Directory/File Entry to optimize code space.
2. The addition of the `_NO_MALLOC_` definable allows better allocation of memory and code space if malloc is no used elsewhere.

Updates in version 2.0

1. The addressing scheme for the Flash Media has changed from using direct addressing to sector addressing. This allows for less code since most of the calculations for addressing were being done using sector addressing anyway.
2. Unions and structures were used to get values to and from the Flash Media so the instructions used could be cut down from using bit shifting and bitwise AND's.
3. The “options.h” setup file has more handles included so that the user has more control over different features, including hard coding the number of bytes per sector of the Flash Media used.
4. `fread()` and `fwrite()` functions add to the Flash File system.

Description

This document describes the setup and usage of both the Secure Digital/Multimedia Card and the Compact Flash packages (sold separately) of the FlashFile. The FlashFile is a file system library for flash media, which reads and writes data in a FAT12/FAT16 format to Secure Digital/Multimedia cards or Compact Flash cards using the Atmel AVR processors. The fully functional file system takes up approximately 25kB in CodeVisionAVR (and IAR EWAVR, ~35kB in ImageCraftAVR Standard w/o code optimizations) of Programmable Flash, but the program flash usage can be lowered if not all functionality is needed (i.e. Read Only, No FAT12 Support, etc.). Use of a FAT12/FAT16 file format allows the user to read, write, append, create, or delete files on a flash media card (SD/MMC and/or Compact Flash) via an Atmel AVR processor with the ability to read, write, append, create, or delete the same files using a PC card reader (and vise versa).

Note: Atmel DataFlash cards are NOT standard PC Multimedia cards and will NOT work with the FlashFile as is.

Overall Scheme

The FlashFile library functions by hooking up a standard FAT12/FAT16 formatted Secure Digital/Multimedia card to the SPI bus of the Atmel AVR Processor or Compact Flash card connected to I/O ports of the AVR Processor. When the file system is included in the user's project, Standard Input/Output C Functions used for file handling like `fopen`, `fclose`, `fgetc`, `fputc`, etc. are available to the Atmel AVR Processor to read from and write to files located on a SD/MMC or Compact Flash card.

Progressive Resources LLC

Flash File

When using SD/MMC media, the “sd_cmd.c” file interfaces with the processor and handles all of the “behind the scenes” interfacing between the SD/MMC card and the Atmel AVR processor. When using Compact Flash media, the “cf_cmd.c” file interfaces with the processor and handles all of the “behind the scenes” interfacing between the Compact Flash card and the Atmel AVR processor. The “file_sys.c” file is where all of the “C” file commands are located.

The 25kB in CodeVisionAVR (and IAR EWAVR, ~35kB in ImageCraftAVR Standard) of Programmable Flash that the library takes up is for fully functional read and write commands to the selected flash media, with both FAT12 and FAT16 translations. The amount of programmable flash used for the FlashFile library can be significantly decreased if support for FAT12 and/or writing to the card are not needed.

The FlashFile library makes use of the memory allocation functions in the Standard Library (stdlib.h) when handling files. Only **CodeVisionAVR v1.24.0** or later supports the memory allocation functions and is required to use the FlashFile library (without modification); all Versions of ICCAVR and IAR EWAVR support the allocation functions, however it was only tested on (CodeVisionAVR v1.25.6) ICCAVR v6.31A standard and IAR EWAVR v3.20c.

A complete example of how to use the FlashFile is located at the end of this document.

Implementation

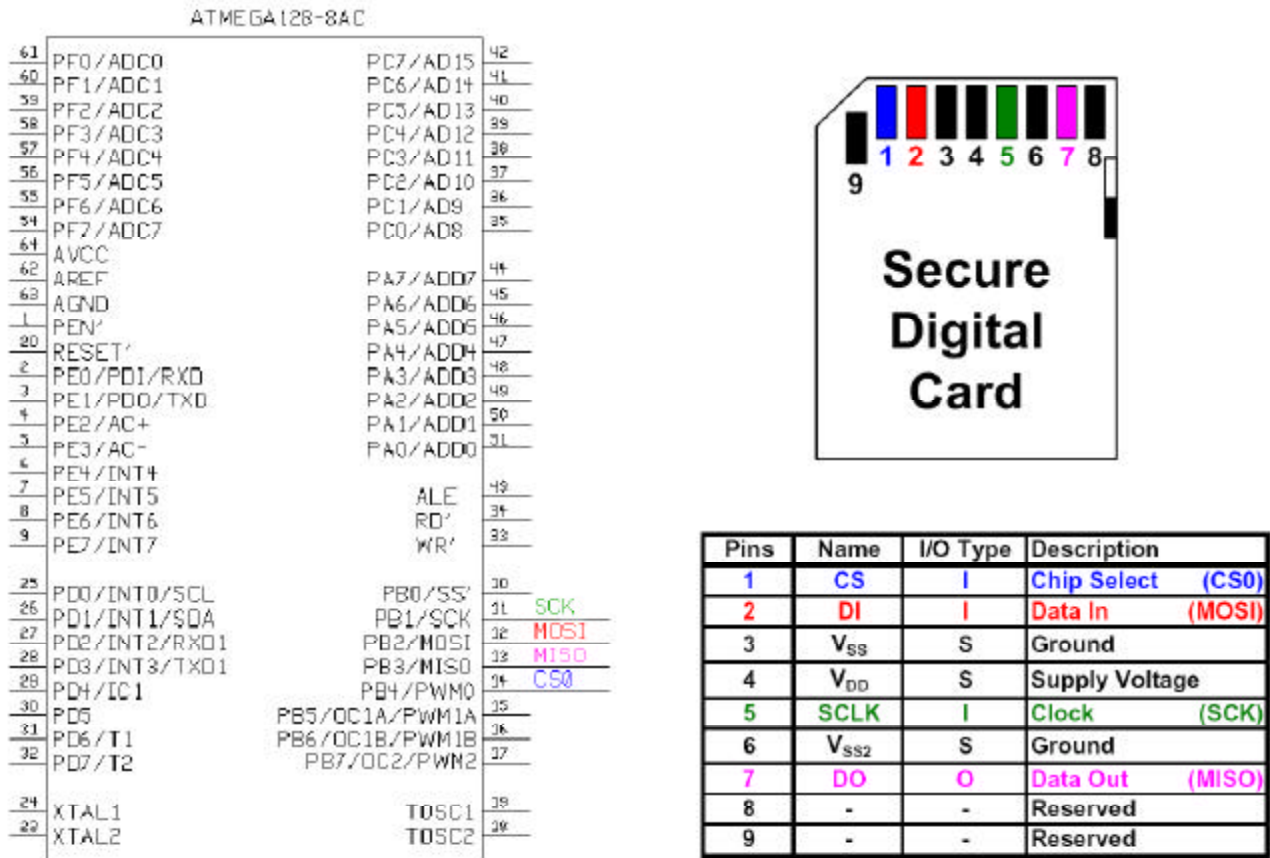
The FlashFile library is implemented by inserting source files into the code produced by the CodeVisionAVR wizard or written by a programmer (a complete example is presented in a later section). The library files are “sd_cmd.c” or “cf_cmd.c” (depending on media used), “file_sys.c”, “options.h”, (“twi.c” if using the real time clock option), and their respective header files. The “options.h” file should be included in the main project source file, and the other source files should be located in the directory specified in “options.h” (see the **OPTIONS.H Header File** section for details).

Secure Digital and Multimedia Cards

The Secure Digital card should be hooked up to the SPI bus of the Atmel AVR Processor with the Atmel AVR processor as the master. The test code included with the project assumes an Atmel ATMega128 (with 3.3V conversions if necessary) is used, with the MOSI, MISO, and SCLK pins of the SD card connected to the SPI bus of the Mega128. CS0 is the card select of the SD card, and could be changed to use any output pin desired (as long as it is properly defined by SD_CS_ON(), SD_CS_OFF(), and CS_DDR_SET() in the “options.h” file).

Progressive Resources LLC

Flash File



Hardware Diagram for an ATMEGA128 Interfaced with a Secure Digital Card

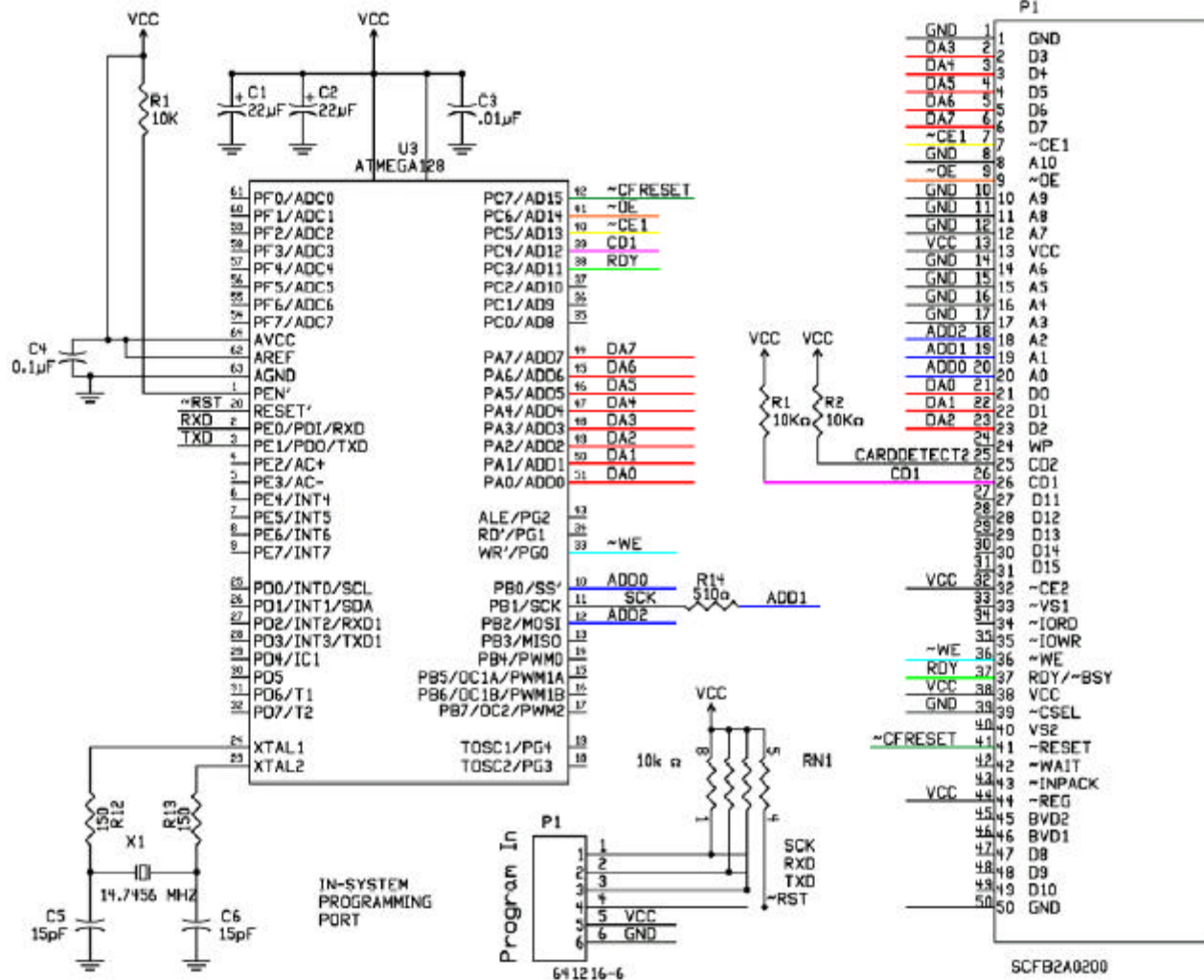
Note: Secure Digital Cards are rated at a Maximum of **3.6V**, all voltage signals to and from the card should be changed so that a “high” or a Logic “1” and the V_{dd} is 3.3V, and a “low” or a Logic “0” is 0V.

Compact Flash Cards

The Compact Flash card requires 17 connections from the CF socket to the Atmel AVR Processor. The data lines (DA0-DA7) connect to a single port of the Atmel AVR processor, while the address lines (ADDR0-ADDR2) are connected to the lower 3 bits of a port (use of the other lines in the port require bit addressing or proper masking). The test code included with the project assumes an Atmel ATMEGA128 is used with the following configuration:

- PORTA connected to the CF data lines
- PORTB.0-2 connected to the CF address lines
- PORTC.7 connected to the CF Reset line
- PORTC.6 connected to the CF OE (output enable) line
- PORTC.5 (or GND) connected to the CF CE1 (Card Enable 1) line
- PORTC.4 connected to the CF CD1 (Card Detect 1) line
- PORTC.3 connected to the CF RDY (Ready/Not Busy) line
- PORTG.0 connected to the CF WE (Write Enable) line

Flash File



Hardware Diagram for an ATmega128 Interfaced with a Compact Flash Card

File System

The FlashFile is setup so that it can read and write to either FAT12 or FAT16 formatted flash media. Most flash cards 64MB and lower come preformatted in FAT12, although if formatted on the PC, all cards 32MB and above are formatted in FAT16. FAT12 support is only needed for reading/writing to flash cards that are less than 32MB (must check format if 64MB or less); anything larger uses FAT16. Media larger than 2 GB must be formatted in FAT32, which is currently not supported by the FlashFile library (SD/MMC cards are currently only available up to 1GB, while Compact Flash cards are currently only available up to 2GB). If partitions are used, the FlashFile will assume the first partition will be used to read and write files and folders. Filenames used in the FlashFile must be in the 8.3 format, which mean up to 8 characters (no spaces or symbols other than '-' and '_') and the 3 character file type. If a file that needs to be accessed was previously saved in the long file name format, the 8.3 filename format must be used to access the card with the Atmel AVR processor (i.e. if a file is called "Long Filename.txt" in the PC, the AVR processor will see it as "LONGFI~1.TXT").

Progressive Resources LLC

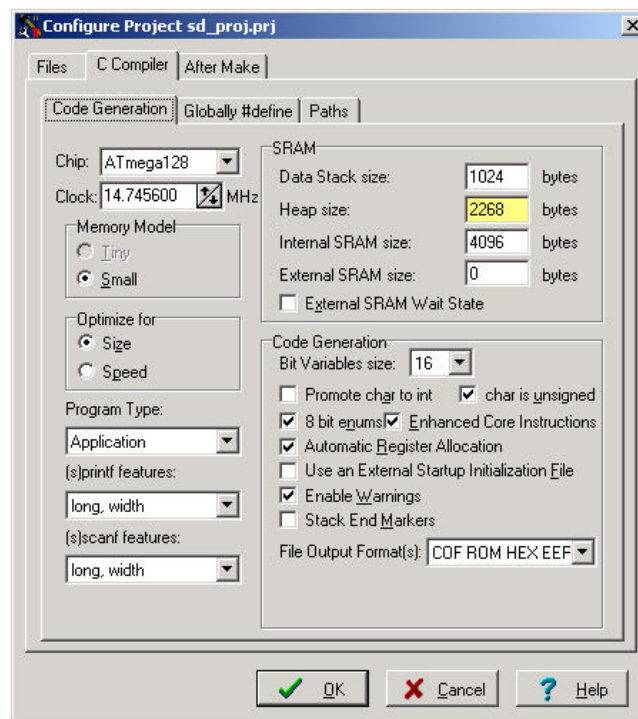
Flash File

Memory Considerations

The Secure Digital cards must be read and written in blocks of 512 bytes. This means that a 512 byte buffer must be held for reading and writing to the card at all times. All global variables (including the SD buffer) take up approximately 630 bytes and the data stack takes up approximately 128 bytes of SRAM (for IAR). The Heap size of the file system is variable, based on the number of files that need to be open simultaneously. Each file that is open requires 560 bytes in the heap (512 bytes for the FILE data buffer, 40 bytes for the needed FILE structure information, + 8 bytes for the FILE pointer overhead). This means that if only one file needs to be opened at a time, the Heap size must be at least 560 bytes, two open files at once requires at least a 1120 byte Heap size, three open files at once requires at least a 1680 byte Heap size, etc. Heap sizes larger than the required amount requires more SRAM use, but gives the user a margin of error to deal with when allocating memory space.

CodeVision AVR Heap Setup

To set up the Heap size in CodeVisionAVR v1.24 or later (earlier versions do not support memory allocation), from the main menu select “*Project => Configure*” and select the “*C Compiler => Code Generation*” tabs. Under “*SRAM => Heap size*”, enter the Heap size the project requires:



CodeVisionAVR Project Configuration Menu

ImageCraft AVR Heap Setup

To set up the Heap size in ImageCraft AVR, the “void _NewHeap(void *start, void *end)” function must be called when initializing the AVR processor (before the while(1) in the main() function). The *start variable is a pointer address of where the heap starts, and the *end variable is a pointer address of where the heap ends. The global library variable extern

Progressive Resources LLC

Flash File

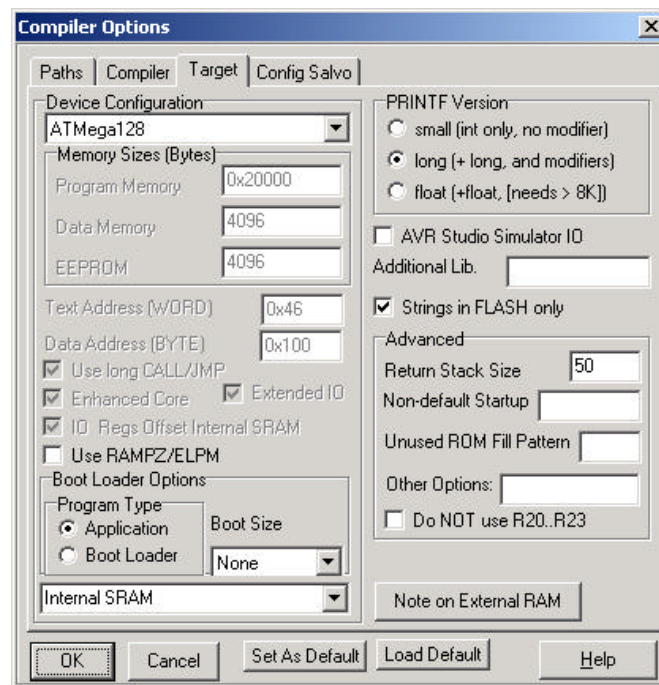
`char _bss_end` holds the SRAM address of the last used global variable. A common starting address for the heap is `&_bss_end + 1`, with the ending address for the heap at `&_bss_end + _heap_size_ + 1`.

Example:

To declare a heap size of 2048 bytes (2kB):

```
extern char _bss_end;
main()
{
    init_devices();
    _NewHeap(&_bss_end + 1, &_bss_end + 2048 + 1);
    // remaining code typed below
}
```

ImageCraft AVR automatically saves constant strings (i.e in a printf statement “Hello World”) to both the SRAM and Flash space. Since the heap is required to be at least 560 bytes, be sure to select “Strings in FLASH only” and use “cprintf” instead of “printf” if applicable (the “options.h” file #defines printf as cprintf for where they are required in code) in the “Compiler Options => Target” window. Since Long variables are used also in “read_directory” and debugging, select “long (+ long, and modifiers)” under “FPRINTF Version”. The “Return Stack Size” must be set to at least 30 to deal with all of the nested functions in the FlashFileSD.



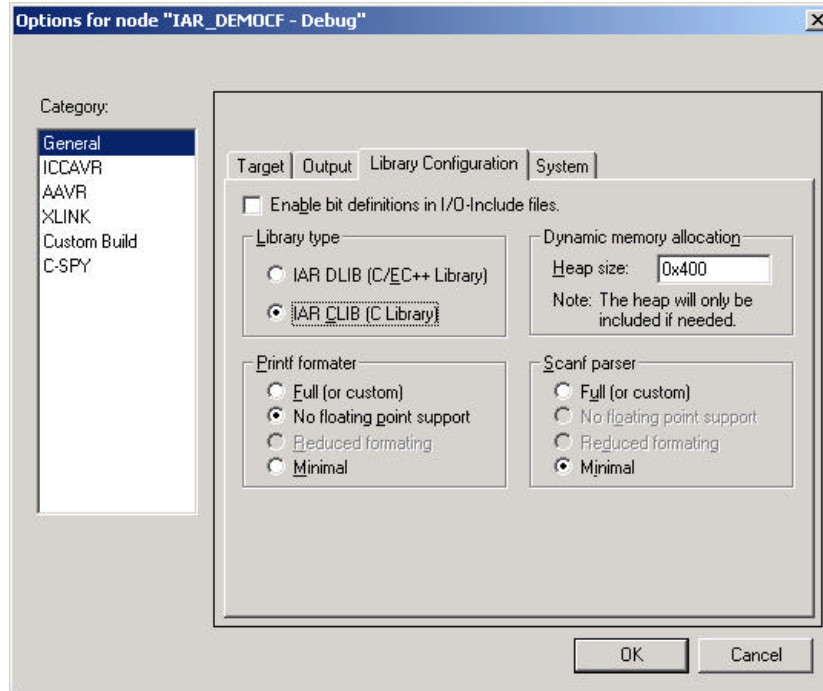
ICC Compiler Options

IAR EWAVR Heap Setup

To set up the Heap size in IAR EWAVR from the main menu select “Project => Options” and select the “General => Library Configuration” tabs. Under “Heap size”, enter the Heap size the project requires:

Progressive Resources LLC

Flash File



IAR EWAVR Project Configuration Menu

IAR EWAVR also automatically saves constant strings (i.e in a printf statement “Hello World”) to both the SRAM and Flash space. Since the heap is required to be at least 560 bytes, be sure to deselect “Place string literals and constants in initialized SRAM” and use “printf_P” instead of “printf” if applicable (the “options.h” file #defines printf as printf_P for where they are required in code) in the “ICCAVR => Code” window. Since Long variables are used also in “read_directory” and debugging, select “Full (or custom)” or “No floating point support” under “General => Library Configuration => Printf formatter”. The “Data Stack Size” under “General => System” must be set to at least 256 if the printf() or fprintf() functions are used so that the internal data parsing function does not overflow the data stack.

V2.10 Update

An update to the FlashFile was added so that a malloc() function is no longer required, allowing the #define _NO_MALLOC_ to be in the “options.h” file and specifying the number of files to be opened at once will allocate that space to global memory.

Timing Considerations

The FlashFile is setup so that all files are read and written to using a 512 byte SRAM buffer for each file. Because both the Secure Digital/Multimedia cards and Compact Flash cards must be read and written to using these 512 byte buffers, there may be short delays while reading and writing the card data when program needs to fill and/or flush the buffer. If a large card is being used (512MB, 256MB, 128MB) and there is a large portion of it previously filled, the initial write (the first time a write to file is called) may take longer than usual due to the processor looking for an empty space. If the card being used has a lot of fragmented data, read and write operations may also take longer since the processor has to jump around to different parts of the card to read or write data. The best

Progressive Resources LLC

Flash File

method to reduce the fragmentation on a card is to copy all of the data to the PC, delete all of the data on the card, and then copy the data saved on the PC back to the card.

The file system used (i.e. FAT12/FAT16) and the size of media used (i.e. 32MB, 64MB, 128MB, etc.) in the project also determines the speed that the data transfer works at. A FAT12 system usually uses larger cluster sizes, which means that the FAT table does not have to be updated as often as a FAT16 system.

Disk Space Management

The FlashFile uses the FAT12/16 specifications for files, which means that the disk space used is dependant on the cluster size of the file. The total disk space available depends on the size of media used. Each file takes up disk space in the same manner that a PC would, the disk space is equal to $n * \text{cluster size}$ (where n is the size of the file divided by the cluster size, + 1 (if any partial cluster is used, else 0)).

example:

GIVEN:

sector size = 512 bytes (most flash media)

cluster size = 8 sectors = 4096 bytes (dependant on FAT type and media size)

file1 size = 1,000 bytes

file2 size = 12,288 bytes

file3 size = 100,000,000 bytes

FILE1 SIZE (ON DISK):

$(\text{file1 size}) \% (\text{cluster size}) = (1,000) \% (4096) = 1,000 \text{ (Not 0)}$

$\text{disk space used} = ((\text{file1 size}) / (\text{cluster size}) + 1) * (\text{cluster size})$
 $= ((1,000) / (4096) + 1) * (4096)$
 $= (0 + 1) * (4096)$
 $= 4096 \text{ bytes used on disk}$

FILE2 SIZE (ON DISK):

$(\text{file2 size}) \% (\text{cluster size}) = (12,288) \% (4096) = 0$

$\text{disk space used} = (\text{file2 size})$
 $= 12,288 \text{ bytes used on disk}$

FILE3 SIZE (ON DISK):

$(\text{file3 size}) \% (\text{cluster size}) = (100,000,000) \% (4096) = 256 \text{ (Not 0)}$

$\text{disk space used} = ((\text{file3 size}) / (\text{cluster size}) + 1) * (\text{cluster size})$
 $= ((100,000,000) / (4096) + 1) * (4096)$
 $= (24,414 + 1) * (4096)$
 $= 100,003,840 \text{ bytes used on disk}$

(NOTE: If the file size is exactly divisible by the cluster size, that file size is the same size that is used on the disk.)

This is how the file system MUST use disk space for it to be compatible with a PC system. Any other disk space management system would not allow you to view any of the data on a PC, that was written from the device.

Progressive Resources LLC

Flash File

Directory Support

The FlashFile is setup so that directories and sub-directories can be accessed when a file name or path is entered in a command. File commands (fopen, fcreate, remove, etc.) and directory commands (mkdir, chdir, rmdir, etc.) can be used to access files or folders by inputting the full path name or relative path name. A full path name string has a leading back slash ('\') to indicate that it is relative to the root directory; a relative path name string starts with a folder name, a ".." (indicating one directory back), or just the file name.

Note: When entering a backslash in a constant string, the backslash must be preceded by a backslash to prevent the compiler from interpreting the backslash as a control character (i.e. '\\ = 0x5C, "\\\" is = 0x5C, 0x5C).

Examples:

Given:

Folder Trees: ROOT\FOLDER1\FOLDER2\FOLDER3\
 ROOT\LEVEL1\LEVEL2\LEVEL3\
Current Directory: ROOT\FOLDER1\

Problem:

Create the file "TEST.TXT" in the ROOT\LEVEL2\ directory

Solution 1:

```
chdir("\\LEVEL1\\LEVEL2");        // change to the \LEVEL1\LEVEL2 directory
fcreate("TEST.TXT");            // create "TEST.TXT" in the current directory
```

Solution 2:

```
chdir("../LEVEL1\\LEVEL2");       // go back a folder, then change to LEVEL1\LEVEL2 directory
fcreate("TEST.TXT");            // create "TEST.TXT" in the current directory
```

Solution 3:

```
fcreate("\\LEVEL1\\LEVEL2\\TEST.TXT"); // create "TEST.TXT" in the directory \LEVEL1\LEVEL2\
```

The FlashFile has some limitations on directory usage; if used beyond these limitations unexpected results may occur.

- * Folder names must be accessed in the 8.3 Dos format, an 8 character name and a 3 character extension (always 3 spaces [0x20] for folders)

Real-Time Clock

The FlashFile is setup so that a real-time clock can be used to date/time stamp the creation and modification of files. The directory in which the file information is stored has the date/time information stored, and can be read/modified with a PC. The real-time clock option of the FlashFile can be turned off in the control block of the main header file. If the real-time clock is not in use, the date/time stamp for the file is incremented (so the user knows the file is newer than before) whenever the file is modified.

The "twi.c" file included with the FlashFile has all of the interface functions for a Philips PCF8563 real-time clock/calendar hooked up to the I²C bus of the Atmel AVR processor (see the Progressive Resources LLC Mega128.NET development board for hookup information). Using a different real-time clock requires replacing the real-time clock routines in the twi.c file. The real-time clock function **char rtc_get_timeNdate(char *hour, char *min, char *sec, char *date, char *month, int *year)** is required by the FlashFile to use the real-time clock option. A search in all files for "_RTC_ON_" will find where real-time clock functions are used in the files.

Progressive Resources LLC

Flash File

OPTIONS.H Header File

Control Block

All of the compiling options for the FlashFile project are defined in the “options.h” file of the project; all of the control and project compile options for the project. Since the FlashFile can take up a considerable amount of flash code space on the AVR processor, the compiling options enable the user to reduce the amount of flash code space used depending on the user’s needs. Section A of the “options.h” file lists the compile options for the file.

- * `_OPTIONS_H_` - this option is just coding so the file is not included more than once, and symbols are not redefined.
- * `_RTC_ON_` - this option enables the real-time clock options, time and date stamping files on creation and modification; if this feature is commented out, the time/date code on the file increments whenever modified.
- * `_SECOND_FAT_ON_` - this option allows the system to update both FAT tables when writing files; if not in use, the system wipes out the second FAT table when creating or modifying a file so the system does not conflict with the PC.
- * `_FAT12_ON_` - this option enables the FAT12 file system to be supported by the project; cards 64MB and under can come pre-formatted in FAT12, although all cards 32MB and above are formatted in FAT16 when formatted by a PC.
- * `_READ_ONLY_` - this option allows users the option to significantly reduce their code space by eliminating all writing functions to the Secure Digital cards; any attempts to write to the card will error out or be invalid.
- * `_DEBUG_ON_` - this option enables an output to the serial port so that the status of the card could be displayed in HyperTerminal. This also must be on to use the `read_directory()` and `GetVolID()` functions.
- * `_NO_MALLOC_` - this option is used if the user does not want to use a `malloc()` function. One FILE structure will automatically be allocated in global memory if this option is defined.
- * `_BytsPerSec_512_` - In all current flash media, there are 512 bytes per sector, and there really is no need to have it as a variable. Defining `_BytsPerSec_512_` will hard code all references to `BPB_BytsPerSec` as 0x200 or 512. This will cut down on code space and speed the functions up a bit since `<< 9` and `>> 9` can replace `* 512` and `/ 512`, and `& 0x1FF` can replace `% 512`.
- * `_CVAVR_`, `_ICCAVR_`, & `_IAR_EWAVR_` - Only ONE of these three options should be in the code, the other two should be commented out or unexpected results could occur. `_CVAVR_` if using the CodeVisionAVR IDE, `_ICCAVR_` if using the ImageCraftAVR IDE, and `_IAR_EWAVR_` if using the IAR Embedded Workbench AVR IDE.
- * `_LITTLE_ENDIAN_` & `_BIG_ENDIAN_` - Only ONE of these two options should be defined in the code based on the type processor used. This will determine how unions will be defined in the code. All AVR processors are `_LITTLE_ENDIAN_`, and this option is only designed for customer who wish to port the code to a different type of processor.
- * `_FF_MAX_FILES_OPEN` - This is the maximum number of files that can be opened at once (if `_NO_MALLOC_` is defined, if a `malloc()` function is used, this is ignored).

Progressive Resources LLC

Flash File

- * `_SD_MMC_MEDIA_` & `_CF_MEDIA_` - Only ONE of these two options should be included in the code, the other should be commented out. `_SD_MMC_MEDIA_` should be included if a Secure Digital or Multimedia Card is used, or `_CF_MEDIA_` should be included if Compact Flash card is used.
- * `_MEGA128DEV_` - this option sets up the SD/MMC SPI bus to a PRLLC MEGA128Dev board (else the SPI bus is setup for a PRLLC MEGA128.NET board)
- * `SD_CS_ON()`, `SD_CS_OFF()`, & `CS_DDR` (For SD/MMC cards only) - These are macros defined to be the I/O pin of the Atmel AVR processor that the chip select line of the SPI bus is connected to, so the microcontroller can control the SD card (mask bits to your needs).
- * `cf_data_out`, `cf_data_in`, & `cf_data_ddr` (For Compact Flash cards only) - These are macros defined for the data port of the CF card.
- * `cf_addrport` (For Compact Flash cards only) - This is the macro for the port connected to the address port of the CF card.
- * `CD1` & `RDY` (For Compact Flash cards only) - These are macros defined for bit inputs to the “Card Detect 1” and “Ready” signals of the CF card.
- * `RESET_LO()` & `RESET_HI()`, `WE_LO()` & `WE_HI()`, and `OE_LO()` & `OE_HI()` (For Compact Flash cards only) - These are macros to turn the “Reset”, “Write Enable”, and “Output Enable” signals to the CF card “HI” and “LO”.
- * `_FF_MAX_FPRINT` - this determines the maximum length that a `fprintf()` string can be; the longer the string, the more that can be written in one line of an `fprintf()` command, however the memory requirements for the program will also be larger.
- * `_FF_PATH_LENGTH` - this determines how long the full path (directory) string can be.

Standard Library <Includes>

Some of the functions in the FlashFileSD require the use of several standard C libraries. Section B of the “options.h” file includes the libraries required and #defines needed in some of the functions. If using ICC or IAR, and “printf”, <stdio.h> must be included before “options.h” or “printf” will be redefined. (See “Memory Considerations” for use of printf in ICC or IAR if applicable)

FlashFile “Includes”

The actual FlashFile files “sd_cmd.h”, “sd_cmd.c”, “file_sys.h”, “file_sys.c”, “twi.h” (if applicable), and “twi.c” (if applicable) must also be included to use the file system functions. Section C of the “options.h” file does this automatically, based on the path inserted. The default path assumes that the files are located one directory back of the project folder, then in a folder called “Flash”. If the file are located in a different directory (with respect to the project directory) the paths for each #include must be changed to that path.

```
#ifndef _OPTIONS_H_
#define _OPTIONS_H_
// Control Block START//
#define _RTC_ON_
#define _SECOND_FAT_ON_
#define _FAT12_ON_
// #define _READ_ONLY_
#define _DEBUG_ON_
#define _DIRECTORIES_SUPPORTED_
#define _NO_MALLOC_
#define _BytsPerSec_512_
#define _LITTLE_ENDIAN_
```

A

Progressive Resources LLC

Flash File

```
// #define _BIG_ENDIAN_

// The settings below should be modified
// to match your hardware/software settings
#define _CVAVR_
// #define _ICCAVR_
// #define _IAR_EAVR_

#define _SD_MMC_MEDIA_
// #define _CF_MEDIA_

// #define _MEGA128DEV_
// #define _MEGA AVRDEV_

#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define schar signed char
#define sint signed int
#define slong signed long
// #define SPI_RETRY SPCR

#ifdef _NO_MALLOC_
#define _FF_MAX_FILES_OPEN 1
#endif

#ifdef _SD_MMC_MEDIA_
// #define _ALL_SD_CMDS_
#ifdef _MEGA128DEV_
#define SD_CS_OFF() PORTD |= 0x40
#define SD_CS_ON() PORTD &= 0xBF
#define CS_DDR_SET() DDRD |= 0x40
#else
#ifdef _MEGA AVRDEV_
#define SD_CS_OFF() PORTB |= 0x10
#define SD_CS_ON() PORTB &= 0xEF
#define CS_DDR_SET() DDRB |= 0x10
#else
#define SD_CS_OFF() PORTB |= 0x10
#define SD_CS_ON() PORTB &= 0xEF
#define CS_DDR_SET() DDRB |= 0x10
#endif
#endif
#endif
#ifdef _CF_MEDIA_
#define cf_data_out PORTA
#define cf_data_in PINA
#define cf_data_ddr DDRA

#define cf_addrport PORTB

#define CD1 (PINC & 0x10)
#define RDY (PINC & 0x08)

#define RESET_LO() PORTC &= 0x7F
#define RESET_HI() PORTC |= 0x80
#define WE_LO() PORTG &= 0xFE
#define WE_HI() PORTG |= 0x01
#define OE_LO() PORTC &= 0xBF
#define OE_HI() PORTC |= 0x40
#endif

#define _FF_MAX_FPRINTF 50
#define _FF_PATH_LENGTH 50
// Control Block END

#ifdef _DEBUG_ON_
#include <stdio.h>
#endif
#ifdef _CVAVR_
#ifdef _MEGA AVRDEV_
```

Progressive Resources LLC

Flash File

B

```
#define _AVR_LIB_ <mega32.h>
#else
#define _AVR_LIB_ <mega128.h>
#endif
#define CLI()      #asm("cli")
#define SEI()      #asm("sei")
#endif
#ifdef _ICCAVR_
#include <macros.h>
#define _AVR_LIB_ <iom128v.h>
#endif
#ifdef _IAR_EWAVR_
#include <pgmspace.h>
#define _AVR_LIB_ <iom128.h>
#define CLI()      asm("cli")
#define SEI()      asm("sei")
#define flash      __farflash
#endif
#include _AVR_LIB_
#include <stdarg.h>
#ifndef _NO_MALLOC_
#include <stdlib.h>
#endif
#include <ctype.h>
#include <string.h>

#ifdef _SD_MMC_MEDIA_
#include "..\flash\sd_cmd.h"
#else
#ifdef _CF_MEDIA_
#include "..\flash\cf_cmd.h"
#endif
#endif
#include "..\flash\file_sys.h"
#ifdef _RTC_ON_
#include "..\flash\twi.h"
#endif
#ifdef _SD_MMC_MEDIA_
#include "..\flash\sd_cmd.c"
#else
#ifdef _CF_MEDIA_
#include "..\flash\cf_cmd.c"
#endif
#endif
#include "..\flash\file_sys.c"
#ifdef _RTC_ON_
#include "..\flash\twi.c"
#endif
#endif
```

C

“OPTIONS.H” code listing

FlashFile Function Calls

The functions below are provided by the FlashFile library and are included in the “*sd_cmd.c*” or “*cf_cmd.c*” file (depending on the media used), and the “*file_sys.c*” files. These functions are used to communicate to the Secure Digital card, and organize the data in the files in a FAT12/16 format. All function explanations and examples assume the “*options.h*” file has already been included and variable type declarations have been defined as above.

uchar initialize_media(void)

This function is required to initialize the Flash card so reading and writing to the card is possible. This function also reads the card’s partition table and boot sector, storing the card

Progressive Resources LLC

Flash File

information needed to handle the file system. No data can be read from or written to the card without this function being run first. This function returns a *1* if the card is successfully initialized, and a *0* if the initialization fails.

Example:

```
if (initialize_media())      // If the media is initialized ok,
    putsf("OK");             // output "OK"
else                         // Otherwise
    putsf("ERROR");          // output "ERROR"
```

sint **quickformat(void)**

This function is used to do a quick format of the Flash media. When called, all data on the current media will be destroyed. If the function is successful a '0' is returned, if the function fails an EOF is returned.

Example:

```
if (initialize_media())      // If the media is initialized ok,
    quickformat();           // Delete all information on the card
```

void **GetVolID(void)**

This function prints the Flash media's serial number and volume label to the serial port (UART/USART must be properly set up) of the processor. (Note: The volume serial number and label can also be accessed through the global variables located in the "sd_cmd.c" file once "initialize_media()" is called. The serial number is stored as the unsigned long *BP_VolID*, and the volume label is stored in the character string *BP_VolLabel[12]*)

Example:

```
if (initialize_media())      // If the media is initialized ok,
    GetVolID();              // Send Volume information of card to serial port
```

```
Terminal Output
Volume Serial: [0x8D8293D]
Volume Label:  [SD64CARD  ]
```

void **read_directory(void)**

This function outputs the list of files located in the current directory of the Flash card to the serial port (UART/USART must be properly set up) so the user can see what files (and their size in bytes) are available for reading and writing to the card. This function also shows the working directory string of the project relative to the root directory. This function uses the printf defined in the Standard Input/Output Library, and may take up a large portion of code space if not used anywhere else. This function does not return any information other than the data output to the serial port.

Example:

```
if (initialize_media())      // If the media is initialized ok,
    read_directory();        // Send file list of card to serial port
```

```
Terminal Output (Assume TEST is a directory, TEST1.TXT and TEST2.TXT are files):
File Listing for:  ROOT\
[TEST]
TEST1.TXT          [131289] bytes
TEST2.TXT          [229327] bytes
```

Progressive Resources LLC

Flash File

sint fget_file_info(uchar *NAME, ulong *F_SIZE, uchar *F_CREATE, uchar *F_MODIFY, uchar *F_ATTRIBUTE, uint *F_CLUS_START)

This function stores the information of the file designated by the character string *NAME, into strings designated by the pointers *F_SIZE for file size, *F_CREATE for create time/date, *F_MODIFY for last modified time/date, *F_ATTRIBUTE for the attribute byte, and *F_CLUS_START for the starting cluster of the file. The function returns a '0' if the variables are successfully updated, or an EOF if an error occurs (i.e. Invalid path or filename, Read error).

Example:

```
uchar filename[12];           // String to save file name
ulong filesize;               // Variable to save file size in
uchar create_date_str[20];    // String to save create time/date
uchar modify_date_str[20];    // String to save last modified time/date
uchar file_attr;              // Variable to save file attribute byte to
uint file_clus_start;         // Variable to save starting cluster of file to

strcpyf(filename, "TESTFILE.TXT"); // Save file name to string

// Save file attributes for "TESTFILE.TXT" to variables
fget_file_info(filename, &filesize, create_date_str, modify_date_str, &file_attr,
&file_clus_start);
```

sint fget_file_infoc(uchar flash *NAME, ulong *F_SIZE, uchar *F_CREATE, uchar *F_MODIFY, uchar *F_ATTRIBUTE, uint *F_CLUS_START)

This function is the same as "fget_file_info", except that a constant strings can be used to directly open a file whose name is located in the in codeflash data area.

Example:

```
ulong filesize;               // Variable to save file size in
uchar create_date_str[20];    // String to save create time/date
uchar modify_date_str[20];    // String to save last modified time/date
uchar file_attr;              // Variable to save file attribute byte to
uint file_clus_start;         // Variable to save starting cluster of file to

// Save file attributes for "TESTFILE.TXT" to variables
fget_file_infoc("TESTFILE.TXT", &filesize, create_date_str, modify_date_str, &file_attr,
&file_clus_start);
```

sint chdir(schar *F_PATH)

This function changes the working directory to the directory designated by the character string *F_PATH. If the string starts with a '\', the path is relative to the root directory; a '\' in the middle of the path string indicates sub-directories. A ".." within the string (between or before a '\') indicates the previous directory relative to the current directory. This function returns a '0' if the working directory is successfully changed, or an EOF if the working directory could not be changed (i.e. Folder does not exist, Read failure, Path does not exist).

Example:

```
if (initialize_media())        // If the media is initialized ok,
    read_directory();          // Send file list of card to serial port
// Save folder name to string
strcpyf(foldername, "TESTFOLD"); // Save folder name to string
flag = chdir(foldername);      // This call changes the working directory to [TESTFOLD]
if (flag==0)
    read_directory();          // Send file list of card to serial port
```

Progressive Resources LLC

Flash File

Terminal Output (Assume TESTFOLD is a directory):

```
File Listing for:  ROOT\
[TESTFOLD]
TEST1.TXT          [131289]  bytes
TEST2.TXT          [229327]  bytes
```

```
File Listing for:  ROOT\TESTFOLD\
[TESTDIR]
TEST3.TXT          [32623]   bytes
TEST4.TXT          [43974]   bytes
```

sint **chdirc**(schar flash *F_PATH)

This function is the same as “chdir”, except that a constant strings can be used to directly change directories to a directory whose name is located in the in codeflash data area.

Example:

```
if (initialize_media())           // If the media is initialized ok,
    read_directory();             // Send file list of card to serial port
flag = chdirc("\\TESTFOLD\\TESTDIR"); // This call changes the working directory to
                                     // [\TESTFOLD\TESTDIR]
if (flag==0)
    read_directory();             // Send file list of card to serial port
```

Terminal Output (Assume TESTFOLD is a directory):

```
File Listing for:  ROOT\
[TESTFOLD]
TEST1.TXT          [131289]  bytes
TEST2.TXT          [229327]  bytes
```

```
File Listing for:  ROOT\TESTFOLD\TESTDIR\
FILE1.TXT          [252684]  bytes
FILE2.TXT          [185201]  bytes
```

FILE ***fopen**(uchar *NAME, uchar MODE)

This function opens the file designated by the character string *NAME, and associates it to an input and/or output stream based on the MODE handle. The function allocates the required memory to handle files, then returns a FILE pointer to the allocated space. If there is an error allocating the memory or opening the file, a NULL pointer is returned. (Note: Filenames used must be in an 8.3 DOS format, long filenames are not currently supported)

The MODE's can be READ, WRITE, or APPEND:

- * **READ** – Opens a file from the beginning, to read only
- * **WRITE** – Destroys contents of a file and opens it from the beginning for writing
- * **APPEND** – Opens a file to the end for writing

Example:

```
strcpyf(filename, "TEST.TXT");    // Save file name to string
fp = fopen(filename, READ);        // This call opens the file "TEST.TXT" in the current
                                     // directory for reading, referenced by the type FILE
                                     // pointer *fp
```

FILE ***fopenc**(uchar flash *NAMEC, uchar MODEC)

This function is the same as “fopen”, except that a constant string is used to directly open a file whose name is located in the in codeflash data area.

Progressive Resources LLC

Flash File

Example:

```
fp = fopen("\\FOLD\\TEST.TXT", READ);    // This call opens the file "TEST.TXT" in the
                                         // directory [FOLD] for reading
```

sint **mkdir**(*schar* **F_PATH*)

This function creates a directory designated by the character string **F_PATH*. This function returns a '0' if the folder is successfully created, and an *EOF* if the folder could not be created (i.e. Folder with same name already exists, Write to media failure, Media is full).

Example:

```
if (initialize_media())                // If the media is initialized ok,
    read_directory();                  // Send file list of card to serial port
// Save file name to string
strcpyf(foldname, "TESTFOLD");        // Save folder name to string
flag = mkdir(foldname);               // This call creates the folder "TESTFOLD" on the SD card
read_directory();                     // Send file list of card to serial port
```

Terminal Output (Assume TEST is a directory, TEST1.TXT and TEST2.TXT are files):

```
File Listing for:  ROOT\
[TEST]
TEST1.TXT          [131289] bytes
TEST2.TXT          [229327] bytes
```

```
File Listing for:  ROOT\
[TEST]
TEST1.TXT          [131289] bytes
TEST2.TXT          [229327] bytes
[TESTFOLD]
```

FILE* **fcreate(*uchar* **NAME*, *uchar* *MODE*)

This function creates a file of size 0 bytes, designated by the character string **NAME*, with file attributes *MODE*, then opens the file in WRITE mode. The *MODE* switch designates what will be written to the "attribute" byte of the file (the "archive" bit is always assumed to be set). Multiple *MODE*s can be used by inserting a bitwise "OR" symbol '|' between the different attributes being set. If a file with the same name already exists, it over writes the file to a blank file, then opens the file in WRITE mode (unless the existing file is "read only"). This function returns a *FILE* pointer to the opened FILE structure if the file is successfully created, a *NULL* pointer is returned if the file could not be created (i.e. File with same name already exists as "READ_ONLY", Write to media failure, Media is full).

The *MODE*(s) can be *ATTR_READ_ONLY*, *ATTR_HIDDEN*, and/or *ATTR_SYSTEM*:

- * *ATTR_READ_ONLY* – Once the file is closed, it cannot be modified with this attribute
- * *ATTR_HIDDEN* – The file will not be visible to the user
- * *ATTR_SYSTEM* – The file will be marked as a system file

Example:

```
if (initialize_media())                // If the media is initialized ok,
    read_directory();                  // Send file list of card to serial port
// Save file name to string
strcpyf(filename, "TEST.TXT");        // Save file name to string
file1 = fcreate(filename, 0);         // This call creates the file "TEST.TXT" onto the SD card
fclose(file1);                        // Close the file
read_directory();                     // Send file list of card to serial port
```

Terminal Output (Assume TEST is a directory, TEST1.TXT and TEST2.TXT are files):

```
File Listing for:  ROOT\
```

Progressive Resources LLC

Flash File

```
[TEST]
TEST1.TXT      [131289] bytes
TEST2.TXT      [229327] bytes

File Listing for: ROOT\
[TEST]
TEST1.TXT      [131289] bytes
TEST2.TXT      [229327] bytes
TEST.TXT       [0] bytes
```

***FILE fcreatec(uchar flash *NAMEC, uchar MODE)**

This function is the same as “fcreate”, except that a constant string is used to directly create a file whose name is located in the in codeflash data area.

Example:

```
file = fcreatec("TEST.TXT", 0);           // This call creates and opens the file "TEST.TXT"
```

sint rmdir(schar *F_PATH)

This function deletes the directory designated by the character string *F_PATH. This function returns a ‘0’ if the folder is successfully removed, and an EOF if the folder could not be deleted (i.e. Folder does not exist, Folder not empty, Write to media failure, Read only file).

Example:

```
strcpyf(foldername, "TESTFOLD");           // Save folder name to string
flag = rmdir(foldername);                   // This call deletes the folder [TESTFOLD]
```

sint remove(uchar *NAME)

This function deletes the file designated by the character string *NAME. This function returns a ‘0’ if the file is successfully removed, and an EOF if the file could not be deleted (i.e. File does not exist, Write to media failure, Read only file).

Example:

```
// Save file name to string
strcpyf(filename, "TEST.TXT");              // Save file name to string
flag = remove(filename);                    // This call deletes the file "TEST.TXT" from the SD card
```

sint removec(uchar flash *NAMEC)

This function is the same as “remove”, except that a constant strings can be used to directly delete a file whose name is located in the in codeflash data area.

Example:

```
flag = removec("TEST.TXT");                 // This call deletes the file "TEST.TXT" from the SD card
```

sint rename(uchar *NAME_OLD, uchar *NAME_NEW)

This function renames one file designated by the character string *NAME_OLD, to be designated by the character string *NAME_NEW. This function returns a ‘0’ if the file is successfully renamed, and an EOF if the file could not be renamed (i.e. File does not exist, File with new name already exists, Write to media failure, Read only file). If using full directory

Progressive Resources LLC

Flash File

addressing, the *NAME_OLD* string can have the full path length, where the *NAME_NEW* has just the new name of the file.

Example:

```
// Save file names to strings
strcpyf(filename1, "FOLDER\\TEST1.TXT");
strcpyf(filename2, "TEST2.TXT");
flag = rename(filename1, filename2);
```

```
// Save file name to string
// Save file name to string
// This call renames the file "TEST1.TXT"
// located in the directory [FOLDER] to
// "TEST2.TXT"
```

*sint fclose(FILE *rp)*

This function saves the data (if applicable) and closes the file designated by *FILE* pointer **rp* and frees the associated memory allocated for the file. This function returns a '0' if the file is closed, and an *EOF* if an error occurred while closing the file (i.e. *FILE* pointer is *NULL*).

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to append
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
flag = fclose(fp); // save and close file associated with FILE pointer *fp
```

*sint fflush(FILE *rp)*

This function writes all buffered data of the file designated by *FILE* pointer **rp* to the Secure Digital card. This function returns a '0' if the file is saved successfully, and an *EOF* if an error occurred while saving the file (i.e. *FILE* pointer is *NULL*, File is Read Only).

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to append
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
flag = fflush(fp); // save file associated with FILE pointer *fp
```

*sint freemem(FILE *rp)*

This function frees the associated memory allocated for the file designated by *FILE* pointer **rp*. This function returns a '0' if the memory is freed successfully, and an *EOF* if an error occurred while saving the file (i.e. *FILE* pointer is *NULL*).

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to append
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
flag = freemem(fp); // Free memory associated with FILE pointer *fp
```

*slong ftell(FILE *rp)*

This function returns the current position (relative to the beginning of the file) of the data pointer for the file designated by *FILE* pointer **rp*. This function returns an *EOF* if an error occurred while finding the position (i.e. *FILE* pointer is *NULL*, Read failure).

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to append
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
location = ftell(fp); // find the position within data of the file
```

Progressive Resources LLC

Flash File

sint fseek(FILE *rp, ulong off_set, uchar mode)

This function moves the data pointer associated with the file designated by *FILE* pointer **rp*. The position the data pointer is moved to is based on the *off_set* and *mode* handles. This function returns a '0' if the data pointer is successfully moved, and an *EOF* if an error occurs during the positioning of the data pointer in the file (i.e. The *off_set* relative to the *mode* is outside the range of the file, Read failure).

The *mode*'s can be *SEEK_CUR*, *SEEK_END*, or *SEEK_SET*:

- * *SEEK_CUR* – Positions by *off_set* relative to the current data pointer location
- * *SEEK_END* – Positions by *off_set* relative to the end of the file
- * *SEEK_SET* – Positions by *off_set* relative to the beginning of the file

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to append
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
location = ftell(fp); // find the position within data of the file
do_stuff(fp); // do stuff to file associated with FILE pointer *fp
flag = fseek(fp, location, SEEK_SET); // returns data pointer to "location"
```

sint fgetc(FILE *rp)

This function returns the character at the location the data pointer is pointing to in the file designated by *FILE* pointer **rp*, then increments the data pointer. This function returns an *EOF* if an error occurs (i.e. *NULL* file pointer, Read failure).

Example:

```
fp = fopen("TEST.TXT", READ); // open file "TEST.TXT" to read
putchar(fgetc(fp)); // output the first data byte, increment file pointer
```

sint fread(void *rd_buff, uint dat_size, uint num_items, FILE *rp)

This function reads *num_items* of size *dat_size* bytes from the file designated by *FILE* pointer **rp*, into the buffer designated by the pointer **rd_buff*, from the location the data pointer is pointing to. This function returns the number of successfully read items, less the number of errors.

Example:

```
uint temp_buff[10];
fp = fopen("TEST.TXT", READ); // open file "TEST.TXT" to read
if (fread(temp_buff, 2, 10, fp) != 10) // read 10 words of data from the file
    return (EOF); // return an error if not 10 items
```

schar *fgets(schar *buffer, sint n, FILE *rp)

This function returns a character pointer to the string designated by the pointer **buffer*, filling the string with data from the file designated by *FILE* pointer **rp*, up to *n* characters or a line feed ('\n'). This function returns an *EOF* if an error occurs (i.e. *NULL* file pointer, Read failure).

Example:

```
uchar data_buff[20]; // read data buffer
uchar *pntr; // Data pointer
fp = fopen("TEST.TXT", READ); // open file "TEST.TXT" to read
```

Progressive Resources LLC

Flash File

```
pntr = fgets(data_buff, 20-1, fp); // save first data line of file to the string data_buff
```

sint **ungetc**(uchar *file_data*, FILE **rp*)

This function decrements the data pointer in the file designated by *FILE* pointer **rp*, then pushes the character *file_data* back to the file. This function returns the character *file_data* if successful, or it returns an *EOF* if an error occurs (i.e. *NULL* file pointer, Read failure).

Example:

```
fp = fopen("TEST.TXT", READ); // open file "TEST.TXT" to read
flag = fseek(fp, 0, SEEK_END); // move the data pointer to the end of the file
putchar(ungetc('T', fp)); // Put a 'T' in the last data byte, then -- pntr
```

sint **fputc**(uchar *file_data*, FILE **rp*)

This function inserts the character *file_data*, in the file designated by *FILE* pointer **rp*, at the location the data pointer is pointing to, then increments the data pointer. If the character to be written is at the start of a new sector, this function will first flush and save the data buffer before writing the next character to the file. This function returns the value of the character *file_data* if the input was successful, or an *EOF* if an error occurs (i.e. *NULL* file pointer, Write failure, File in Read Only mode).

Example:

```
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to the end in append mode
fputc('S', fp); // write an 'S' to the file, increment file pointer
```

sint **fwrite**(void **wr_buff*, uint *dat_size*, uint *num_items*, FILE **rp*)

This function inserts *num_items* of size *dat_size* bytes from the buffer designated by the pointer **wr_buff*, into the file designated by *FILE* pointer **rp*, at the location the data pointer is pointing to. If the string to be written crosses sectors or is at the start of a new sector, this function will first flush and save the data buffer of the previous sector before filling the next write buffer of the file. This function returns the number of successfully written items, less the number of errors.

Example:

```
uint temp_buff[10];
uint i;
for (i=0; i<10; i++)
    temp_buff[i] = (i * 0x200) + i; // Fill buffer with data
fp = fopen("TEST.TXT", APPEND); // open file "TEST.TXT" to the end in append mode
if (fwrite(temp_buff, 2, 10, fp) != 10) // add the 10 words of data to the end of the file
    return (EOF); // return an error if not 10
```

sint **fputs**(uchar **file_data*, FILE **rp*)

This function inserts the string designated by the pointer **file_data*, in the file designated by *FILE* pointer **rp*, at the location the data pointer is pointing to, followed by a line feed (0x0A) and carriage return (0x0D). If the string to be written crosses sectors or is at the start of a new sector, this function will first flush and save the data buffer of the previous sector before filling the next write buffer of the file. This function returns a '0' if the input string was successfully

Progressive Resources LLC

Flash File

added to the file, or an *EOF* if an error occurs (i.e. *NULL* file pointer, Write failure, File in Read Only mode).

Example:

```
strcpyf(inputstring, "Hello World!"); // Save string to SRAM
fp = fopenc("TEST.TXT", APPEND);      // open file "TEST.TXT" to the end in append mode
fputs(inputstring, fp);                // add "Hello World!\r\n" to the end of the file
```

sint **fputc**(*uchar* *flash *file_data*, *FILE *rp*)

This function is the same as “fputs”, except that a constant string located in the in codeflash data area can be used as a direct input to the file designated by the *FILE* pointer **rp*.

Example:

```
fp = fopenc("TEST.TXT", APPEND);      // open file "TEST.TXT" to the end in append mode
fputc("Hello World!", fp);            // add "Hello World!\r\n" to the end of the file
```

void **fprintf**(*FILE *rp*, *uchar* *flash *pstr*, ...)

This function inserts the constant string **pstr*, at the location the data pointer is pointing to, in the file designated by *FILE* pointer **rp*, with the same options that printf supports (i.e. %x, %lx, %X, %lX, %d, %ld, etc.).

Example:

```
fp = fopenc("TEST.TXT", APPEND);      // open file "TEST.TXT" to the end in append mode
sum = 3 + 4;                          // assign sum the value of 7
fprintf(fp, "The number %d is the answer", sum); // output the string to the file
// "The number 7 is the answer" will be written to the file incrementing the data
// pointer as the string is written
```

Note: The maximum length of both the input string and the formatted string is defined in the “options.h” file as `_FF_MAX_FPRINTF`.

sint **feof**(*FILE *rp*)

This function indicates whether the data pointer is at the end of the file designated by *FILE* pointer **rp*. This function returns a *1* if the data pointer is at the end of the file, a *0* if the data pointer is not at the end of the file, or an *EOF* if an error occurs (i.e. *NULL* file pointer).

Example:

```
fp = fopenc("TEST.TXT", READ);        // open file "TEST.TXT" to read
while (feof(fp)==0)                  // output file stream if not at the end or error
    putchar(fgetc(fp));
```

Example FlashFile Application

The example application program for the FlashFile (“demo.prj” included in the package) is designed to create the data file “demo.dat” on a preformatted SD/MMC or Compact Flash card (FAT12 or FAT16). If no card exists (or the card is not formatted correctly) PORTB.6 toggles until an initialization of a card is successful. If the file already exists, the program deletes the existing data in that file and replaces it with the new data. The first line of data will be the Date/Time stamp of the file, followed by a line of column headers, then each line of data. A comma between each column and a line feed/carriage return for each row separates the data so the file can be directly

Progressive Resources LLC

Flash File

imported into a spreadsheet program like MS Excel. The SD/MMC example application is designed to run directly on the Progressive Resources LLC Mega-128.NET development board; the Compact Flash example is setup to run on an Atmel ATmega128 (connections described in the "Implementation" section of this document).

Note: The included demo project is pre-setup to integrate the media referenced (SD/MMC or CF) in the ordered package. If both the SD/MMC and CF packages were purchased, the media used depends on the "options.h" file as described in the "OPTIONS.H Header File" section of this document.

Including the FlashFile

The FlashFile files must be included in the project to use any of the communications and/or file system functions for the preferred flash card. At the top of the main file, declaring the options header enables the use of any of the functions. Any information that needs to be accessed from the main source file must also be declared (i.e. global variables from other files' function calls).

```
#include "options.h"

// Declare your global variables here
extern unsigned int BPB_BytsPerSec;
#ifdef _RTC_ON_
    extern unsigned char rtc_hour, rtc_min, rtc_sec;
    extern unsigned char rtc_date, rtc_month;
    extern unsigned int rtc_year;
#endif
#ifdef _ICCAVR_
    extern char _bss_end;
#endif
```

Importing the Data

Once the file is created, processing the document into a spreadsheet program like MS Excel is simple. The program that will be used to view the file must be opened, then select "file => open" and select "All Files (*.*)" under "Files of Type", then select the desired file to be opened (in this case the file to be opened will be "DEMO.DAT"). When opening a non "*.xls" file, Excel should automatically start the Import Wizard to see how the data is separated. If the FlashFile is used properly when gathering data, a delimiter will be used to separate the data (a comma is used in the demo), and the import wizard will put all of the data into a properly formatted rows and columns. An example of importing the data into MS Excel is listed below:

Progressive Resources LLC

Flash File

Text Import Wizard - Step 1 of 3

The Text Wizard has determined that your data is Fixed Width.
If this is correct, choose Next, or choose the data type that best describes your data.

Original data type

Choose the file type that best describes your data:

☒ Delimited - Characters such as commas or tabs separate each field.
☐ F*ixed width* - Fields are aligned in columns with spaces between each field.

Start import at row: File origin:

Preview of file F:\DEMO.DAT.

1	"12/30/2003, 13:37:26"
2	Column 0, Column 1, Column 2, Column 3, Column 4, Column 5, Column 6
3	0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
4	10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
5	20, 21, 22, 23, 24, 25, 26, 27, 28, 29,

Cancel < Back Next > Finish

MS Excel - Import Wizard: Select "Delimited"

Text Import Wizard - Step 2 of 3

This screen lets you set the delimiters your data contains. You can see how your text is affected in the preview below.

Delimiters

☐ Tab ☐ Semicolon ☒ Comma ☐ Treat consecutive delimiters as one
☐ Space ☐ Other:

Text qualifier:

Data preview

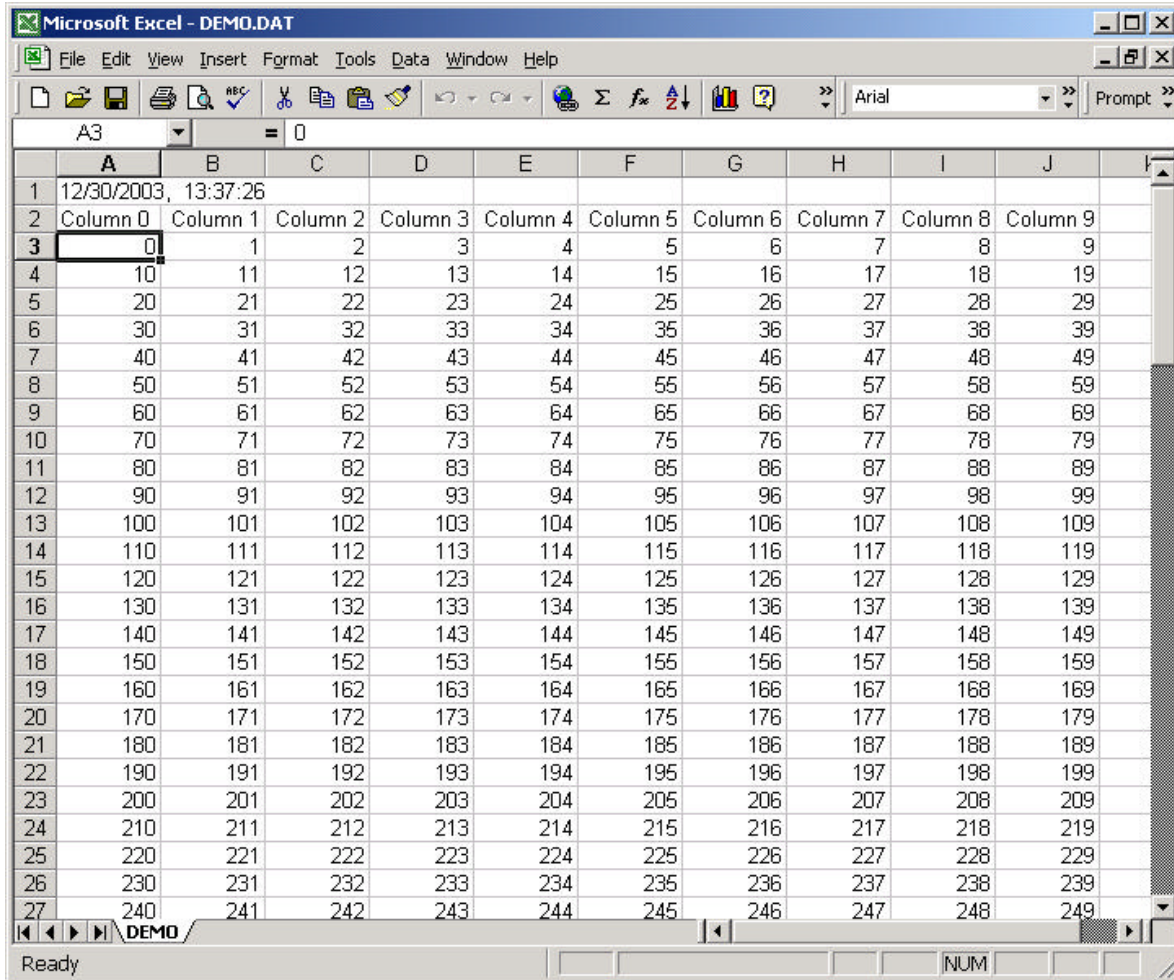
12/30/2003, 13:37:26	Column 1	Column 2	Column 3	Column 4	Column 5
Column 0	1	2	3	4	5
0	11	12	13	14	15
10	21	22	23	24	25

Cancel < Back Next > Finish

MS Excel - Import Wizard: Deselect "Tab" - Select "Comma"

Progressive Resources LLC

Flash File



MS Excel – Open file “DEMO.DAT”

Full Example Code Listing

```
/*  
Project : FlashFileSD Example  
Version :  
Date : 12/23/2003  
Author : Erick M Higa  
Company : Progressive Resources LLC  
Comments:  
This is a simple example program for the FlashFileSD
```

```
Chip type : ATmega128  
Program type : Application  
Clock frequency : 14.745600 MHz  
Memory model : Small  
External SRAM size : 0  
Data Stack size : 1024
```

```
*/
```

```
#include "options.h"
```

```
#ifndef _UART_INT_  
#define rx_counter0 (UCSR0A & 0x80)  
#define tx_counter0 ((!UCSR0A) & 0x40)
```

Progressive Resources LLC

Flash File

```
#define rx_counter1 (UCSR1A & 0x80)
#define tx_counter1 (!UCSR1A) & 0x40)
#endif

void port_init(void)
{
    PORTA = 0xFF;    DDRA = 0x00;
    PORTB = 0xFF;    DDRB = 0xD0;
    PORTC = 0xFF;    DDRC = 0x00;    //m103 output only
    PORTD = 0xFF;    DDRD = 0x00;
    PORTE = 0xFF;    DDRE = 0x00;
    PORTF = 0xFF;    DDRF = 0x00;
    PORTG = 0x1F;    DDRG = 0x00;
}

//UART0 initialisation
// desired baud rate: 115200
// actual: baud rate:115200 (0.0%)
// char size: 8 bit
// parity: Disabled
void uart0_init(void)
{
    UCSR0B = 0x00; //disable while setting baud rate
    UCSR0A = 0x00;
    UCSR0C = 0x06;
    UBRR0L = 0x07; //set baud rate lo
    UBRR0H = 0x00; //set baud rate hi
    UCSR0B = 0x18;
}

//UART1 initialisation
// desired baud rate:115200
// actual baud rate:115200 (0.0%)
// char size: 8 bit
// parity: Disabled
void uart1_init(void)
{
    UCSR1B = 0x00; //disable while setting baud rate
    UCSR1A = 0x00;
    UCSR1C = 0x06;
    UBRR1L = 0x07; //set baud rate lo
    UBRR1H = 0x00; //set baud rate hi
    UCSR1B = 0x18;
}

//call this routine to initialise all peripherals
void init_devices(void)
{
    //stop errant interrupts until set up
    CLI(); //disable all interrupts
    XDIV = 0x00; //xtal divider
    XMCRA = 0x00; //external memory
    port_init();
    uart0_init();
    uart1_init();

    MCUCR = 0x00;
    EICRA = 0x00; //extended ext ints
    EICRB = 0x00; //extended ext ints
    EIMSK = 0x00;
    TIMSK = 0x00; //timer interrupt sources
    ETIMSK = 0x00; //extended timer interrupt sources
    SEI(); //re-enable interrupts
    //all peripherals are now initialised
}

// Declare your global variables here
extern unsigned char rtc_hour, rtc_min, rtc_sec;
extern unsigned char rtc_date, rtc_month;
```

Progressive Resources LLC

Flash File

```
extern unsigned int rtc_year;
#ifdef _ICCAVR_
    extern char _bss_end;
#endif

void main(void)
{
    FILE *pnt1;
    unsigned long c, n;

    init_devices();

#ifdef _ICCAVR_
    // Must set up heap size in ICC Only
    _NewHeap(&_bss_end + 1, &_bss_end + 1001);
#endif

#ifdef _RTC_ON_
    twi_setup();
#endif

    PORTB |= 0xD0;

    // initialize the Secure Digital card
    while (initialize_media()==0)
    {
        // Blink LED while waiting to initialize
        PORTB ^= 0x40;
    }
    PORTB &= 0x5F;

    // Create File

    pnt1 = fcreatec("demo.dat", 0);
    while (pnt1==0)
        pnt1 = fcreatec("demo.dat", 0);

    // Write to file
#ifdef _RTC_ON_
    // if real time clock enabled, get and print time and date to file
    rtc_get_timeNdate(&rtc_hour, &rtc_min, &rtc_sec, &rtc_date, &rtc_month, (int *)&rtc_year);
    fputc(0x22, pnt1); // put a " in before time/date
    if ((rtc_month/10)==0)
        fputc('0', pnt1);
    fprintf(pnt1, "%d/", rtc_month);
    if ((rtc_date/10)==0)
        fputc('0', pnt1);
    fprintf(pnt1, "%d/", rtc_date);
    fprintf(pnt1, "%d ", rtc_year);
    if ((rtc_hour/10)==0)
        fputc('0', pnt1);
    fprintf(pnt1, "%d:", rtc_hour);
    if ((rtc_min/10)==0)
        fputc('0', pnt1);
    fprintf(pnt1, "%d:", rtc_min);
    if ((rtc_sec/10)==0)
        fputc('0', pnt1);
    fprintf(pnt1, "%d", rtc_sec);
    fputc(0x22, pnt1); // put a " in after time/date
    fputc("", pnt1);
#endif

    fprintf(pnt1, "Column %d, Column %d, Column %d, Column %d, Column %d, Column %d, Column %d, Column %d, Column %d, Column %d,\r\n", 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

    for (c=0; c<100; c++)
    {
        // print numbers separated by commas
        for (n=0; n<10; n++)
            fprintf(pnt1, "%ld, ", (((long)c*10)+(long)n));
        if (fputc('\r', pnt1)==EOF)
            break;
    }
}
```

Progressive Resources LLC

Flash File

```
        break;           // line feed/carriage return
    if (fputc('\n', ptr1)==EOF)
        break;           // line feed/carriage return
    PORTB ^= 0x40;
    PORTB ^= 0x80;
}

fclose(ptr1);

printf("\r\n\r\nDONE!!!");
PORTB &= 0xBF; // Keep LED on when done
while (1)
{ // Blink LED when done
    PORTB ^= 0x80;
    for (c=0; c<100000; c++)
        ;
};
}
```

Progressive Resources LLC

Flash File

Software License

The use of Progressive Resources LLC FlashFile for Secure Digital or Compact Flash indicates your understanding and acceptance of the following terms and conditions. This license shall supersede any verbal or prior verbal or written, statement or agreement to the contrary. If you do not understand or accept these terms, or your local regulations prohibit "after sale" license agreements or limited disclaimers, you must cease and desist using this product immediately. This product is © Copyright 2003 by Progressive Resources LLC, all rights reserved. International copyright laws, international treaties and all other applicable national or international laws protect this product. This software product and documentation may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine readable form, without prior consent in writing, from Progressive Resources LLC and according to all applicable laws. The sole owner of this product is Progressive Resources LLC.

Operating License

You have the non-exclusive right to use any enclosed product but have no right to distribute it as a source code product without the express written permission of Progressive Resources LLC. Use over a "local area network" (within the same locale) is permitted provided that only a single person, on a single computer uses the product at a time. Use over a "wide area network" (outside the same locale) is strictly prohibited under any and all circumstances.

Liability Disclaimer

This product and/or license is provided as is, without any representation or warranty of any kind, either express or implied, including without limitation any representations or endorsements regarding the use of, the results of, or performance of the product, Its appropriateness, accuracy, reliability, or correctness. The user and/or licensee assume the entire risk as to the use of this product. Progressive Resources LLC does not assume liability for the use of this product beyond the original purchase price of the software. In no event will Progressive Resources LLC be liable for additional direct or indirect damages including any lost profits, lost savings, or other incidental or consequential damages arising from any defects, or the use or inability to use these products, even if Progressive Resources LLC have been advised of the possibility of such damages.

Progressive Resources LLC
4105 Vincennes Road
Indianapolis, IN 46268

<http://www.prlc.com>