CHAPTER 10

Memory and Data Management

Chapter Outline

10.1 A Memory Review

10.1.1 Memory Function Types

Broadly speaking, a microprocessor needs memory for two reasons: to hold its program, and to hold the data that it is working with; these are often called *program memory* and *data memory.*

To meet these needs a number of different semiconductor memory technologies is available, which can be embedded on the microcontroller chip. Memory technology is divided broadly into two types: volatile and non-volatile. Non-volatile memory retains its data when power is removed, but tends to be more complex to write to in the first place. For historical reasons it is still often called ROM (read only memory). Non-volatile memory is generally

Fast and Effective Embedded Systems Design. DOI: 10.1016/B978-0-08-097768-3.00010-6

Copyright © 2012 Elsevier Ltd. All rights reserved.
Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

required for program memory, so that the program data is there and ready when the processor is powered up. Volatile memory loses all data when power is removed, but is easy to write to. Volatile memory is traditionally used for data memory; it is essential to be able to write to memory easily, and there is little expectation for data to be retained when the product is switched off. For historical reasons it is often called RAM (random access memory), although this terminology tells us little that is useful. These categorizations of memory, however, give an over-simplified picture. It can be useful to change the contents of program memory, and there are times when we want to save data long term. Moreover, new memory technologies now provide non-volatile memory that is easy to write to.

10.1.2 Essential Electronic Memory Types

In any electronic memory we want to be able to store all the 1s and 0s that make up our data. There are several ways that this can be done, and a few essential ones are outlined here.

A simple one-bit memory is a coin. It is stable in two positions, with either 'heads' facing up, or 'tails'. We can try to balance the coin on its edge, but it would pretty soon fall over. The coin is stable in two states, and we call this *bistable*. It could be said that 'heads' represents Logic 1 and 'tails' Logic 0. With eight coins, an 8-bit number can be represented and stored. If we had 10 million coins, we could store the data that makes up one photograph of good resolution, but that would take up a lot of space indeed!

There are various electronic alternatives to the coin, which take up much less space. One is to use an electronic bistable (or 'flip-flop') circuit, as shown in Figure 10.1. The two circuits of

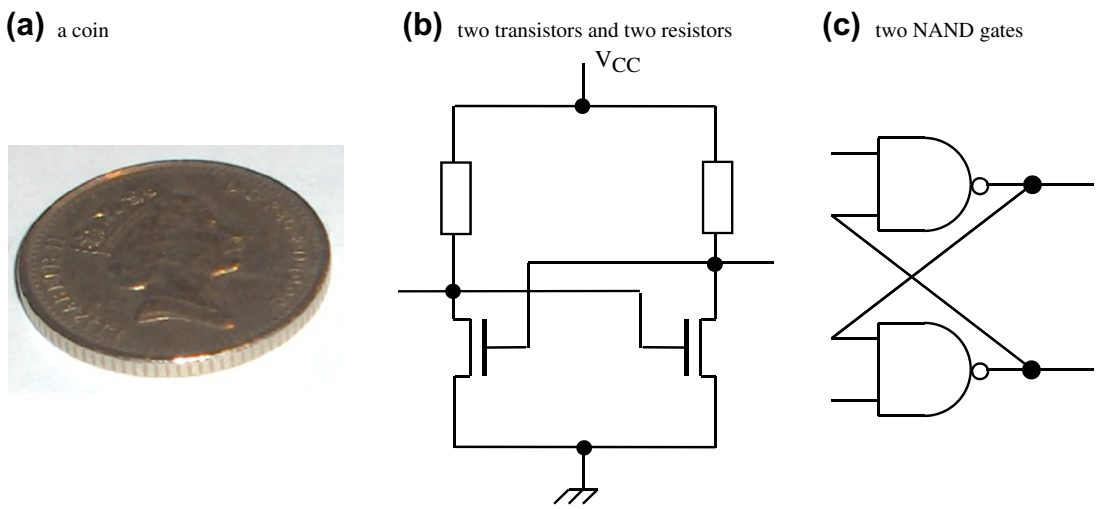

Figure 10.1:
Three ways of implementing a one-bit memory unit

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

Figure 10.1b and c are stable in only two states, and each can be used to store one bit of data. Circuits like these have been the bedrock of volatile memory.

Looking at the bigger picture, there are several different types of volatile and non-volatile memory, as shown in Figure 10.2 and described in some detail in References 1.1 and 10.1. Static random access memory (SRAM) consists of a vast array of memory cells based on the circuit of Figure 10.1b. These have to be addressable, so that just the right group of cells is written to, or read from, at any one time. To make this possible, two extra transistors are added to the two outputs. To reduce the power consumption, the two resistors are usually replaced by two transistors also. That means six transistors per memory cell. Each transistor takes up a certain area on the integrated circuit (IC), so when the circuit is replicated thousands or millions of times, it can be seen that this memory technology is not actually very space efficient. Despite that, it is of great importance; it is low power, can be written to and read from with ease, can be embedded onto a microcontroller, and hence forms the standard way of implementing data memory in most embedded systems. All data is lost when power is removed.

Dynamic random access memory (DRAM) is intended to do the same thing as SRAM with a reduced silicon area. Instead of using a number of transistors, one bit of information is stored in a tiny capacitor, like a small rechargeable battery. Such capacitors can be fabricated in large numbers on an IC. In order to select the capacitor for reading or writing, a simple transistor switch is required. Unfortunately, owing to the small capacitors and leakage currents on the chip, the memory loses its charge over a short period of time (around 10 to 100 ms). So the DRAM needs to be accessed every few milliseconds to refresh the charges, otherwise the information is lost. DRAM has about four times larger storage capacity than SRAM at about the same cost and chip size, with a compromise of the extra work involved in regular refreshing. It is, moreover, power hungry, so inappropriate for any

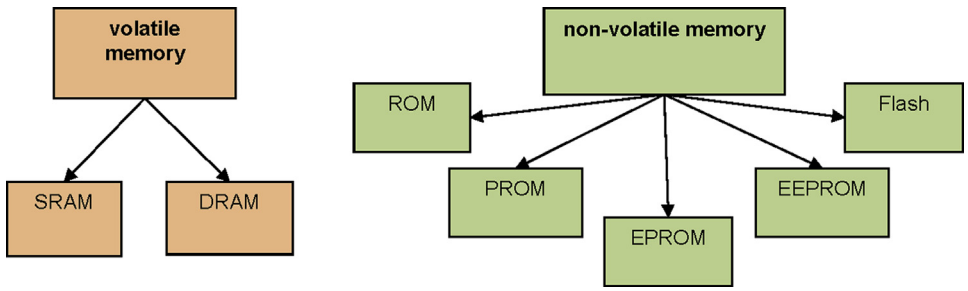

Figure 10.2:
Electronic memory types. SRAM: static random access memory; DRAM: dynamic random access memory; ROM: read only memory; PROM: programmable ROM; EPROM: electrically programmable ROM; EEPROM: electrically erasable and programmable ROM

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

battery-powered device. DRAM has found wide application as data memory in mains-powered computers, such as the PC.

The original ROMs and programmable read only memories (PROMs) could only ever be programmed once, and have now completely disappeared from normal usage. The first type of non-volatile reprogrammable semiconductor memory, electrically programmable read only memory (EPROM), represented a huge step forward – a non-volatile memory could now be reprogrammed. With a process called *hot electron injection* (HEI), electrons can be forced through a very thin layer of insulator, onto a tiny conductor embedded within the insulator, and can be trapped there almost indefinitely. This conductor is placed so that it interferes with the action of a field effect transistor (FET). When it is charged/discharged, the action of the FET is disabled/enabled. This modified FET effectively becomes a single memory cell, far more dense than the SRAM discussed previously. Moreover, the memory effect is non-volatile; trapped charge is trapped charge! This programming does require a comparatively high voltage (around 25 V), so generally needs a specialist piece of equipment. The memory is erased by exposure to intense ultraviolet light; EPROMs can always be recognized by the quartz window on the IC, which allows this to happen.

The next step beyond HEI was *Nordheim Fowler Tunneling*. This requires even finer memory cell dimensions, and gives a mechanism for the trapped charge to be retrieved electrically, which had not previously been possible. With electrically erasable and programmable read only memory (EEPROM), words of data are individually writeable, readable and erasable, and are non-volatile. The downside of this is that more transistors are needed to select each word. In many cases this flexibility is not required. A revised internal memory structure led to *flash* memory; in this, the ability to erase individual words is not available. Whole blocks have to be erased at any one time, 'in a flash'. This compromise leads to a huge advantage: flash memory is very high density indeed, more or less the highest available. This memory type has been a key feature of many recent products that we have now become used to, like digital cameras, memory sticks, solid-state hard drives and so on. A curious feature of flash and EEPROM, unlike most electronics, is that they exhibit a wear-out mechanism. Electrons can get trapped in the insulator through which they are forced when a write operation takes place. Therefore this limitation is often mentioned in datasheets, for example a maximum of 100 000 write/erase cycles. This is a very high number, and is not experienced in normal use.

Although EPROM had become very widely used, and had been integrated onto microcontrollers, it was rapidly and completely replaced by flash memory. Now in embedded systems, the two dominant memory technologies are flash and SRAM. A glance back at Figures 2.2 and 2.3 shows how important they are to the mbed. Program memory on the LPC1768 is flash, and data memory is SRAM. On the mbed card, the 'USB disk' is a flash IC. The other technology that we are likely to meet at times is EEPROM; this is still used where the ability to rewrite single words of data remains essential.

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

10.2 Using Data Files with the mbed

Armed with a little knowledge about memory technologies, let's explore how to access and use the mbed memory. The **LocalFileSystem** library allows us to set up a local file system for accessing the mbed flash universal serial bus (USB) disk drive. This allows programs to read and write files on the same disk drive that is used to hold the mbed programs, and which are accessed from the host computer. Once the system has been set up, the standard C/C++ file access functions can be used to open, read and write files.

10.2.1 Reviewing some Useful C/C++ Library Functions

In C/C++ we can open files, read and write data and also scan through files to specific locations, even searching for particular types of data. The functions for input and output operations are all defined by the C Standard Input and Output Library (**stdio.h**), introduced in Section B.9 of Appendix B. Here, we will use the functions summarized in Table 10.1 (this is effectively Table B.6, repeated here for convenience).

Data can be stored in files (as chars) or as words and strings (as character arrays). The mbed provides a mechanism to allow storage in its USB disk, to save data that can be recalled at a later date.

10.2.2 Defining the mbed Local File System

The compiler needs to know where to store and retrieve files; this is done using the mbed **LocalFileSystem** declaration. This sets up the mbed as an accessible flash memory storage unit and defines a directory for storing local files. To implement, simply add the line:

Table 10.1: Useful stdio library functions

Function	Format	Summary action
fopen	`FILE * fopen ( const char * filename, const char * mode);`	Opens the file of name filename
fclose	`int fclose ( FILE * stream );`	Closes a file
fgetc	`int fgetc ( FILE * stream );`	Gets a character from a stream
fgets	`char * fgets ( char * str, int num, FILE * stream );`	Gets a string from a stream
fputc	`int fputc ( int character, FILE * stream );`	Writes a character to a stream
fputs	`int fputs ( const char * str, FILE * stream );`	Writes a string to a stream
fseek	`int fseek ( FILE * stream, long int offset, int origin );`	Moves file pointer to specified location

str: An array containing the null-terminated sequence of characters to be written.
stream: Pointer to a FILE object that identifies the stream where the string is to be written.

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

```
LocalFileSystem local("local"); //Create local file system named "local"
```

to the declarations section of a program.

10.2.3 Opening and Closing Files

A file stored on the mbed (in this example called '**datafile.txt**') can therefore be opened with the following command:

```
FILE* pFile = fopen("/local/datafile.txt","w");
```

C code feature

The **fopen( )** function call uses the * operator to assign a pointer with name **pFile** to the file at the specific location given. From here onwards we can access the file by referring to its pointer (**pFile**) rather than having to use the specific filename.

We also need to specify whether we want read or write access to the file. This is done by the 'w' specifier, which is referred to as an *access mode* (in this case denoting *write* access). If the file does not already exist the **fopen( )** function will automatically create it in the specified location. Several other file open access modes, and their specific meanings, are shown in Table B.7 and are elaborated further in Reference 10.2. The three most common access modes are given also in Table 10.2. The *append* access mode (denoted by 'a') is useful for opening an existing file and writing additional data to the end of that file.

When you have finished using a file for reading or writing it is essential to close it, for example using:

```
fclose(pFile);
```

If you fail to do this you might lose all access to the mbed (see Section 10.3.1)!

10.2.4 Writing and Reading File Data

If the intention is to store numerical data, this can be done in a simple way by storing individual 8-bit data values. The **fputc** function allows this, as follows:

```
char write_var=0x0F;
fputc(write_var, pFile);
```

Table 10.2: Common access modes for fopen

Access mode	Meaning	Action
'r'	Read	Opens an existing file for reading
'w'	Write	Creates a new empty file for writing. If a file of the same name already exists it will be deleted and replaced with a blank file
'a'	Append	Appends to a file. Write operations result in data being appended to the end of the file. If the file does not exist a new blank file will be created

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

This stores the 8-bit variable **write_var** to the data file. The data can also be read from a file to a variable as follows:

```
read_var = fgetc(pFile);
```

Using the **stdio.h** functions, it is also possible to read and write words and strings and search or move through files looking for particular data elements. The C/C++ **fseek( )** function can be used to search through text files.

10.3 Example mbed Data File Access

10.3.1 File Access

Program Example 10.1 creates a data file and writes the arbitrary value 0x23 to that file. The file is saved on the mbed USB disk. The program then opens and reads back the data value and displays it to the screen in a host terminal application.

```
/* Program Example 10.1: read and write char data bytes
*/
#include "mbed.h"
Serial pc(USBTX,USBRX);           // setup terminal link
LocalFileSystem local("local");   // define local file system
int write_var;
int read_var;                     // create data variables

int main () {
  FILE* File1 = fopen("/local/datafile.txt","w");    // open file
    write_var=0x23;                                  // example data
    fputc(write_var, File1);                         // put char (data value) into file
    fclose(File1);                                   // close file

  FILE* File2 = fopen ("/local/datafile.txt","r");   // open file for reading
    read_var = fgetc(File2);                         // read first data value
    fclose(File2);                                   // close file
    pc.printf("input value = %i \n",read_var);       // display read data value
}
```

Program Example 10.1 Saving data to a file

Create a new project and add the code in Program Example 10.1. Run the program, and verify that the data file is created on the mbed and read back correctly. If you navigate to and open the file **datafile.txt** in a standard text editor program (such as Microsoft Wordpad), you should see a hash character (#) in the top left corner. This is because the ASCII character for 0x23 is the hash character (recall Table 8.3).

Exercise 10.1

Change Program Example 10.1 to experiment with other data values; check these against their associated ASCII codes. Write the numbers 1–10 and view them on the screen.

■

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

Note that when the microcontroller program opens a file on the local drive, the mbed will be marked as 'removed' on a host PC. This means the PC will often display a message such as 'insert a disk into drive' if you try to access the mbed at this time; this is normal, and stops both the mbed and the PC trying to access the USB disk at the same time. Note also that the USB drive will only reappear when all file pointers are closed in your program, or the microcontroller program exits. If a running program on the mbed does not correctly close an open file, you will no longer be able to see the USB drive when you plug the mbed into your PC. It is therefore important for a programmer to take care when using files to ensure that all files are closed when they are not being used.

If a running program on the mbed does not exit correctly, to allow you to see the drive again (and load a new program), use the following procedure:

1. Unplug the mbed.
2. Hold the mbed reset button down.
3. While still holding the button, plug in the mbed. The mbed USB drive should appear on the host computer screen.
4. Keep holding the button until the new program is saved onto the USB drive.

10.3.2 String File Access

Program Example 10.2 creates a file and writes text data to that file. The file is saved on the mbed. The program then opens and reads back the text data and displays it to the screen in a host terminal application.

```
/* Program Example 10.2: Read and write text string data
*/
#include "mbed.h"
Serial pc(USBTX,USBRX);             // setup terminal link
LocalFileSystem local("local");     // define local file system
char write_string[64];              // character array up to 64 characters
char read_string[64];               // create character arrays (strings)

int main () {
  FILE* File1 = fopen("/local/textfile.txt","w");      // open file access
  fputs("lots and lots of words and letters", File1);  // put text into file
  fclose(File1);                                       // close file

  FILE* File2 = fopen ("/local/textfile.txt","r");     // open file for reading
  fgets(read_string,256,File2);                        // read first data value
  fclose(File2);                                       // close file
  pc.printf("text data: %s \n",read_string);           // display read data string
}
```

Program Example 10.2 Saving a string to a file

Compile and run the program, and verify that the text file is created and read back correctly – if you open the file **textfile.txt**, found on the mbed, the correct text data should be found within.

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

When reading data from a file, the file pointer can be moved with the **fseek( )** function. For example, the following command will reposition the file pointer to the 8th byte in the text file:

```
fseek (File2 , 8 , SEEK_SET ); // move file pointer to byte 8 from the start
```

The **fseek( )** function needs three input terms; first, the name of the file pointer; secondly, the value to offset the file pointer to; and thirdly, an 'origin' term which tells the function where exactly to apply the offset. The term **SEEK_SET** is a predefined origin term (defined in the **stdio** library) which ensures that the 8-byte offset is applied from the start of the file.

Exercise 10.2

Add the following **fseek( )** statement to Program Example 10.2 just prior to the data being read back:

```
fseek (File2 , 8 , SEEK_SET ); // move file pointer to byte 8
```

Verify that Tera Term only displays the data after byte 8 of the data file. Remember byte values increment from zero, so it will actually be after the 9th character in the file.

■

10.3.3 Using Formatted Data

C code feature

It is possible to format data stored in a file. This can be done with the **fprintf( )** function, which has very similar syntax to **printf( )**, except that the filename pointer is also required. We may want, for example, to log specific events to a data file and include variable data values such as time, sensor input data and output control settings. Program Example 10.3 shows use of the **fprintf( )** function in an interrupt controlled push-button project. Each time the push-button is pressed, the light-emitting diode (LED) toggles and changes state. Also on each button press, the file **log.txt** is updated to include the time elapsed since the previous button press, and the current LED state. Program Example 10.3 also implements a simple debounce timer (as described in Section 9.10) to avoid multiple interrupts and file write operations.

```
/* Program Example 10.3: Interrupt toggle switch with formatted data logging to text
file
*/

#include "mbed.h"
InterruptIn button(p30);          // Interrupt on digital input p30
DigitalOut led1(LED1);            // digital out to onboard LED1
Timer debounce;                   // define debounce timer
LocalFileSystem local("local");   // define local file system
void toggle(void);                // function prototype

int main() {
```

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

```
 debounce.start();               // start debounce timer
 button.rise(&toggle);           // attach the toggle function to the rising edge
}
void toggle() {                  // perform toggle if debounce time has elapsed
    if (debounce.read_ms()>200)
    led1=!led1;                                  // toggle LED
    FILE* Logfile = fopen ("/local/log.txt","a"); // open file for appending
    fprintf(Logfile,"time=%.3fs: setting led=%d\n\r",debounce.read(),led1.read());
    fclose(Logfile);                             // close file
    debounce.reset();                            // reset debounce timer
  }
  }
```

Program Example 10.3 Push-button LED toggle with formatted data logging

Note that the text file **log.txt** may not display full formatting in a simple text viewer such as Microsoft Notepad. If line breaks are not displaying correctly then try a more advanced text file viewer such as Microsoft Wordpad.

Exercise 10.3

Create a program which prompts the user to type some text data into a terminal application. When the user presses return the text is captured and stored in a file on the mbed.

Ensure that the data is correctly written to the data file by opening it with a standard text viewer program.

10.4 Using External Memory with the mbed

A flash SD (secure digital) card can be used with the mbed via the serial peripheral interface (SPI) protocol, as described in Reference 10.3. Using a micro SD card with a card holder cradle (as shown in Figure 10.3), it is possible to access the SD card as an external memory.

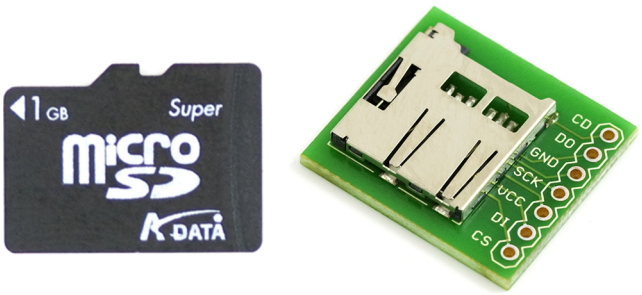

Figure 10.3:
An SD card with holder. *(Image reproduced with permission of SparkFun Electronics)*

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

Table 10.3: Connections for SPI access to the SD card

MicroSD breakout	mbed pin
CS	8 (DigitalOut)
DI	5 (SPI mosi)
Vcc	40 (Vout)
SCK	7 (SPI sclk)
GND	1 (GND)
DO	6 (SPI miso)
CD	No connection

Table 10.4: Library files and import paths for implementing the SD card interface

Library	Import path
SDFileSystem	http://mbed.org/users/simon/programs/SDFileSystem/5yj8f
FATFileSystem	http://mbed.org/projects/libraries/svn/FATFileSystem/trunk?rev=29

The SD card can be configured into SPI communication mode, which requires the serial connections described in Table 10.3. This example uses the mbed SPI port on pins 5, 6 and 7 and an arbitrary digital output on pin 8 to act as the SPI chip select signal.

To implement the SD card interface, it is necessary to import the mbed libraries shown in Table 10.4.

Having imported the described files and libraries, and connected the SD card as suggested, Program Example 10.4 writes a test text file to the card.

```
/* Program Example 10.4: writing data to an SD card
*/
#include "mbed.h"
#include "SDFileSystem.h"
SDFileSystem sd(p5, p6, p7, p8, "sd");              // MOSI, MISO, SCLK, CS
Serial pc(USBTX, USBRX);

int main() {
  FILE *File = fopen("/sd/sdfile.txt", "w");        // open file
  if(File == NULL) {                                // check for file pointer
    pc.printf("Could not open file for write\n");    // error if no pointer
  }
  else{
    pc.printf("SD card file successfully opened\n"); // if pointer ok
  }
  fprintf(File, "Here's some sample text on the SD card");  // write data
  fclose(File);                                             // close file
}
```

Program Example 10.4 Writing data to an SD card

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

Compile Program Example 10.4 and verify that the SD card is correctly accessed by viewing the created text file **sdfile.txt** in a standard text editor program.

Program Example 10.4 requires an enhanced definition statement for the **SDFileSystem** object, which in this case is named **sd**. Within the interface definition, `SDFileSystem sd(p5, p6, p7, p8, "sd")`, we see that not only are the SPI interface connection pins defined, but the name of the **SDFileSystem** object is also defined as an input variable in quotation marks. Notice that the line:

```
if(File == NULL) {
```

effectively performs an error check to ensure that the file was opened correctly by the previous **fopen( )** call. If the file pointer **File** has a **NULL** value then it means it has not been created and the **fopen( )** call was not successfully implemented.

Exercise 10.4

Create a program which records 5 seconds of analog data to the screen and to a text file on an SD card. Use a potentiometer to generate the analog input data with sample period of 100 ms. Ensure that your data file records the elapsed time and voltage data.

You can then open your data file from within a standard spreadsheet application, such as Microsoft Excel. This will enable you to plot a chart and to visualize the analog data.

10.5 Introducing Pointers

Pointers are used in C/C++ to indicate where a particular element or block of data is stored in memory. Pointers are also discussed in Section B.8.2. We will look here in a little more detail at a specific mbed example using functions and pointers.

When a pointer is defined it can be set to a particular memory address and C/C++ syntax allows the data at that address to be accessed. Pointers are required for a number of reasons; one is because the C/C++ standard does not allow us to pass arrays of data to and from functions, so we must use pointers instead. In some programming languages (such as Matlab), it is possible to develop a mean average calculation function which reads in a data array, calculates the average from the sum of the data divided by the number of data values and returns the calculated mean. Using the mbed, however, we cannot pass the array of data into the function, so instead we need to pass the pointer value which points to the data array.

Pointers are defined similarly to variables but by additionally using the * operator. For example, the following declaration defines a pointer called **ptr** which points to data of type **int**:

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

```
int *ptr;       // define a pointer which points to data of type int
```

The specific address of a data variable can also be assigned to a pointer by using the **&** operator, for example:

```
int datavariable=7;   // define a variable called datavariable with value 7
int *ptr;             // define a pointer which points to data of type int
ptr = &datavariable;  // assign the pointer to the address of datavariable
```

In program code the * operator can also be used to get the data from the given pointer address, for example:

```
int x = *ptr;     // get the contents of location pointed to by ptr and
                  // assign to x (in this case x will equal 7)
```

Pointers can also be used with arrays, because an array is really just a number of data values stored at consecutive memory locations. So if the following is defined:

```
int dataarray[]={3,4,6,2,8,9,1,4,6}; // define an array of arbitrary values
int *ptr;                            // define a pointer
ptr = &dataarray[0];                 // assign pointer to the address of
                                     // the first element of the data array
```

the following statements will therefore be true:

```
*ptr == 3;            // the first element of the array pointed to
*(ptr+1) == 4;        // the second element of the array pointed to
*(ptr+2) == 6;        // the third element of the array pointed to
```

So array searching can be done by moving the pointer value to the correct array offset. Program Example 10.5 implements a function for analyzing an array of data and returns the average of that data.

```
/* Program Example 10.5: Pointers example for an array average function
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);                                  // setup serial comms
char data[]={5,7,5,8,9,1,7,8,2,5,1,4,6,2,1,4,3,8,7,9}; //define some input data
char *dataptr;              // define a pointer for the input data
float average;              // floating point average variable

float CalculateAverage(char *ptr, char size);   // function prototype

int main() {
  dataptr=&data[0];    // point pointer to address of the first array element
  average = CalculateAverage(dataptr, sizeof(data));   // call function
  pc.printf("\n\rdata = ");
  for (char i=0; i<sizeof(data); i++) {      // loop for each data value
    pc.printf("%d ",data[i]);               // display all the data values
  }
  pc.printf("\n\raverage = %.3f",average);    // display average value
}
// CalculateAverage function definition and code
float CalculateAverage(char *ptr, char size) {
```

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

```
  int sum=0;          // define variable for calculating the sum of the data
  float mean;         // define variable for floating point mean value
  for (char i=0; i<size; i++) {
    sum=sum + *(ptr+i);       // add all data elements together
  }
  mean=(float)sum/size;        // divide by size and cast to floating point
  return mean;
}
```

Program Example 10.5 Averaging function using pointers

C code feature

Looking at some key elements of Program Example 10.5, we can see that the pointer **dataptr** is assigned to the address of the first element of the **data** array. The **CalculateAverage( )** function takes in a pointer value that points to the first value of the data array and a second value that defines the size of the array. The function returns the floating point mean value. There is also an additional C/C++ keyword used here, **sizeof**, which deduces the size of a particular array. This gets the size (i.e. the number of data elements) in the array **data**. Also note that the calculation of the mean value is in the form of an integer divided by a char, yet we want the answer to be a floating point value. To implement this, we *cast* the equation as floating point by using **(float)**, which ensures that the resultant mean value is to floating point precision.

Pointers are used for a number of reasons, especially in C++ programs which rely heavily on functions, methods and passing data between these. Pointers can also be used to improve programming efficiency and speed, by directly accessing memory locations and data. In general, the use of pointers in the programming seen in this book is required because of a deficiency in the C/C++ capabilities, i.e. not being able to pass arrays to functions. Where possible, we try to avoid the use of pointers in order to keep code simple and readable, but sometimes they offer the best solution to a programming challenge.

10.6 Mini-Project: Accelerometer Data Logging on Exceeding Threshold

In this mini-project you are challenged to create a program that records the acceleration profile encountered in a simple vibrating cantilever. It is then possible to plot the acceleration data as shown in Figure 10.4. This project develops from the mini-project in Section 9.11. Design and implement a new project to the following specification:

1. Enable the mbed to access external SD card memory.
2. Attach a SPI accelerometer to a simple plastic cantilever with a flying lead to the mbed.
3. Program the accelerometer to cause an interrupt trigger when excessive acceleration is encountered.
4. Create an interrupt routine to log 100 data samples to a file on the SD card. You may wish to use the **fprintf( )** function to format accelerometer data in the text file.

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

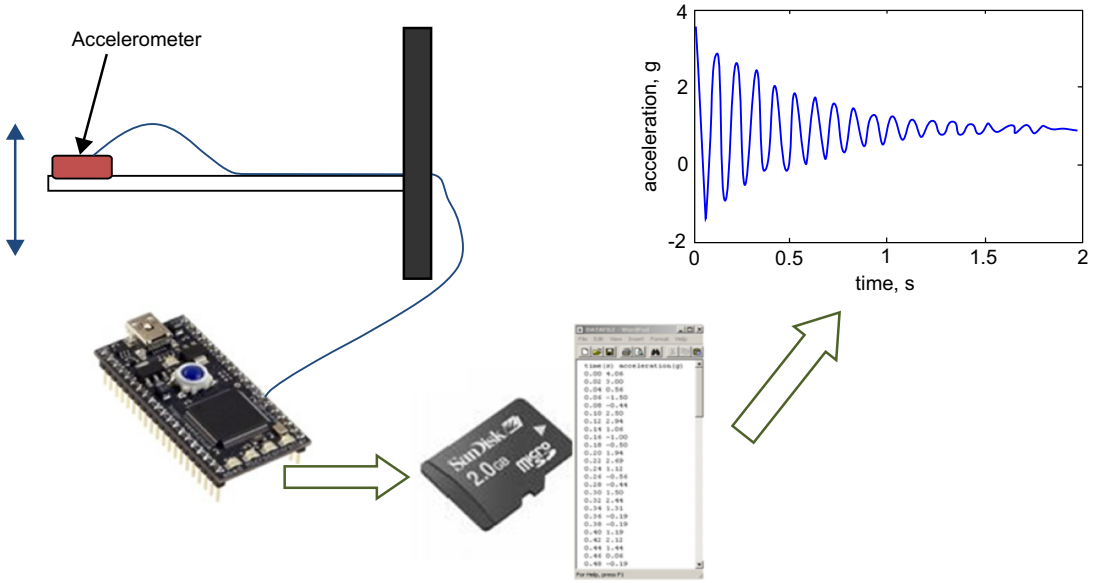

Figure 10.4:
Accelerometer data-logging mini-project

5. When a certain acceleration threshold is exceeded, the data should be logged. The text file can then be opened in a spreadsheet software program, such as Microsoft Excel, to plot the recorded acceleration waveform.

Note: to achieve a suitable sample period you may want to apply an mbed Ticker or Timer to ensure that the accelerometer logs data regularly. A sampling frequency of around 50 Hz should be sufficient to record a detailed acceleration waveform.

Chapter Review

- Microprocessors use memory for holding the program code (program memory) and the working data (data memory) in an embedded system.
- A coin or a logic flip-flop/bistable can be thought of as a single 1-bit memory device which retains its state until the state is actively changed.
- Volatile memory loses its data once power is removed, whereas non-volatile can retain memory with no power. Different technologies are used to realize these memory types, including SRAM and DRAM (volatile) and EEPROM and flash (non-volatile).
- The mbed has 512 kb of flash and 64 kb of SRAM in the LPC1768 IC, and a further 16-megabit USB memory area.
- Files can be created on the mbed for storing and retrieving data and formatted text.
- An external SD memory card can be interfaced with the mbed to allow larger memory.

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.

- Pointers point to memory address locations to allow direct access to the data stored at the pointed location.
- Pointers are generally required owing to the fact that C/C++ does not allow arrays of data to be passed into functions, so a pointer to the array data must be passed instead.

Quiz

1. What does the term *bistable* mean?
2. How many bistables would you expect to find in the mbed's SRAM?
3. What are the fundamental differences between SRAM and DRAM type memory?
4. What are the fundamental differences between EEPROM and flash type memory?
5. Which C/C++ command would open a text file for adding additional text to the end of the current file.
6. Which C/C++ command should be used to open a text file called 'data.txt' and read the 12th character?
7. Give a practical example where data logging is required and explain the practical requirements with regard to timing, memory type and size.
8. Give one reason why pointers are used for direct manipulation of memory data.
9. Give the C/C++ code that defines an empty five-element array called *dataarray* and a pointer called *datapointer* which is assigned to the first memory address of the data array.
10. How is a pointer used to access and manipulate the different elements of a data array?

References

10.1. Grindling, G. and Weiss, B. (2007). Introduction to Microcontrollers. https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/theory-material/Microcontroller.pdf
10.2. C++ stdio.h fopen reference. http://www.cplusplus.com/reference/clibrary/cstdio/fopen/
10.3. SD Group (Matsushita Electric Industrial Co. and SanDisk Corporation) (2006). SD Physical Layer Simplified Specification, Version 2.00. http://www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf

Copyright © 2012. Elsevier Science & Technology. All rights reserved.

Toulson, Rob, and Tim Wilmshurst. Fast and Effective Embedded Systems Design : Applying the ARM Mbed, Elsevier Science & Technology, 2012.
ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=947406.
Created from hud on 2019-11-24 09:57:44.