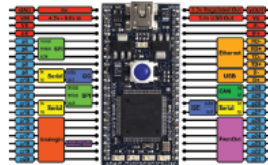




FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN

Applying the
ARM mbed



Second Edition

Rob Toulson and Tim Wilmshurst



Chapter 10: Memory and Data Management

rt rev. 12.9.16

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

Toulson, R. & Wilmshurst, T. (2016). Fast and Effective Embedded Systems Design - Applying the ARM mbed (2nd edition), Newnes, Oxford, ISBN: 978-0-08-100880-5.

www.embedded-knowhow.co.uk

Memory function types

A microprocessor needs memory for two reasons: to hold its program, and to hold the data that it is working with; we often call these *program memory* and *data memory*. To meet these needs there are a number of different semiconductor memory technologies available, which can be embedded on the microcontroller chip.

Memory technology is divided broadly into two types: volatile and non-volatile.

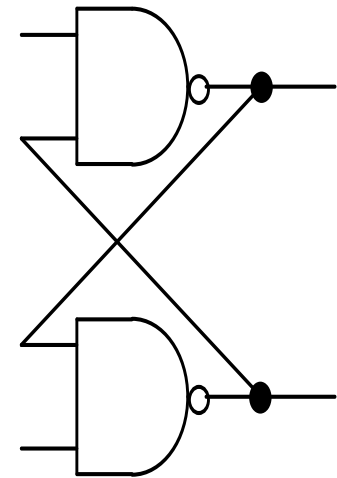
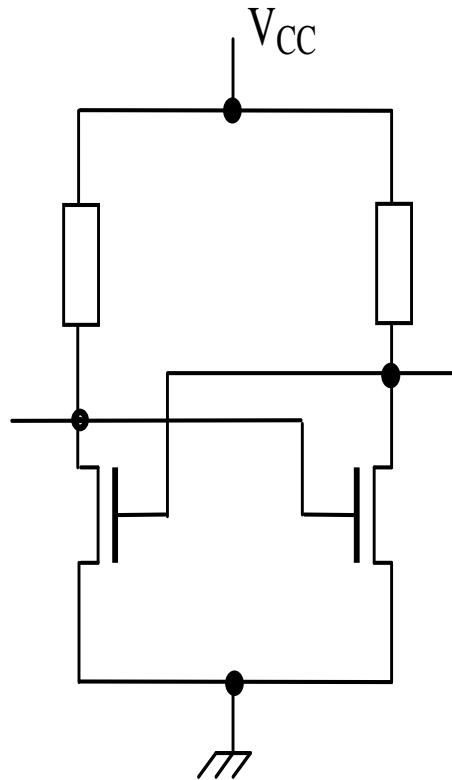
- **Non-volatile memory** retains its data when power is removed, but tends to be more complex to write to in the first place. For historical reasons it is still often called ROM (Read Only Memory). Non-volatile memory is generally required for program memory, so that the program data is there and ready 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's 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).

Essential electronic memory types

- In any electronic memory we want to be able to store all the 1s and 0s which make up our data.
- 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.
- We recognise that the coin is stable in two states, we call this *bistable*.
- It could be said that “heads” represents logic 1, and “tails” logic 0. With 8 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 a number of electronic alternatives to the coin, which take up much less space.
- One is to use an electronic bistable (or “flip-flop”) circuit.

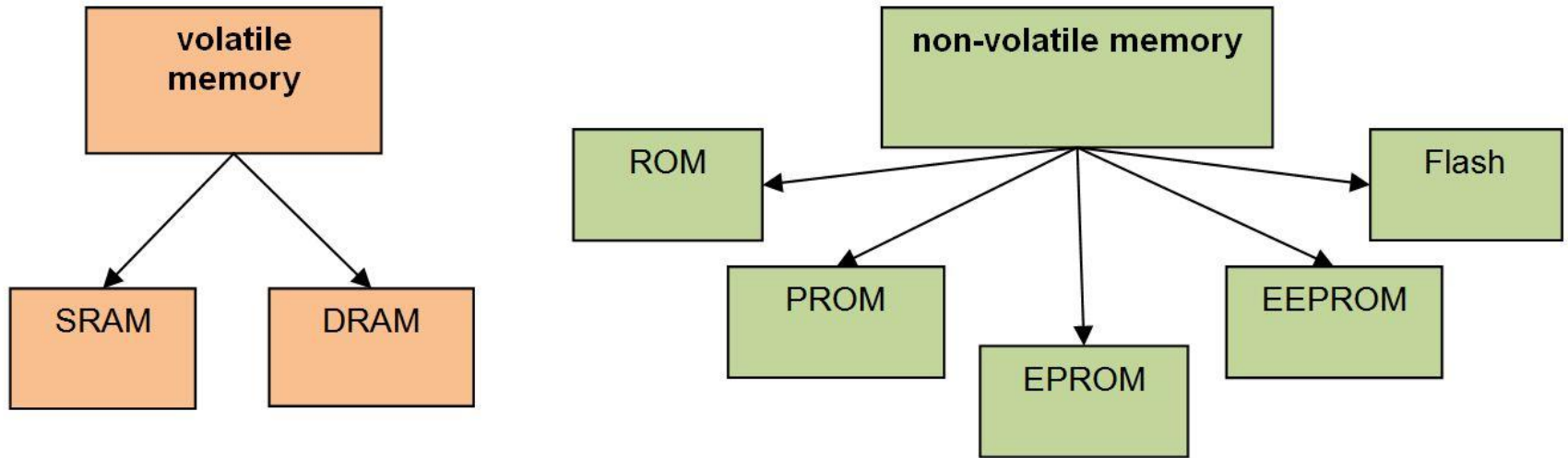
Essential electronic memory types

The two logic circuits shown are stable in only two states, and each can be used to store one bit of data.



Essential electronic memory types

There are a number of different types of volatile and non-volatile memory:



RAM	Random Access Memory
SRAM	Static RAM
DRAM	Dynamic RAM

ROM	Read Only Memory
PROM	Programmable ROM
EPROM	Electrically Programmable ROM
EEPROM	Electrically Erasable PROM
FLASH	A cheap and rapid form of EEPROM

Introducing pointers

Before delving into the embedded world of memory and file access, we first need to cover a little C/C++ background on *pointers*.

Pointers are used to indicate where a particular element or block of data is stored in memory.

When a pointer is defined it can be set to a particular memory address and C/C++ syntax allows us to access the data at that address.

Pointers are required for a number of reasons; one is because the C/C++ standard does not allow arrays of data to be passed to and from functions, so in this case we must use pointers instead.

For example, we may wish to pass an array of 10 data values to a function in order to perform a simple mean average calculation, but in C/C++ this is not possible and causes a syntax error if attempted. Instead, to achieve this, it is necessary to pass a single pointer value as an input argument to the function.

In this instance the pointer essentially describes the memory address of the first element of the data array and is usually accompanied by a single argument that defines the size of the data array in question.

Defining pointers

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

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

We can also use the `*` operator in a program to get the data from a pointer address, for example:

```
int x = *ptr;    // get the contents of location ptr and assign to x
```

We can also use pointers with arrays, because an array is really just a number of data values stored at consecutive memory locations; for example:

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

Using pointers with arrays and functions

```
/* 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}; // 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 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) {
    int sum=0;                          // variable for calculating the sum of the data
    float mean;                         // 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;
}
```


Useful C/C++ library functions for data control

Function	Format	Summary Action
fopen	<code>FILE *fopen(const char *filename, const char *mode);</code>	opens the file of name filename
fclose	<code>int fclose(FILE *stream);</code>	closes a file
fgetc	<code>int fgetc(FILE *stream);</code>	gets a character from a stream
fgets	<code>char *fgets(char *str, int n, FILE *stream);</code>	gets a string of n chars from a stream
fputc	<code>int fputc(int character, FILE *stream);</code>	writes character to a stream
fputs	<code>int fputs(const char *str, FILE *stream);</code>	writes a string to a stream
fprintf	<code>int fprintf(FILE *stream, const char *format, ...);</code>	writes formatted data to a stream
fseek	<code>int fseek(FILE *stream, long int offset, int origin);</code>	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 data is to be written (see Section B10 for more information on streams in C/C++).

... Indicates that additional formatted arguments may be specified in a list

Using data files on the mbed

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 following line to the declarations section of a program:

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

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");
```

(note the “w” refers to a file with write access)

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

```
fclose(pFile);
```

Example mbed data file access

```
/* 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
}
```

String file access

```
/* 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
}
```

Read further:
Writing formatted data,
Section 10.4.3

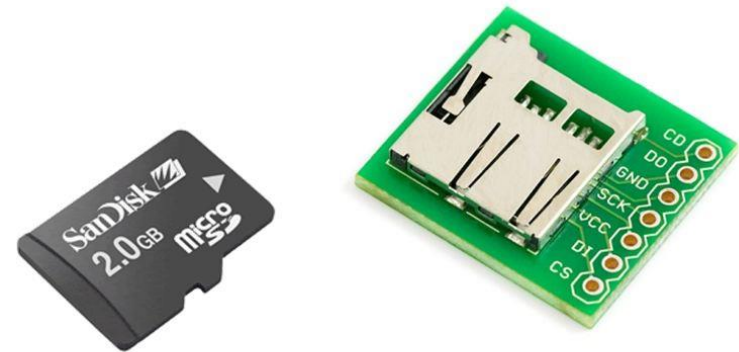
Using external memory with the mbed

A flash SD (Secure Digital) card can be used with the mbed via the SPI protocol

Using a micro SD card with a card holder cradle (as shown) it is possible to access the SD card as an external memory

The SD card can be connected to the mbed as shown in the following wiring table:

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



The following mbed libraries are also required:

SDFileSystem library by Simon Ford.

<https://developer.mbed.org/users/simon/code/SDFileSystem>

FatFileSystem by mbed.

https://developer.mbed.org/users/mbed_unsupported/code/FatFileSystem

Using external USB flash memory with the mbed



The mbed **USBHost** library includes a class called **USBHostMSD**, which is specifically intended to allow the mbed to utilise an external flash mass storage device (MSD) on the USB bus.

This makes it simple and convenient to use standard USB flash drives that can hold many gigabytes of data.

Conveniently the mbed application board has built-in USB connectors attached to pins 31 and 32, with on-board pull-down resistors in place.

When operating in USB host mode, the mini USB selector switches on the application board need setting accordingly.

Insert a flash drive into the USB Type A connector, as shown.



Using external USB flash memory with the mbed



Program Example 10.5 implements the USB flash storage.

In order for the program to compile, the mbed official **USBHost** library will need to be imported to the project.

Note that the **USBHost** library contains a number of sub-libraries within, and you should be careful to check that every library is the most recent version.

```
/* Program Example 10.5: writing data to an USB flash storage device
*/

#include "mbed.h"
#include "USBHostMSD.h"

int main() {

    USBHostMSD usb("usb"); // define USBHostMSD object

    while(!usb.connect()) { // try to connect a USB storage before continuing
        wait(0.5);
        printf("Connecting to USB MSD\n");
    }

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


Chapter quiz questions

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. Describe the purpose of pointers and explain how they used to access the different elements of a data array.
6. What C/C++ command would open a text file for adding additional text to the end of the current file.
7. What C/C++ command should be used to open a text file called "data.txt" and read the 12th character.
8. Give a practical example where data logging is required and explain the practical requirements with regards to timing, memory type and size.
9. Give one reason why pointers are used for direct manipulation of memory data.
10. Write the C/C++ code that defines an empty 5 element array called **dataarray** and a pointer called **datapointer** which is assigned to the first memory address of the data array.

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. A number of different technologies are used to realise these memory types including SRAM and DRAM (volatile) and EEPROM and Flash (non-volatile).
- The LPC1768, on the mbed, has 512 KB of Flash memory, and 64 KB of SRAM.
- 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
- The **stdio.h** library contains functions that allow us to create, open and close files, as well as read data from and write data to files.
- Files can be created on the mbed for storing and retrieving data and formatted text.
- An external SD memory card or USB Flash drive can be interfaced with the mbed to allow larger memory.

Read further:

Try out the data logging mini-project described in Section 10.7