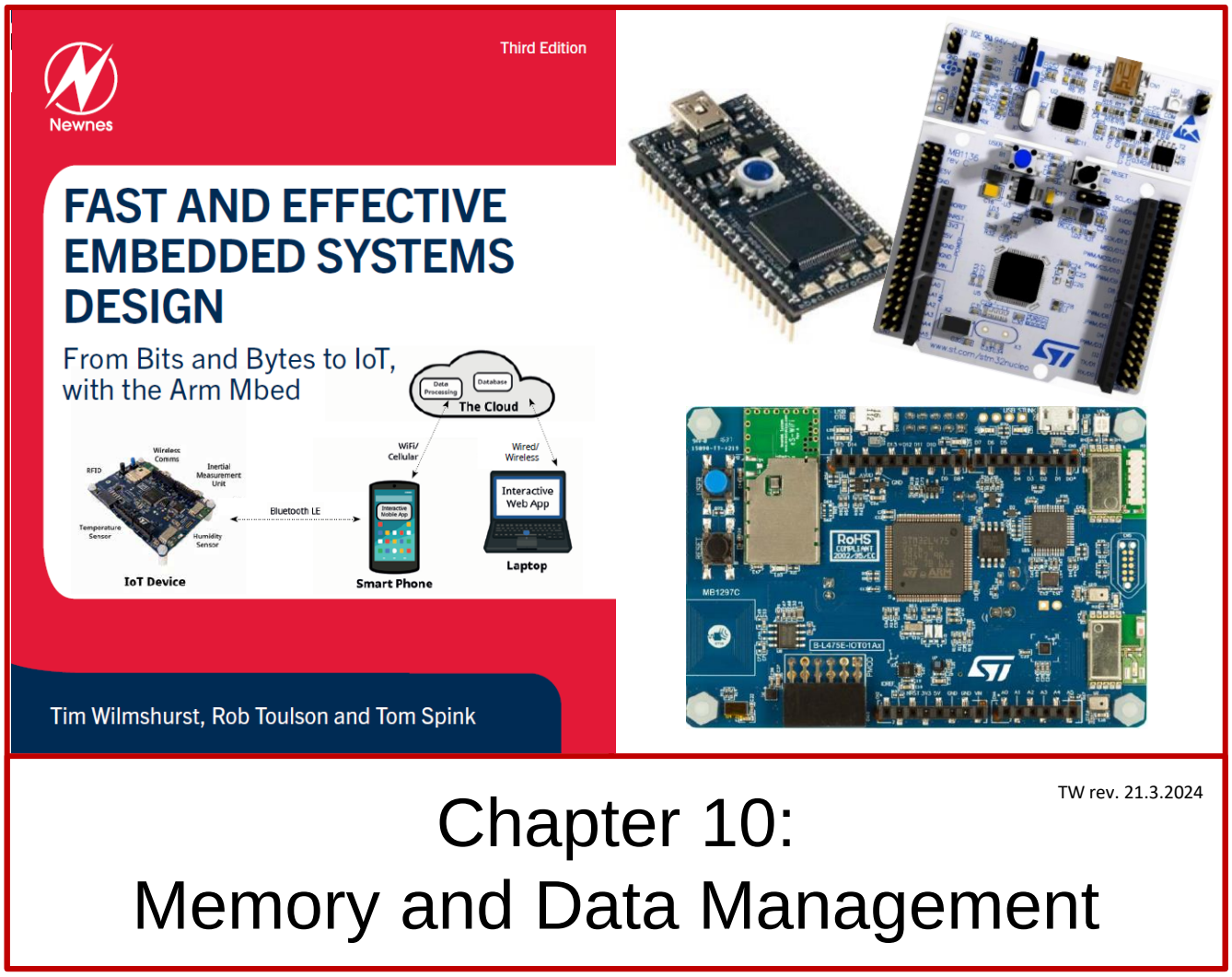


TW rev. 21.3.2024

Chapter 10: Memory and Data Management

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

Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

www.embedded-knowhow.co.uk

Memory 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.

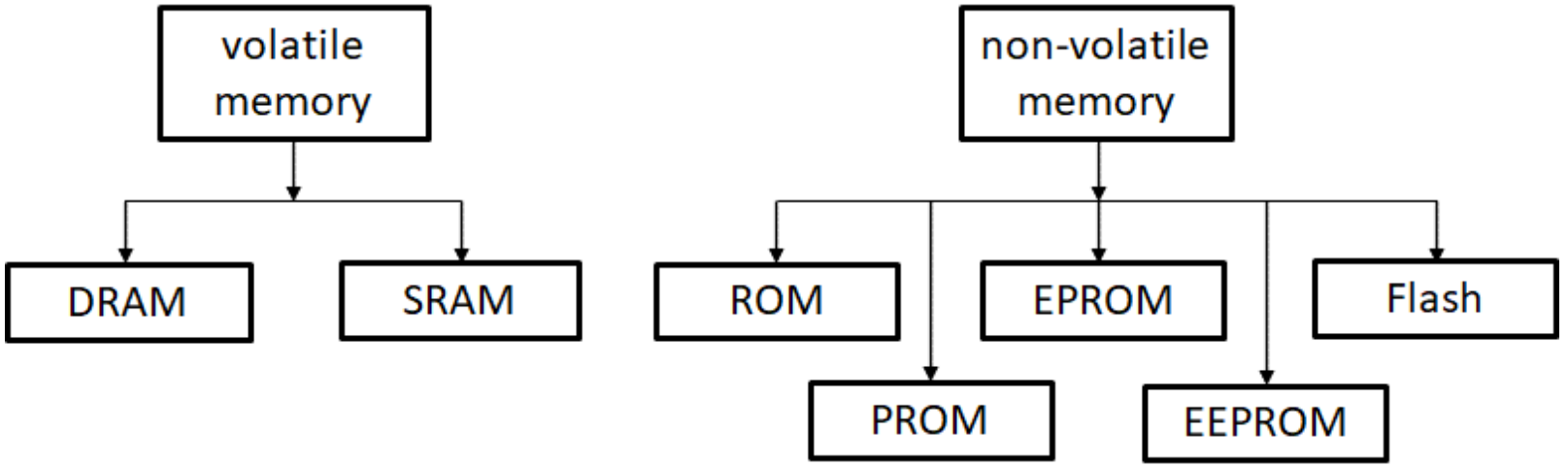
Memory technology is divided broadly into two types: volatile and non-volatile.

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

- **Volatile memory** loses all data when power is removed, but is easy to write to. This 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

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



DRAM: Dynamic Random Access Memory
EPROM: Erasable Programmable Read Only Memory
EEPROM: Electrically Erasable Programmable Read Only Memory
PROM: Programmable Read Only Memory
ROM: Read Only Memory
SRAM: Static Random Access Memory

Electronic Memory Types

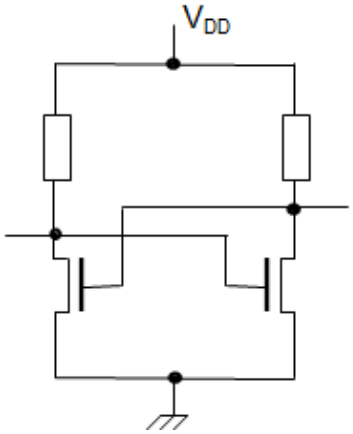
- In any electronic memory we want to be able to store all the 1s and 0s which make up our data.
- A simple one-bit memory is a coin. It is stable in two positions, with either “heads” facing up, or “tails”.
- We can try to balance the coin on its edge, but it would pretty soon fall over.
- We recognise 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.

Electronic Memory Types

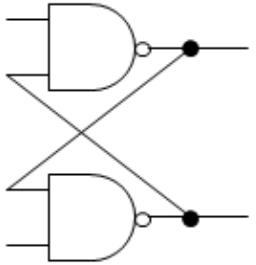
The main memory types are shown here. The characteristics of a single cell are then reflected in the characteristics of a whole memory array, made from that technology.



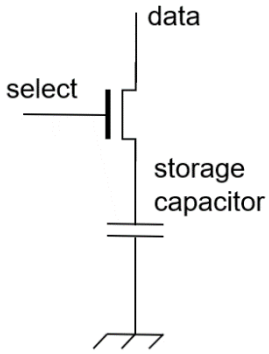
(a) a coin



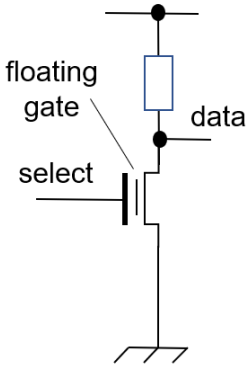
(b) Bistable, two transistors and two resistors



(c) Bistable, two NAND gates



(d) charged capacitor



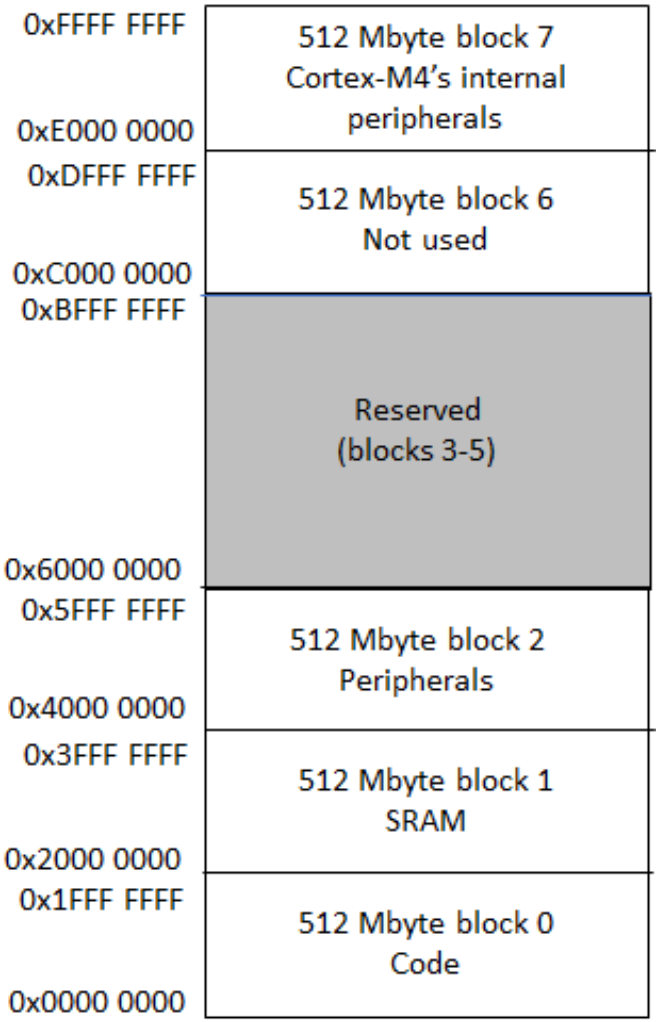
(e) floating gate

Memory Mapping

When a mass of memory has been embedded onto the microcontroller IC, each memory location, typically of one byte, is allocated a binary address. This is a unique number by which that location is identified, and hence accessed while a program runs.

Different parts of memory are allocated different places in the resulting “memory map”. To these are added the registers which control the peripherals, and possibly other features of the microcontroller.

A simplified version of the F401RE microcontroller memory map, itself an application of the Arm Cortex-M memory map, is shown here.



Introducing 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, 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 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
```

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

This program example implements a function for calculating the average of an array. It requires no external connections, and can be run on any Mbed platform, returning display data to a serial monitor.

The pointer **dataptr** is assigned to the address of the first element of the **data** array.

```
/* Program Example 10.1: Pointers example for an array averaging function
                                                                        */
#include "mbed.h"

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
  printf("\n\rdata = ");
  for (char i = 0; i < sizeof(data); i++) {     // loop for each data value
    printf("%d ",data[i]);                      // display all the data values
  }
  printf("\n\raverage = %.3f",average);         // display average value
  printf("\n\rfinished”);
}
//continued over
```

Using Pointers with Arrays and Functions

```
     // CalculateAverage function definition and code
float CalculateAverage(char *ptr, char size) {
  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;
}
```

a) the main.cpp file

```
{
    "target_overrides":{
        "*": {
            "target.printf_lib": "std"
        }
    }
}
```

b) the mbed_app.json file

The **CalculateAverage( )** function takes in a pointer value which points to the first value of the data array and a second value which defines the size of the array. The function returns the floating point mean value. The C/C++ keyword, **sizeof,** deduces the size (i.e. the number of data elements) of a particular array.

File Management in C/C++

Simple arrays, strings or other data groupings are useful in their own way, but we often need to hold much larger data collections, maybe of diverse forms and uncertain size.

Here we turn to files and filing systems. In C/C++ we can create files, open and close them, read and write data from or to them, and scan them in search of particular data.

Functions for input and output operations are defined by the C Standard Input and Output Library **stdio**. Using **stdio**, we can store data in files (as **chars**) or as strings of text.

Example data storage functions appear in the Table in the next slide.

The Table refers to data streams. These are a flow of data, essentially a sequence of bytes, linked to an input or output. Streams are represented in the **stdio** library as pointers to FILE objects, which uniquely identify the stream.

File Management in C/C++

Function	Format	Summary Action
fopen	`FILE *fopen(const char *filename, const char *mode)`	Opens the file of name *filename* with chosen mode.
fclose	`int fclose(FILE *stream)`	Closes a file.
fflush	`int fflush(FILE *stream)`	Writes any unwritten data in file buffer to the file.
fgetc	`int fgetc(FILE *stream)`	Gets a character from a stream.
fgets	`char *fgets(char *str, int n, FILE *stream)`	Gets a string of **n** chars from a stream.
fputc	`int fputc(int character, FILE *stream)`	Writes *character* to a stream.
fputs	`int fputs(const char *str, FILE *stream)`	Writes a string to a stream.
fprintf	`int fprintf(FILE *stream, const char *format, ...)`	Writes formatted data to a stream.
fseek	`int fseek(FILE *stream, long int offset, int origin)`	Moves file pointer to specified location, applying: SEEK_CUR: current pointer posn. SEEK_END: end of file SEEK_SET: start of file

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

The Mbed LPC1768 Local File System

The **LocalFileSystem** library allows the programmer to set up a local file system for accessing the Mbed LPC1768 USB disk. It allows programs to read and write files in this memory, which can then be accessed from the host computer. Once the system has been set up, the standard C/C++ **stdio** file access functions – as seen earlier - can be used to open, read and write 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);
```

Local File System Access

This program example 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 10.2: read and write data bytes to Mbed USB disk
        Runs only on LPC1768                                          */

#include "mbed.h"
LocalFileSystem local("local");       // define local file system
int write_var, read_var;              // create data variables

int main (){
  // open file with name datafile.txt, in write mode
  FILE* File1 = fopen("/local/datafile.txt","w");
  write_var = 0x23;              // example data, 0x23 is ASCII code for #
  fputc(write_var, File1);       // put char (data value) into file
  fclose(File1);                 // close file

  File1 = fopen ("/local/datafile.txt","r");  // open file for reading
  read_var = fgetc(File1);                    // read first data value
  fclose(File1);                              // close file
  printf("input value=%i \n",read_var);       // display read data value
}
```

Local File System: String File Access

Using the **stdio** functions, it is possible to read and write words and strings and search or move through files looking for particular data elements. This program example creates a file and writes a string of text data to it. The file is saved on the Mbed LPC1768 target. The program then opens and reads back the text data and displays it to the screen in a host terminal application.

```
/* Program Example 10.3: Read and write text string data to Mbed USB disk
   Runs only on LPC1768                                                  */

#include "mbed.h"
LocalFileSystem local("local");   //define local file system
char write_string[64];            //character array up to 64 chars
char read_string[64];             //character array up to 64 chars)

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

  File1 = fopen ("/local/textfile.txt","r");  //open file for reading
  fgets(read_string,256,File1);               //read 256 chars of data
  fclose(File1);                              //close file
  printf("text data: %s \n",read_string);     //display read data string
}
```

Local File System: Using Formatted Data (1)

Logging of formatted data can be done with the **fprintf( )** function, with similar syntax to **printf( )**, except that the filename pointer is also required. This program example uses the **fprintf( )** function in a simple interrupt-driven data logging project.

On each button press the file **log.txt** is updated to include the time elapsed since the previous button press, the current LED state, and a time stamp from the RTC.

```
/* Program Example 10.4: Formatted data logging to text file triggered by button
 push, with time stamp.
Runs on LPC1768 only
                                                                              */
#include "mbed.h"
InterruptIn button(p5);          //Interrupt on digital input p5
DigitalOut led1(LED1);           //digital out to onboard LED1
Timer main_timer;                //define elapsed time timer
LocalFileSystem local("local");  //define local file system
bool button_pressed;             //flags button has been pressed
void toggle(void);               //function prototype

using namespace std::chrono;
//continued on next slide
```

Local File System: Using Formatted Data (2)

```
int main() {
  set_time(1666697756);   //Set RTC time (this is Tues, 25 Oct 2022 11:35:56)
  button.rise(&toggle);   //attach the toggle function to the rising edge
  button_pressed = 0;
  main_timer.start();
  while(true){
    if(button_pressed==1){
      //open a text file named log, append mode
      FILE* Logfile = fopen ("/local/log.txt","a");
      //print elapsed time from Timer, and state of led
      fprintf(Logfile,"Elapsed time=%u ms, ",duration_cast<milliseconds>
                                          (main_timer.elapsed_time()).count());
      if(led1.read() == 1)fprintf(Logfile,"led is 1\n\r");
      else fprintf(Logfile,"led is 0\n\r");
      time_t seconds = time(NULL);   //return elapsed seconds
      fprintf(Logfile,"Time of reading = %s\n\n", ctime(&seconds));
      fclose(Logfile);                              // close file
      button_pressed = 0;
    }          //end of if
  }            //end of while
}
//this is ISR
void toggle() {
    wait_us(5000);        //simple debounce to settle
    if (button == 1){
      led1 = !led1;       // toggle LED
      button_pressed = 1;
    }
}
```

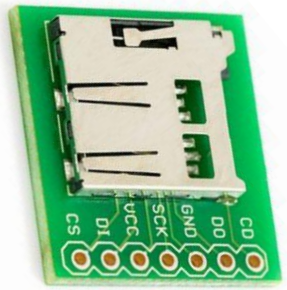
Using External SD Card Memory

A flash SD (Secure Digital) card can be accessed via the SPI protocol, as an external memory. The card can be placed in a card holder cradle (as shown).

The SD card can be connected to one of the Mbed targets as shown in the Table.

MicroSD Breakout	Mbed LPC1768 pin	Nucleo F401RE pin
CS	8 (DigitalOut)	D6
DI	5 (SPI MOSI)	D4
Vcc	40 (Vout)	+3V3
SCK	7 (SPI SCLK)	D3
GND	1 (GND)	GND
DO	6 (SPI MISO)	D5
CD	No connection	No connection





Using External SD Card Memory

The SD card is a “block” device, and uses the **SDBlockDevice** API, summarized in the Table.

Functions	Usage
Constructor	
`SDBlockDevice (PinNames mosi, miso, sclk, cs, uint64_t hz=1000000, bool crc_on=0)`	Creates an SD Block Device on SPI bus specified by pins
Member Functions	
`virtual int deinit()`	Deinitialise a block device
`virtual int erase(bd_addr_t addr, bd_size_t size)`	Erase block and prepare for writing; addr is start of erase block, of size size
`virtual int frequency(uint64_t freq)`	Set transfer frequency, freq.
`virtual bd_size_t get_erase_size (bd_addr_t addr)`	Gets the size (by return) of an erasable block
`virtual mbed::bd_size_t get_program_size()`	Gets the size (by return) of programmable block
`virtual mbed::bd_size_t get_read_size()`	Gets the size (by return) of readable block
`virtual int init()`	Initialise a block device
`virtual int program(const void* buffer, mbed::bd_addr_t addr, mbed::bd_size_t size)`	Program blocks to a block device. The blocks must be erased prior to programming. buffer = buffer to write to, addr = address of block, size = size to read
`virtual int read(void* buffer, mbed::bd_addr_t addr, mbed::bd_size_t size)`	Read block, parameters as above.

Using an SD Card (1)

This program example detects card data, and prints it to the PC screen. It then writes a test text file to the card, reads this back, and displays. The **mbed_app.json** file must also be created; this works for both targets, as the **SDBlockDevice** initialization within the program overwrites pin allocations in the **json** file.

```
/*Program Example 10.5. Writing data to/from SD card.
Works on LPC1768 and Nucleo F401RE, with pin connections shown
*/

#include "mbed.h"
#include "SDBlockDevice.h"

SDBlockDevice serdev(D4, D5, D3, D6);      //use p5,p6,p7,p8 resp. for LPC1768

uint8_t block[512] = "Hello World!\n";    //the string to be stored
uint8_t retrieve[512];                    //data block to store retrieved data

int main(){
  // Initialise the SDBlockDevice
  if (0 != serdev.init()) {
    printf("Init failed \n");
    return -1;
  }
  //continued on next slide
```

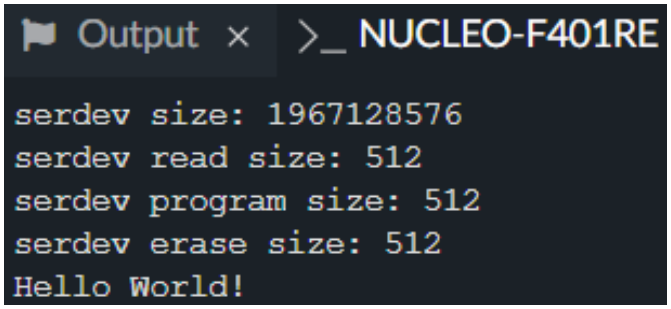
Using an SD Card (2)

```
  printf("serdev size: %llu\n", serdev.size());
  printf("serdev read size: %llu\n", serdev.get_read_size());
  printf("serdev program size: %llu\n", serdev.get_program_size());
  printf("serdev erase size: %llu\n", serdev.get_erase_size());
  // Set the frequency to 5 MHz
  if (0 != serdev.frequency(5000000)) {
    printf("Error setting frequency \n");
  }
  if (0 != serdev.erase(0,serdev.get_erase_size())){
    printf("Error Erasing block \n");
  }
  // Write data block to the device
  if (0 == serdev.program(block, 0, 512)) {
    // Read the data block from the device
    if (0 == serdev.read(retrieve, 0, 512)) { //read to new array
      // Print the contents of the block
      printf("%s", retrieve);
    }
  }
  serdev.deinit();   // de-initialise
}
```

a) the main.cpp program

```
{
  "target_overrides":{
    "*": {
          "target.components_add" : ["SD"],
          "sd.SPI_MOSI" : "D4",
          "sd.SPI_MISO" : "D5",
          "sd.SPI_CLK"  : "D3",
          "sd.SPI_CS"   : "D6"
    }
  }
}
```

b) the mbed_app.json file



Output from Program Example, using 2 MByte SD card

Creating Files with the FAT File System (1)

The previous program example examined the size of the SD card, and wrote and read blocks of data. It did not however create a file, so you couldn't easily read the card in a card reader on your PC. We now apply the FAT file system to the SD card, using a few of the API functions shown in the Table.

Functions	Usage
Constructor	
`FATFileSystem (const char* name, BlockDevice* bd)`	Creates a file system called name, on block device bd.
Member Functions	
`virtual int mount (BlockDevice *bd)`	Mount a file system to a block device; returns 0 on success, negative error code on failure.
`virtual int reformat (BlockDevice *bd, int allocation_unit)`	Reformat a file system, results in an empty and mounted file system; allocation_unit is number of bytes in a cluster; returns 0 on success, negative error code on failure.
`virtual int unmount( )`	Unmount a file system from block device; returns 0 on success, negative error code on failure.
`virtual int mkdir(const char *path, mode_t mode)`	Create a directory in the file system; returns 0 on success, negative error code on failure.

Creating Files with the FAT File System (2)

This program example replicates the function (a simple interrupt-driven data logging project) of Program Example 10.4 (a few slides back), but this time on the SD card.

The **mbed_app.json** file is essential, adjusted for the target board in use, as this identifies both the SD card as the target device and indicates pin connections.

```
/* Program Example 10.6 Using FATfile system to write create a file on the SD card
Works on LPC1768 and Nucleo F401RE, with pin connections shown
 */

#include "mbed.h"
#include <stdio.h>
#include <errno.h>

#include "BlockDevice.h"
#include "FATFileSystem.h"

//Take the system's default block device (from json file)
BlockDevice *bd = BlockDevice::get_default_instance();
FATFileSystem fs("fs");

InterruptIn button(BUTTON1);        //Interrupt input, use p9 for PC1768
DigitalOut led1(LED1);              //digital out to onboard LED1
Timer main_timer;                   //define elapsed time timer
bool button_pressed;                //flags button has been pressed

void toggle(void);                  //function prototype
using namespace std::chrono;
```

Creating Files with the FAT File System (3)

```
int main() {
  printf("--- Let's log some data ---\n");
  set_time(1666697756);  // Set RTC time (this is Tues, 25 Oct 2022 11:35:56)
  button.rise(&toggle);  // attach the toggle function to the rising edge
  button_pressed = 0;
  main_timer.start();
  // Try to mount the filesystem
    printf("Mounting the filesystem");
    fflush(stdout);
    int err = fs.mount(bd);  //non-zero return indicates mount failure
    printf("%s\n", (err ? "Fail :(" : "OK")); //Print OK, or Fail :( on value of err
    if (err) {
      //Reformat if we can't mount the filesystem
      printf("formatting... ");
      fflush(stdout);
      err = fs.reformat(bd);
      printf("%s\n", (err ? "Fail :(" : "OK"));
      if (err) {
        error("error: %s (%d)\n", strerror(-err), err);
      }
    }
```

Creating Files with the FAT File System (4)

```
 while(true){
    if(button_pressed==1){
      //open a text file named log, append mode
      FILE* Logfile = fopen ("/local/log.txt","a");
      //print elapsed time from Timer, and state of led
      fprintf(Logfile,"Elapsed time=%u ms, ",duration_cast<milliseconds>
                                          (main_timer.elapsed_time()).count());
      if(led1.read() == 1)fprintf(Logfile,"led is 1\n\r");
      else fprintf(Logfile,"led is 0\n\r");
      time_t seconds = time(NULL);   //return elapsed seconds
      fprintf(Logfile,"Time of reading = %s\n\n", ctime(&seconds));
      fclose(Logfile);                              // close file
      button_pressed = 0;
    }          //end of if
  }            //end of while
//this is ISR
void toggle() {
    wait_us(5000);        //simple debounce to settle
    if (button == 1){
      led1 = !led1;       // toggle LED
      button_pressed = 1;
    }
}
```

a) the main.cpp program

```
{
  "target_overrides":{
    "*": {
          "target.components_add" : ["SD"],
          "sd.SPI_MOSI" : "D4",
          "sd.SPI_MISO" : "D5",
          "sd.SPI_CLK"  : "D3",
          "sd.SPI_CS"   : "D6"
    }
  }
}
```

b) the mbed_app.json file

replace D4,D5,D3,D6 with p5,p6,p7,p8 for LPC1768

Chapter Review

- Microcontrollers use memory for holding the program code (program memory) and the working data (data memory) in an embedded system.
- Volatile memory loses its data once power is removed, whereas non-volatile can retain data with no power. Different technologies are used to realise these memory types, including SRAM and DRAM (volatile) and EEPROM and Flash (non-volatile).
- The modern microcontroller has large on-chip blocks of both Flash memory and SRAM. These are organised into a single memory map, along with the microcontroller control registers.
- Pointers point to memory locations to give a means of access to the data stored at the pointed location; they provide easy access to arrays, strings and other data structures.
- The **stdio** C library contains functions that allow us to create, open and close files, as well as read data from and write data to files.
- The Mbed Operating System has a wide selection of APIs used to access memory, and create files and filing systems. These include the Local File System for the Mbed LPC1768, and the widely used FAT file system.
- An external SD memory card can be interfaced with an Mbed target to provide extended memory capability.