Embedded Systems Design Course

Applying the mbed microcontroller

Memory and data management

These course notes are written by R.Toulson (Anglia Ruskin University) and T.Wilmshurst (University of Derby). (c) ARM 2012

These course notes accompany the textbook "Fast and effective embedded system design : Applying the ARM mbed"

FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN
Applying the ARM mbed
Rob Toulson and Tim Wilmshurst

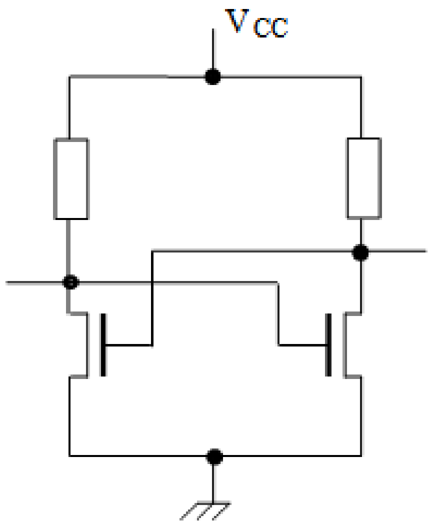
Memory and data management

- Electronic memory types
- Volatile memory
- Non-volatile memory
- Memory function types
- Using data files with the mbed
- Using formatted data
- Extended exercises

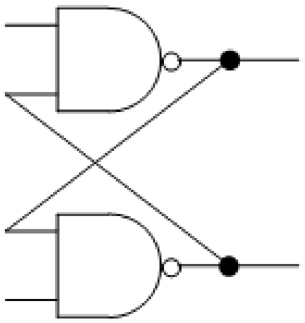
Electronic memory types

- A simple one-bit memory is a coin
  - It is “bistable”, i.e. stable in two positions, with either “heads” facing up, or “tails”
  - Consider “heads” represents logic 1, and “tails” logic 0
  - With 8 coins, an 8-bit number can be represented and stored.
- An alternative to the coin is to use an electronic bistable (“flip-flop”) circuit
  - Circuits b and c are also stable in only two states, each can be used to store one bit of data

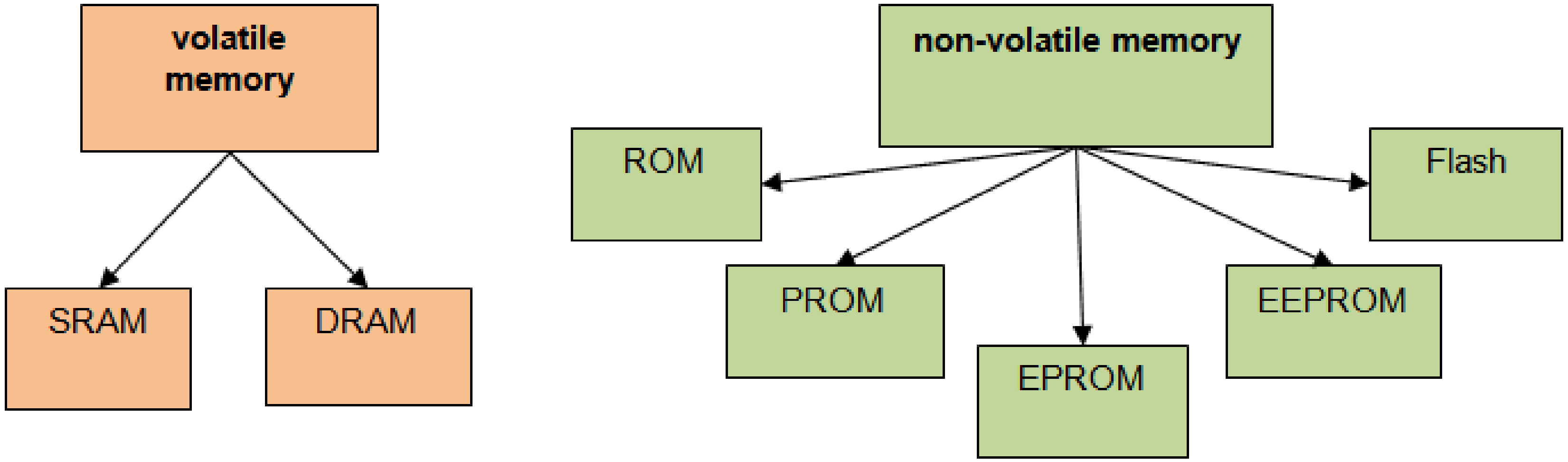
a) a coin



b) two transistors and two resistors


c) two NAND gates

Electronic memory types



- Volatile memory requires power to maintain the stored information.
- Non-volatile memory can maintain stored information, even without a power supply.

Volatile memory

- Random Access Memory (RAM) is an example of volatile memory

- SRAM (Static RAM) stores data using the state of a flip flop.
  - There are usually six transistors in a single SRAM unit
  - It is low power, easy to interface and relatively inexpensive

- DRAM (Dynamic Ram) uses one transistor and one capacitor to store one bit of data
  - DRAM can therefore take up a reduced silicon area
  - But it also requires refresh logic to recharge the capacitor every 10-100 ms
  - Power requirements are also higher in relation to SRAM

Non-volatile memory

- Read Only Memories (ROM) and PROMs (Programmable ROM) can only ever be programmed once.
- Electrically Programmable Read Only Memory (EPROM) is non-destructive using a trapped charge to represent a stored memory value.
  - The programming and erasing process is quite involved however, requiring a high voltage to program and a UV light source to erase
- With Electrically Erasable and Programmable Read Only Memory (EEPROM), bytes of data are individually writeable, readable and erasable.
- Flash memory, is a type of EEPROM but without the ability to erase individual data bytes
  - Multiple bytes have to be erased at any one time, ‘in a flash’. This leads to flash memory having a very high density and low cost.
  - Flash and EEPROM wear out however and can only handle approximately 100,000 write-erase cycles

Memory function types

- Microprocessors use memory for holding the program code (program memory) and the working data (data memory) in an embedded system

- When the processor is powered up the program data needs to be there and ready. Non-volatile memory is generally required for program memory

- There is often no need to retain data when the product is switched off, therefore volatile memory is traditionally preferred for data memory

Using data files with the mbed

- With 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.
- We can store data in files (as chars) or we can store words and strings (as character arrays).
- We can save data files to a specific area of flash memory installed on the mbed. This memory is a page-erase data flash IC which is external to the LPC1768 microprocessor.
- When interfacing the mbed data memory, we use the standard C/C++ commands as defined by the C Standard Input and Output Library (stdio.h).

Using data files with the mbed

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

A summary of useful stdio functions for implementing file access on the mbed

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.

Using data files with the mbed

- The compiler must be told where to store data files. This can be done by using the mbed 'LocalFileSystem' declaration:

```
LocalFileSystem local("local");
```

- This allows programs to read and write files on the same disk drive that is used to program the mbed Microcontroller.

- Once created, the standard C file access functions are used to open, read and write files.

Using data files with the mbed

- We can open a file with the following command:

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

- This assigns a pointer with a shorthand name (pFile) to the file at the specific location given in the fopen statement.

- We also tell the compiler whether we want read or write access to the file. This is done by the “w” syntax. If we put “r” this would make the file open as read only.

- Note also that if the file doesn’t already exist the fopen command will automatically create the file in the specified location.

- When we have finished using a file for reading or writing it is good practice to close the file, for example:

```
fclose(pFile);
```

Using data files with the mbed

Writing data to a file

- If we are looking to store numerical data we can do this in a simple way by storing individual 8-bit data values (chars). The fputc command allows this, as follows:

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

- This stores the variable write_var to the data file.

Reading data from a file

- We can read data from a file to a variable as follows:

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

- Using the stdio.h commands it is also possible to read and write words and strings and search or move through files looking for particular data elements.

Using data files with the mbed

- Exercise 1: Compile a program that creates a data file and writes the value 0x23 to that file. The file is saved on the mbed. The program then opens and reads back the data value and displays it to the screen in a terminal application.
- Open the created file datafile.txt in a word processor, you should see a hash character (#) in the top left corner (the ASCII character for 0x23 is the hash character).

```
#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
}
```

Using data files with the mbed

- Exercise 2: Compile a program that 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 terminal application.

```
#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
}
```

- Open the file textfile.txt, you should see the correct text data.

Using formatted data

- The fprintf command is used to format data when working with files.
- fprintf has very similar syntax to printf, except that the filename pointer is also required.
- An example of fprintf is to log specific events to a data file and include variable data values such as time, sensor input data and output control settings.
- Exercise 3 uses the fprintf statement in an interrupt controlled toggle switch project. Each time the toggle switch pushbutton is pressed, the LED changes state. Also on each toggle event the file log.txt is updated to include the time elapsed since the previous toggle button press and the current LED state.
- Exercise 3 also implements a simple debounce timer to avoid multiple interrupts and file write operations.

Using formatted data

- Exercise 3: Compilea program that uses fprintf statement in an interrupt controlled toggle switch project. On each toggle event and LED switches state and the file log.txt is updated to include the time elapsed since the previous toggle button press and the current LED state.

```
#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() {
  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
  }
}
```

Extended exercises

- Exercise 4: 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.
- Exercise 5: Create a program which records 5 seconds of analogue data to the screen and a text file. Use a potentiometer to generate the analogue input data.
  - You will need to specify a sample period to capture and store an analogue value every, say, 100 ms. Ensure that your data file records time and voltage data, you can then plot a chart of the measured data to visualise the input readings.
- Exercise 6: Create a new project similar to that in Exercise 5, but replace the analogue input with a sensor (e.g. a temperature sensor or accelerometer). Using a suitable sample frequency, log actual sensor data to a formatted data file.

Summary

- Electronic memory types
- Volatile memory
- Non-volatile memory
- Memory function types
- Using data files with the mbed
- Using formatted data
- Extended exercises