CUED-Device-Programming Documentation

Release 1

AJ Kabla, PO Kristensson, J Durrell, F Forni

Aug 17, 2022

Contents:

This page contains the online material for the Computing activity on Device Programming, designed at the Engineering Department of the University of Cambridge.

CHAPTER 1

Introduction

1.1 Device programming activity

1.1.1 Why such an activity?

Across all branches of Engineering, the use of sensor networks and embedded computation has become ubiquitous. Whether you look at the building industry, transportation or biomedical engineering, networks of interconnected sensors able to process and communicate data are increasingly involved in the control and diagnostic of complex operations. This technological trend has now reached most corners of not only industrial processes, but also personal life, encompassed by the fairly generic expression of Internet of Things.

Fig. 1: Source: Wilgengebroed on Flickr.

The modern Engineer must therefore embrace this ubiquity, and make sure that whatever their specialisation is, they master the basic aspects of sensing and communication. This series of practical activities is here to fill this gap and provide engineering students with the computational and experimental skills to approach with more confidence the development of simple embedded devices to perform common tasks. The tasks selected cover the basic operations of a common ARM based microcontroller. Once familiar with a microcontroller, you will soon appreciate that they are all similar, and that your new skills are amazingly transferable.

1.1.2 A few words about sponsors

This activity has been made possible by two sponsors.

- ARM and STMicroelectronics have donated the Nucleo microcontrollers, one per undergraduate student. We are very grateful to them.
- The University of Cambridge provided financial support through a Teaching And Learning Innovation Fund to run a number of student projects and promote the use of Open Source tools in undergraduate education. This grant currently covers the cost of the 3rd year (IIA) project GM2, Technology for the poorest billion, as well as part of the running costs of this second year activity.

CHAPTER 2

Tutorial

Introduction

This part of the device programming activity can be done as soon as the boards are in your hands. Watch a few videos (curated from the web), learn about the general programming approach, and get the LEDs to blink!

You will then think again about how to use functions in your programs, and learn how to use the push button integrated with the board.

The debugging section will make sure you are familiar with what happens when you make errors. It is quite difficult to understand what problems you have when you don't have a screen to report messages - but there are techniques available.

Many people find it useful to pass text to the computer connected to the board for debugging purpose. While the corresponding microcontroller code is easy, it is more difficult to configure your computer to get it to work. Give a try, but don't panic if you can't!

Requirements

1. STM32F746 STMicroelectronics development board (provided)
2. Micro-USB to USB cable (transmitting power & data)
3. Computer with web browser, internet connection and USB port available.

2.1 Getting started

2.1.1 What is a microcontroller?

Watch this short video (https://www.youtube.com/embed/jKT4H0bstH8) if you are not sure what a micro-controller is.

2.1.2 ARM & mbed microcontrollers

Unboxing and testing

- Remove the micro-controller from its packaging, and connect the micro-USB cable to USB PWR slot. Connect the other end to your computer. The microcontroller is powered on.
- The micro-controller is loaded at the first start with a default program that blinks on of the LEDs. The device has 3 LEDs accessible to the user. Press the blue button at the bottom left corner to select another the LED. It should also blink at a different frequency. This is your first interaction with your new microcontroller!

Tip: Keep the plastic packaging to store and transport the microcontroller.

Working with ARM's mbed development environment

This video (https://www.youtube.com/embed/BAzKg3vcB88) will explain to you the gist of how to program ARM based microcontrollers using their mbed development environment.

Register an account on the mbed development platform

- Open an account. Visit http://os.mbed.com/
- Click login/sign-up. Create a new account if you don't have one already.
- Once you are logged in, click on ‘‘Compiler" (top horizontal menu) to access the development environment.
- If this is the first time you use the board with the online mbed environment: at the top right of the screen, click 'no device selected', then 'add platform' and find device 'NUCLEO-F746ZG'.

You are ready to start coding!

2.1.3 A first project

Creation of a new project

- Create new project by clicking on "New",
- Make sure that the platform (i.e. the micro-controller model) is correct, and select the template "mbed OS Blinky LED Hello World",
- Open the "main.cpp" file. The code should look like this:

```
/* mbed Microcontroller Library
 * Copyright (c) 2019 ARM Limited
 * SPDX-License-Identifier: Apache-2.0
 */

#include "mbed.h"
#include "platform/mbed_thread.h"
```

(continues on next page)

(continued from previous page)

```
// Blinking rate in milliseconds
#define BLINKING_RATE_MS          500


int main()
{
    // Initialise the digital pin LED1 as an output
    DigitalOut led(LED1);

    while (true) {
        led = !led;
        thread_sleep_for(BLINKING_RATE_MS);
    }
}
```

To get us started, we will use the code below as a first example. Please delete the sample code above and replace it with the code below.

```
#include "mbed.h"

DigitalOut myled(LED1);

int main() {
        while(true) {
                myled = 1; // LED is ON
                wait(0.2); // 200 ms
                myled = 0; // LED is OFF
                wait(1.0); // 1 sec
        }
}
```

- Press the compile button. If there is no error in your code, a file is then downloaded on your computer, ready to be installed on your microcontroller.

You will notice a number of warning messages related to the use of the function "wait". As you will see in this lab, this function is ineffective as it keeps the processor busy doing nothing. The original template code did a better job. You can read more about this in the documentation if you want, but for now we will continue to use the wait function.

Dissecting the sample code

Basic knowledge of C/C++ programming is now assumed.

Do not hesitate to consult online documentation about C/C++ when appropriate. There are so many good sources available to you! See for instance:

Basic Introduction: https://www.geeksforgeeks.org/c-language-set-1-introduction/

Good set of tutorials: http://www.tutorialspoint.com/cprogramming/

Here are a few comments that may be helpful at this point:

- `main()` is the function that is executed when the microcontroller starts.
- In C/C++, a line of code is terminated with `;`, and a block is delimited by curly brackets `{ ... }`. This is different from python where line returns and indentation provide such information. Python style indentation is however good practice for the readability of your code.

- The main program contains a single "while" loop. The term between parentheses after while should be 0 or false for the loop to end, so this loops never ends.
- The variable `myled` controls the state of LED1. Although it is manipulated as an integer, it is an instance of the class DigitalOut. The pin number is specified when the object is declared, and remains attached to it. LED1 is a shortcut for the pin number associated with the user LED1. These associations are board specific, and defined in the "mbed.h" header file - so we don't need to worry about them.
- The variable `myled` is defined at the top of the code, outside of any function. It is a global variable that will be available to all functions.

Installing the code on your micro-controller

- Connect the micro-controller to your computer using a micro-USB cable. The board should be visible as a USB drive on the computer. If it isn't, you may need to install specific drivers; consult this page to get support. If you are using Windows on versions older than Win 10, try ignoring warnings such as "*Driver not installed correctly*"; it may work well enough already.
- Drag and drop the .bin file obtained at the previous step on the board
- LED at top right corner should be temporarily flashing to indicate that the transfer is happening. The program starts automatically after that.
- You should see a LED1 blinking!

Task

Explore different blinking frequencies and try the other LEDs, LED2 and LED3.

2.2 Keep going

2.2.1 Functions

Let's refresh your mind regarding the declaration and use of functions in C/C++.

- Create a new project on the mbed development site. Select the same template ("Blinky LED test for the ST Nucleo boards"), but give it a new project name. If you were to select a blank template, you would miss the mbed.h header file that contains many important elements for your code.
- Replace the demo code with the code below. What does the **select_led** function do? If you are intrigued by the expression "t%3", look for its definition; it is the remainder after division of t by 3, also called modulo.

```
#include "mbed.h"

// Green LED
DigitalOut led1(LED1);
// Blue LED
DigitalOut led2(LED2);
// Red LED
DigitalOut led3(LED3);


void select_led(int l)
{
```

(continues on next page)

(continued from previous page)

```
        if (l==1) {
                led1 = true;
                led2 = false;
                led3 = false;
        }
        else if (l==2) {
                led1 = false;
                led2 = true;
                led3 = false;
        }
        else if (l==3) {
                led1 = false;
                led2 = false;
                led3 = true;
        }
}



int main() {
         int t=0;
         while(true) {
                select_led(t);
                wait(0.5);
                t=(t%3)+1;
        }
}
```

Task

Modify the program so that select_led(0) turns all the LEDs off, and select_led(-1) turns them all on.

Change the sequence such that the pattern is {all off, led 1, led 2, led 3, all on, all off, etc.}.

For a more immersive experience, try your code while visiting this page .

Task (optional)

Program a LED sequence inspired by this video clip.

2.2.2 Physical input with a push button

The code below exploits a useful inclusion in your development board, a push button!

```
#include "mbed.h"

DigitalIn button(USER_BUTTON);
```

(continues on next page)

(continued from previous page)

```
DigitalOut led2(LED2);

int main()
{
  led2=0;
  while(true)
  {
    if (button == 1)
         led2 = true;
    else led2 = false;
    // Fyi, C programmers often like to turn such tests into logical statements:
    //   led2= !(button == 0);
    // The "!" presents the logical negation.
    wait(0.02); // 20 ms
  }
}
```

Task

Create a new project for it, compile it, install it on your board, and try it. What happens with you press the button? Is that what you expected?

`USER_BUTTON` is a constant defined to correspond to the pin number attached to the blue button.

When pressed button is true (1) and false (0) otherwise. By assigning its value to the LED, we can control the LED with the button.

The movie clip below (https://www.youtube.com/embed/XmWqP8laxxk) explains some of this using external LED and switch. Look at it if you would like more information.

Task

Edit the code so that the blue LED is on when the button is pressed, but the red LED is on when the button is not pressed (or any other LED combinations you could think about).

2.3 Debugging

Debugging is an important part of programming. Due to the lack of interfaces such as screen or sounds, one relies by default on the basic LEDs to investigate program errors. But luckily it is also possible, with a bit of extra effort, to establish text communications with the computer through the USB connection.

In this section, we will explore different types or errors, and techniques to detect them and try to address them.

2.3.1 Errors

Compile time errors

Exercise

Try to compile the code below. Read the three errors; they can be seen in the compile output section at the bottom of the window, or by hovering your mouse over relevant red line in the scroll bar. Fix the errors.

```
#include "mbed.h"

DigitalOut myled(LED1);

int main() {
        while(true) {
                led1 = 1; // LED is ON
                wait(0.2); // 200 ms
                led1 = OFF; // LED is OFF
                wait(1.0) // 1 sec
        }
}
```

Execution time errors

Exercise

Compile the code below. It should not give you any error. Move it to your controller.

```
#include "mbed.h"

// Pin D9 supports Pulse Width Modulation (PWM)
// Pin D8 does not support Pulse Width Modulation (PWM) --> run time error expected.

PwmOut led(D9);

int main() {
        led = (float)0.5;
        while(true) {    }
}
```

You would not see much, but it sends on pin D9 a square signal that you could detect on an oscilloscope. If you are curious and have a bit of spare time, feel free to read about what Pulse Width Modulation (PWM) does; you don't need to look at this now though. This is very handy to control the brightness of LEDs for instance.

As it happens, the pin D9 does support PWM, so all works fine. But pin D8 does not. **Try changing the pin D9 to D8 in the code and observe the result.**

The code should compile without error. But the board will then display its default runtime-error behaviour. This behaviour may depend on the version of the hardware and libraries you are using. You may notice the Ethernet port LED flashing irregularly, or LED 1 flashing with a pattern of 4 long and 4 short blinks. This is the signal that the controller has experienced a runtime error.

The compiler does not fully check the suitability of the pins when the code is compiled, causing the microcontroller to crash when it tries to execute the program on inappropriate pins.

These errors are more subtle to detect as the signals from the board are not clearly documented and not always consistent. If you struggle to identify the runtime error, don't get stuck and progress to the next section. Just remember that such errors exist, and note the importance of testing as you go along, for instance using LED blink patterns to monitor your progress along the executiong of the code, or using the technique introduced below.

Debugging strategies

There is a lot more information online on this topic. You will find a few more ideas there:

https://os.mbed.com/handbook/Debugging

2.3.2 Using the serial port to monitor and debug programs

This section is more advanced, but really useful once you get it to work. Communications between the microcontroller and computer will be developed further in the following tutorial.

Different operating systems will use different software (that you may need to install) in order to talk to the board. Each operating system will also have different naming conventions to identify the port used to connect to the board. Therefore, it is difficult to provide here generic instructions, and you will have to find your way through other docs and tutorials.

You can get your board to send text messages to your computer using Serial communications. What is difficult here is that it depends on the computer connected to the board. Different operating systems will use different software (that you may need to install) to communicate with the board, different names for the port used to connect the board, and they would behave slightly differently. Give it a try, but don't panic if it does not work for you straight away. You can go through the next activity without reading text from the board. >>>>>>> master

Read the first half of the mbed doc on debugging with printf() calls, until the section *Printf() from an interrupt context*.

You will need to use a Terminal to handle the communication with the board and display text. This page may be useful to install one:

https://os.mbed.com/handbook/Terminals

Give it a try, but don't panic if it does not work for you straight away. You can go through the activity 1 without reading text from the board, but serial communication is required for activities 2 and 3.

Example

Exercise

The program below should cycle the three LEDs, but doesn't work quite as expected. You can try it on your device. The third LED is not blinking, and you may assume at first that it is faulty.

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

// Green LED
DigitalOut led1(LED1);
// Blue LED
DigitalOut led2(LED2);
// Red LED
DigitalOut led3(LED3);
```

(continues on next page)

(continued from previous page)

```
void select_led(int l)
{
        if (l==1) {
                led1 = true;
                led2 = false;
                led3 = false;
        }
        else if (l==2) {
                led1 = false;
                led2 = true;
                led3 = false;
        }
        else if (l==3) {
                led1 = false;
                led2 = false;
                led3 = true;
        }
}


int main() {
    pc.baud(9600);
    int t=1;

    pc.printf("Start!\r\n", t);

    while(true) {
          select_led(t);
          pc.printf("LED %d is ON.\r\n", t);
          wait(0.5);
      // cycles the values of t
      // check how the modulo operation (%) works if unsure
          t=(t+1)%3;

    }
}
```

But the output of the program looks like this:

```
Start!
LED 1 is ON.
LED 2 is ON.
LED 0 is ON.
LED 1 is ON.
LED 2 is ON.
LED 0 is ON.
LED 1 is ON.
LED 2 is ON.
LED 0 is ON.
LED 1 is ON.
...
```

Use this information to find the problem!

Catching the output from Python

Serial communications can be used for much more than debugging. The example below shows how to catch the text in python (running on your computer) using the pySerial library. You could process it further if needed.

```
import serial
board = serial.Serial("/dev/ttyACM0", 9600)
# This creates an object able to establish a serial communication channel
# with the board. The first parameter depends on your operating system
# and probably needs to be updated.
# The second is the baud rate. It needs to match the board's settings.

while True:
    line = board.readline()
    print(line)
```

Feel free to test this script. If you are using Linux, you may need to run it as a super-user to gain access to the port, i.e. launch it from a terminal using 'sudo python script_name.py".

Of course you can also communicate the other way around. Serial communication is very handy to get devices to interact with computers, or with each other. More information is available on the arm/mbed website:

https://os.mbed.com/handbook/SerialPC#serial-communication-with-a-pc

2.4 Serial communication

Many protocols exist to allow devices to communicate with each other. In this section, we briefly present the key steps required to exchange information between the microcontroller and a computer connected to it on a USB port. This port was primarilly used, so far, to program the device, but as seen in the previous section, it can also be exploited to pass messages as strings of characters. We will first see how to pass strings from the computer to the device, and then develop a simple python interface to send and receive data, and delegate computational tasks.

2.4.1 Receiving strings from the computer

The code below shows how to capture information from the user through the Serial (USB) port. The function doing all the work is scanf. It reads from the serial port a string, and uses the format given as first parameter to extract variables in specified format. The syntax is not trivial, and rather typical of C programming, but the function is actually very powerful and flexible as a result. In all cases, a variable needs to be declared to receive the value before the function is called. It is passed to the scanf function as a reference, i.e. an address in memory where the variable has to be written. More on this as part of activity one.

```
include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

int main() {

        //      Reading a string from the computer

        char s[32]; // Array of characters to store the name

        pc.printf("\r\nPlease type your name.\r\n");
        pc.scanf("%s",s);
        pc.printf("Your name is %s.\r\n", s);
```

(continues on next page)

(continued from previous page)

```
        //      Reading integers and flaoting point values

        int n;

        pc.printf("\r\nPlease type an integer and I'll tell you what its␣
↪double is.\r\n");
        pc.scanf("%d",&n);
        pc.printf("2 x %d = %d\r\n", n, 2*n);

        float f;

        pc.printf("\r\nLet's do the same with a floating point number...\r\n
↪");
        pc.scanf("%f",&f);
        pc.printf("2 x %f = %f\r\n", f, 2*f);

}
```

Exercise

Try the code above. Depending on your serial terminal, you may or may not see the characters you type, or may have to enter them in a specific text box. Look for the documentation of your serial terminal if needed. Once you get it to work, modify it to request the name and age of the user, and return a single sentence as a string containing the name and age of the user.

2.4.2 Creating a simple python interface

Delegating work to the microcontroller

The code below uses serial input and output to delegate a very simple task to the microcontroller, adding two numbers together! This is obviouly a very ineffective way to add integers, but serves notheless as a proof of concept.

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

int main() {

        int n1, n2;

        while(1){
                pc.scanf("%d",&n1);
                pc.scanf("%d",&n2);
                pc.printf("%d\n", n1+n2);
        }
}
```

Exercise

Try the code above and make sure it works fine. What happens if you try to add fractional numbers? Try to

make it work for floating point numbers.

Python serial library

Here is a piece of python code initiating a serial channel to the microcontroller, sends two strings representing numbers to add, and read the response of the micro-controller, as a string.

```
import serial
board = serial.Serial("/dev/ttyACM0", 9600)

# communicating with strings

board.write(b'2\n')
board.write(b'5\n')
board.readline()
```

The port name `"/dev/ttyACM0"` depends on your operating system and the way it would label USB devices, as discussed in the debugging section. Replace by the port name on your system.

The characters `"\n"` are needed as they correspond to carriage return instructions that the microcontroller monitors before scaning the input.

As it is, it is not very elegant, but the coversions back and forth to strings could be packaged into a python function, only exposing to the user a relatively simple interface.

```
def mysum(n1, n2):
        board.write(str(n1) + b'\n')
        board.write(str(n2) + b'\n')
        return( int(board.readline()) )
```

Exercise

Use this python interface to communicate with your microcontroller. Add some code to the microcontroller program to toggle the state a LED each time a sum is generated, as a visual confirmation that it did the work.

2.4.3 A more complex example

Why not try a more challenging task. . .

Exercise - part a

Create code on the microcontroller to calculate the n first elements of a mathematical series, for instance the Fibonacci series. Read n from the serial port, and output the resulting elements on the serial port too.

Exercise - part b

Write a python function taking n as a parameter, and returning the n first elements of the series as an array or list. The function must delegate the calculating to the microcontroller.

http://www.st.com/en/evaluation-tools/nucleo-f746zg.html

CHAPTER 3

Activity 1: Memory and interrupts

3.1 Learning objectives

This activity will teach you ways to catch user input, respond to it, and record data based on these events. These are very generic aspects of device programming.

For practical reasons, you will learn and practice these skills using the embedded LEDs and button, which somehow constrains what can be done. This activity however sets the foundations to handle more complex sensor data.

3.2 Task to complete

The aim is to program your microcontroller to record a sequence of colours entered by the user, and then play it back. Here is the proposed approach:

- **The board starts by cycling the three LEDs, turning them on one at time, and switching every second:** `LED1` (green) for 1s –> `LED2` (blue) for 1 sec –> `LED3` (red) for 1s –> `LED1` for 1 sec, etc.
- **While the colours are cycling, the user selects a colour by pressing the button.** The colour that is ON at this time is recorded.
- **The process continues until either:**
  - **Option 1**: N colours have been entered (the size of the sequence N is set in the code), or
  - **Option 2**: the user double-clicks the button to indicate the end of the sequence.
- **Once recording is completed, the recorded sequence is played back on the LEDs.**

Option 1 is simpler, and is the recommanded task for most students. Option 2 is more challenging and may be preferred by students who have more experience, and/or those who want to extend develop their skills beyond what is required for this activity.

The video below presents a demo of the first option, so that you know what to aim for (https://www.youtube.com/embed/PDv8u4roZXs).

3.3 What you may need to learn

To complete the task, you may need to learn a few important aspects of low level programming:

- **To record a sequence, a data structure is needed.** In python, or during the Mars Lander exercise, you used clever data structures that could easily change size to accommodate more data. When programming simple devices, one tends to keep tighter control on memory, allocating buffers with a specific size. We will look at how simple arrays work in C and study examples to store and access data in them.
- **User interactions, such as pressing a button, are events that need to be monitored.** Micro-controllers have a mechanism for this called *interrupts*, whereby you can attach specific actions to specific events. You will learn how to handle these interrupts effectively.
- **Talking about memory at a low level requires us to learn about pointers.** A pointer is essentially a number that represents the location in memory of a particular data structure or function code. Pointers will be useful to tell interrupts which function to call when the button is pressed, or to keep track of where the memory buffer containing your data is located in memory.

Before getting started with the main task, you are invited to learn about the prerequisites mentioned above. Please take them in the right order. This should give you the background knowledge needed to tackle the activity. It may take you 2-3 hours to go through it.

3.3.1 Pointers

This page provides background information to better understand the techniques used in this activity. You will find a lot more information online about this topic online, possibly too much. The paragraphs below are really highlighting the minimum required to not be surprised by the some the syntax used later.

Memory and address of variables

You already know that a key difference between C/C++ and python is the fact that variables have a specific type and need to be declared.

When a variable is declared in a program:

```
int x;
```

the compiler allocates a region of the available memory to contain data of the type specified. For an integer, it would allocate 4 bytes (32 bits) to store it. Whenever you refer to x in your code, the compiler understands that you talk about the content stored in the area of memory that was allocated when you created the variable x.

When you write:

```
x=5;
```

the compiler writes in the block of memory associated with x the binary representation of the integer 5.

In general, you don't need to know where your variable is in memory; you just need to know it is called x. In some occasions it is however convenient to know the location in memory of a variable, called its *address*. In this section, we will discuss about how to manipulate variables (and functions) using their memory address.

Considering again the variable x, one can get its address using the notation &x. The type of &x is generally not the same as the type of x. An object storing an address of a particular data-type is called a *pointer*.

You can declare pointers like other variables, although the syntax is a bit unusual. The code:

```
int* p;
p = &x;
```

declares a pointer to an integer. In principle, it does not really matter whether the pointers to an int or anything else; an address in memory is just a number regardless of what it points to. A pointer to an integer could in practice point to an area of memory where you stored a float or even some code. The compiler uses however this information to help you check consistency - you could do horrible things with pointer if you don't use them properly. The following code should for instance return an error:

```
float x;
int* p;
p = &x;
```

How to manipulate data stored at a particular address?

While &x represents the address of x, *p represents the object stored at the address p. The line:

```
*p=3;
```

would store 3 in the variable x.

Why is this useful?

In C/C++, when a function is called with parameters, the value of the parameters are passed, but not the variables themselves. So the function cannot modify the values of its parameters.

Imagine you want to create a function that swaps the values of its parameters. You could write something like this:

```
void swap(int a, int b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
}
```

But this would not work, because if you call swap(x,y), the variables a and b in the swap function would relate to new integer variables that contain copies of the values of x and y.

Pointers offer a way to solve this issue. Look at the following code and try to understand what it does.

```
void swap(int* a, int* b)
{
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
```

Now instead of passing the values to the function, we provide the location of the data in memory. The function can now manipulate the content stored in memory to achieve the swap.

This function would work. To call it, we would need to pass the address of the variable instead of their values, i.e. call it with:

```
swap(&x, &y)
```

to swap the content of the variable x and y;

Task

Without writing a full program and compiling it, think about how you would create a function neg(x) that changes the sign of the variable x. Hint: the function would be called using neg(&x)...

Function pointers

Pointers can also contain the address of a section of compiled code in memory, rather than data. This allows us to pass a function as a parameter to another function, by passing the address of its code. We will use this later to tell the microcontroller what to do (i.e. what code to execute) when particular events occur.

For now, let's just look at a typical situation where this would be useful. Imagine that you want to find the second derivative of a function f. To find a good numerical estimate, you can use the central finite difference relationship that you studied in first year:

$$\frac{d^2f}{dx^2} = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

Your implementation is likely to be generic enough to be applied to any function. Passing the function as a parameter is useful to make sure that such numerical methods can be applied to any suitable function.

Study the code below and focus on the implementation of the `second_derivative` function. It uses the USB Serial communication method introduced in the tutorial section to output results, which is useful here to monitor what happens.

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

float f_1(float x)
{
        return(x-x*x);
}


float second_derivative( float (*f)(float), float x)
{
        float h = 0.001;
        float d2fdx2;
        d2fdx2 = ( f(x-h) - 2 * f(x) + f(x+h) ) / (h*h);
        return d2fdx2;
}


int main() {
        pc.baud(9600);
        float x=1.;

        pc.printf("Function Pointer test program. \r\n");

        pc.printf("Function value: f(%f)=%f \r\n", x, f_1(x));

        pc.printf("Second derivative: %f \r\n", second_derivative(f_1, x));

}
```

This example may look confusing if you are reading attentively enough. Why didn't we pass the address of the function, using `second_derivative(&f_1, 1)`? Why didn't we call the function f using `(*f)(x)` in the second_derivative function?

The reason is that a function name is treated by the compiler as a pointer. You could also have used the following syntax for the calculation of the second derivative:

```
d2fdx2 = ( (*f)(x-h) - 2 * (*f)(x) + (*f)(x+h) ) / (h*h);
```

and for the function call:

```
second_derivative(&f_1, 1)
```

but this is less readable. Feel free to try it.

Comment: Note that we used again the printf function to display the output of the calculation. The expression `%f` indicates that we want to insert there a float number. All the parameters to be inserted in the output are additional parameters to the printf function. Printf is a very powerful function to produce text output. Feel free to explore further the range of printf formatting options.

Task

Start a new project with the code above, add a function called first_derivative to calculate the first derivative of a function using the approach above, and print both the first and second derivative of $f(x) = x^2$ for $x = 2$.

$$\frac{df}{dx} = \frac{f(x+h) - f(x-h)}{2h}$$

Note that this requires you to be able to read the text output as presented in the debugging section. If you can't get the text output to work at this stage, just think about what you would do but don't worry too much about executing the code.

3.3.2 Arrays

Declaring arrays and accessing their elements

Arrays in C/C++ are structures containing a set number **N** of elements of a given type **T**. To declare an array **a** of 10 integers, one would write:

```
int a[10];
```

Elements in the array are indexed between 0 and N-1. Brackets are used to access the content of each element. Here are two examples:

```
a[0]=2;
x=a[0]+3;
```

The whole array can also be initiated at the point of declaration:

```
int a[5] = {0,1,2,3,4};
```

In memory, arrays are contiguous sections of memory that are allocated to contain exactly the amount of data requested. Because all entries in an array are of the same type (and the same size in memory), accessing the elements of an array is very fast: to find the location in memory of the nth element, you add to the address of the first element n times the

size on an element. To find the location of the next element, simply add the size of the element to the address current element.

If you try to access the content of an array beyond the range allocated, expect to get random results, or a crash at execution time.

Always double check what you are doing with arrays!

The lack of reproducibility of the errors they generate makes debugging sometimes difficult.

Passing arrays to functions

Because arrays can be large structures, arrays are not passed to functions by being copied. In fact, the variable **a** declared above is de facto a pointer. The expression *a in your code would return the first element of the array.

The following function would for instance set all the elements of an array to zero:

```
int data[5];

void set_to_zero(int* a, int n)
{
    int i=0;
    while(i<n)
    {
        a[i]=0;
        i=i+1;
    }
}

main()
{
    set_to_zero(data, 5);
}
```

Example

Study the code below, guess what it would do and try it on your board.

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

const int N = 4;
int led_cycle[N]={1,2,3,2};

void select_led(int l)
{
        if (l==1) {
                led1 = true;
                led2 = false;
                led3 = false;
        }
        else if (l==2) {
                led1 = false;
                led2 = true;
```

(continues on next page)

(continued from previous page)

```
                led3 = false;
        }
        else if (l==3) {
                led1 = false;
                led2 = false;
                led3 = true;
        }
}

int main() {
         int t=0;
         while(true) {
                select_led(led_cycle[t]);
                wait(0.5);
                t=(t+1)%N;
        }
}
```

Task

Change the code to repeat the sequence {Red, Blue, Red, Green, Blue, Green}.

3.3.3 Optional - Dynamic memory allocation

This section of the tutorial is useful for the second variant of the activity. It presents more advanced features of memory management and data manipulation using pointers.

Memory allocation during program execution

In C and C++ we can request memory from the system. This is achieved using the functions malloc, realloc and free. The function malloc (memory allocation) takes as input the size in bytes you wish to allocate in memory and returns a pointer of type void * which points to the first byte of memory:

```
void *p = malloc(10); // Allocates ten bytes; p points to the first byte
```

If the memory allocation request would fail (for instance, due to no memory being available), then the pointer p is set to NULL. It is good practice to check that the returned pointer is not NULL to verify that the memory allocation indeed succeeded.

Memory reallocation

If we already have some memory allocated, then we can grow it with the function realloc (reallocate). realloc takes as input a pointer to the first byte of an existing memory allocation and a desired new size of this memory allocation and returns a pointer of type void * which points to the first byte of the new memory allocation:

```
void *np = realloc(p, 20);
```

The above function call takes our old memory allocation pointed to by the pointer p and request its new size to be 20 bytes. The new pointer np points to the first byte of this new memory allocation. The old content is preserved in the new memory allocation (up to the new size of the new memory allocation). Note that the old pointer p now points to an invalid memory area and is useless. This pointer is often set to NULL to signal that it does no longer point to anything useful:

```
p = NULL;
```

This is known as a null-pointer. If p was not set to NULL it would be known as a dangling pointer. These are very dangerous as any attempt to dereference them or use them in any other way results in undefined behaviour (possibly a crash, such as a segmentation fault).

Freeing allocated memory

When we are done using the memory we need to return it to the system. Failing to do so means we have introduced a memory leak. To free the new pointer np we use the function free and then set the pointer to NULL:

```
free(np);
np = NULL;
```

Note that only dynamically allocated memory can be modified with realloc. The functions malloc, realloc and free are all designed to work with dynamically allocated memory. Memory allocations you did not create must not be passed to realloc or free as this results in undefined behaviour.

Checking successful memory allocation

realloc can fail to reallocate memory, for example due to no memory being available. It is good practice to therefore verify that the returned pointer is not NULL. If realloc fails then the original pointer still points to a valid memory allocation and will need to be freed manually. A common pattern is therefore as follows:

```
void *np = realloc(p, 20);
if (np == NULL) {
        free(p);
        // Insert code to handle memory allocation failure...
}
```

To use memory allocation in practice, it is useful to transform the pointer of type void * to something more useful. For example, if we want to allocate memory for storing integers then we would like the pointer to be of type int *. We would also like the memory to be allocated so that a particular number of integers fit within the allocated memory area.

Therefore, in practice, to allocate memory for 8 integers we would write:

```
int *p = (int *)malloc(sizeof(int)*8);
```

The sizeof operator returns the size of type int (4 bytes on your Mbed device), which we then multiply by 8 to get the total number of bytes we need to allocate from the system.

The `(int *)` command before malloc is known as *casting*. Just like in a theatre play, where an actor can be recast into many roles, a return value or variable can be recast into a different type. The casting operator changes the return type of `malloc` from `void *` (a pointer to any type) to `int *` (a pointer to an integer).

Having allocated the memory and cast the returned pointer to a pointer of type `int *`, we can now read and write to the memory area by dereferencing the pointer and using pointer arithmetic:

```
int *xp = p;
*xp = 16; Set first integer in the memory area to 16
xp++; // Proceed to the next integer in the allocated memory area
*xp = 32; // Set the second integer in the memory area to 32
```

We can also use array operations on our newly allocated memory area, as arrays in C and C++ are essentially the same as allocated memory areas with a fixed type (note however that pointers to arrays cannot be modified by calling malloc, realloc and free as they are not dynamically allocated by the system):

```
printf("First integer: %d"\r\n", p[0]);
printf("Second integer: %d"\r\n", p[1]);
```

Here is a complete program that demonstrates the above concepts:

```
#include "mbed.h"

int main()
{
        int *p = (int *)malloc(sizeof(int)*4);
        int *xp = p;
        *xp = 2;
        xp++;
        *xp = 4;
        xp++;
        *xp = 8;
        xp++;
        *xp = 16;

        for (int i = 0; i < 4; i++) {
                printf("[%d]: %d\r\n", i, p[i]);
        }

        int *np = (int *)realloc(p, sizeof(int)*16);
        if (np == NULL) {
                free(p);
                printf("Memory allocation failure!\r\n");
        }
        else {
                for (int i = 4; i < 16; i++) {
                        np[i] = 255;
                }
                for (int i = 0; i < 16; i++) {
                        printf("[%d]: %d\r\n", i, np[i]);
                }
                free(np);
        }
}
```

3.3.4 Interrupts

LED toggle

Let's look at one of the previous tutorial's examples:

```
#include "mbed.h"

DigitalIn button(USER_BUTTON);
DigitalOut led2(LED2);

int main()
{
  led2=0;
  while(true)
  {
```

(continues on next page)

(continued from previous page)

```
            if (button == 1)
                    led2 = true;
            else led2 = false;
    wait(0.02);
  }
}
```

What if we would like now to toggle the state of the LED each time we press the button: if the LED is off, pressing the button turns it on, but if the LED is on, pressing the button turns it off.

Instinctively one would want to write the code below:

```
#include "mbed.h"

DigitalIn button(USER_BUTTON);
DigitalOut led1(LED1);

int main()
{
  led1=0;
  while(true)
  {
        if (button == 1)
                led1 = ! led1;
                // The symbol ! is the logical NOT
        wait(0.02);
  }
}
```

But this doesn't really work. Try it out, and try to find out why.

The problem is that the state of the LED keeps alternating as long as the button is pressed. What we need to capture is not the *state* of the button, but a particular *event*: the change of state of the button when it is pressed.

How to solve this problem?

You could increase the duration of the wait, hoping that the user would release the button by the time it ends. But then you might miss the input all together.

One could monitor carefully the state of the button, and switch colour when the state changes. It would be manageable for such a simple task, but hardly scalable for complex interactions. Luckily, micro-controllers have mechanisms to help you do this, either at the hardware level, or at a low software level (kernel), such that the user doesn't need to worry about it.

Pin interrupts

Let's look at the code:

```
#include "mbed.h"

InterruptIn button(USER_BUTTON);

DigitalOut led1(LED1);

// Callback function to associate with the press button event
void onButtonPress()
{
```

(continues on next page)

(continued from previous page)

```
        led1 = !led1;
}

int main()
{   // attach the address of the callback function to the rising edge
        button.rise(onButtonPress);
        while(true)
        {   // You could do something useful here
        }
}
```

As you can see, this looks simple enough! The line:

```
InterruptIn button(USER_BUTTON);
```

creates an object of type InterruptIn that gives you a handle to monitor events on the pin `USER_BUTTON`.

The line:

```
button.rise(onButtonPress);
```

assigns a particular function with the "rise" event on the pin, which corresponds here to the button being pressed. It may appears slightly counter intuitive that the event is called rise when you are pushing the button down... but this refers to the fact that the input (voltage) on the corresponding pin is transitioning from O to 1 (Vcc).

The function `onButtonPress` is called a *callback function*. It doesn't take any parameter, and doesn't return anything either. But it changes the state of the LED when the button is pressed.

Try the code and see what happens.

You will find that this somehow works, but it is still slightly random. This is because the button is not perfect. When you press it, its state can fluctuate for a short time, a process called *bouncing* (https://www.youtube.com/embed/hAVQpKVck9s).

Mechanical switches and debouncing

Bouncing is a common problem. There are different ways to solve this issue. Some involve hardware solutions, trying to prevent rapid oscillations for instance using low pass filters. But here we are stuck with this button on the board... So the way forward is to fix it with software, another common approach.

We will see here a quick and dirty fix to confirm that the issue is indeed related to switch bouncing. In the next section, we will discuss proper solutions to this problem.

What we want is to prevent the *onButtonPress* function to be called multiple times when the button is pressed and its state fluctuates for a little while. To do this, we just need to force the program to wait a short time after each call of the callback function. This can be achieve by adding a wait function call in the callback function.

Because it is not good practice to add a wait call in an interrupt (interrupt calls should execute fast), the latest mbed compilers only allow us to use the *wait_us* function. Here, "us" stands for microseconds, which is the unit of the duration to be passed as parameter. We will see in the next tutorial how to use properly interrupts without relying on the wait functions.

Try to change the code of the callback function to:

```
void onButtonPress()
{
        led1 = !led1;
```

(continues on next page)

(continued from previous page)

```
        wait_us(300000);
}
```

You should find at this point that the toggle behaves properly. Hooray!

In the next section, we will explain why this solution is not good practice, and develop a more complex example that will show you how to properly use interrupts.

3.3.5 Interrupts (continued)

LED toggle 2.0

Let's assume that we want to toggle the state `LED1` with the push button, while the microcontroller is busy doing something very important, such as monitoring the temperature of a nuclear plant. To illustrate the progress of this other important task with the hardware present on the board, we will simply get the micro-controller to flash another LED, and assume that it is important to do it in a very regular manner.

We could start with the following code:

```
#include "mbed.h"

InterruptIn button(USER_BUTTON);

DigitalOut led1(LED1);
DigitalOut led3(LED3);

void onButtonPress()
{
        led1 = !led1;
        wait_us(300000);
}

int main()
{
        button.rise(onButtonPress);
        while(true)
        {   // This is where important calculations are made...
                led3 = !led3;
                wait(0.1);
        }
}
```

Try the code. `LED3` should blink, and the button toggle the state of `LED1`. So all seems to work. But...

Focus now on what happens to the very important task, flashing the LED. It stops for a little while when the button is pressed, while the micro-controller is running the `wait` statement in the callback function. As seen previously, we need this time delay to prevent bouncing. But this also prevents the important task to be performed properly. It would also block other interrupts that may be required to handle additional events in a more complex application. **It is a general rule to spend as little time as possible in interrupts. Comments such as wait, or even printf, may cause your micro-controller to not behave properly.**

The wait statement is only here to prevent the button to trigger multiple interrupts when pressed. We could do this differently: get the callback function to deactivate the button interrupt for a short time, and then call another function later on to turn it back on again. During this time interval, we want of course the main function to continue its important job of flashing the LED.

We could deactivate an interrupt with the following statement:

```
button.rise(NULL);
```

`NULL` is a generic C/C++ constant. When a pointer value is `NULL`, it indicates that the pointer points to nothing. The code line above therefore replaces the address of the callback function with a value that unambiguously indicates that there is no call-back function to call.

But how to reattach the interrupt to the callback function after a while, without using a `wait` statement? Time to talk about timers and time interrupts.

Time interrupts

Time interrupts allows us to trigger a callback function after a given amount of time. Two main methods are available with mbed:

- **Timeout** if a callback function needs to be called only once after a time delay,
- **Ticker** if the callback function needs to be called at regular time intervals.

To use a time interrupt, we need to declare a variable of type `Timeout`.

```
Timeout event_timeout;
```

We can now attach to event_timeout a callback function and indicate the time interval before it is called. This is done with the following statement:

```
event_timeout.attach(event_callback_function, time_interval);
```

where *event_callback_function* is the name of the function to call, and *time_interval* is expressed in seconds. Time is counted from the moment when the callback function is attached.

Now look at the following code, and try it on your board.

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led3(LED3);

InterruptIn button(USER_BUTTON);

Timeout button_debounce_timeout;
float debounce_time_interval = 0.3;


void onButtonStopDebouncing(void);

void onButtonPress(void)
{
        led1 = !led1;
        button.rise(NULL);
        button_debounce_timeout.attach(onButtonStopDebouncing, debounce_time_
↪interval);

}

void onButtonStopDebouncing(void)
{
```

(continues on next page)

(continued from previous page)

```
        button.rise(onButtonPress);
}

int main()
{
        button.rise(onButtonPress);
        while(true)
        {   // This is where important calculations are made...
                led3 = !led3;
                wait(0.1);
        }
}
```

Is the problem fixed?

Comment about function declarations

Note the line:

```
void onButtonStopDebouncing(void);
```

It seems that we declare the function twice. Why?

This is because the functions `onButtonStopDebouncing` and `onButtonPress` call each other.

If you remove the first declaration of `onButtonStopDebouncing`, the compiler will tell you that `onButtonStopDebouncing` is not defined in the function `onButtonPress`, which is correct, because it is defined further down in the code. But if you swap the order of the function, then the compiler will complain that `onButtonPress` is not declared in `onButtonStopDebouncing`.

This is why we have to introduce an early declaration of `onButtonStopDebouncing` before we write the code of the function `onButtonPress`. It tells the compiler what the function `onButtonPress` will be (types of parameters and output), which is all the information it needs to compile `onButtonPress` properly.

No time to waste!

The solution above is very satisfactory. We are not wasting time any more in the interrupts. Having done this, it now looks like the code inside the main function is not optimal either; we are still wasting time stuck in a wait statements. Maybe there is also a better way to blink a LED while allowing the processor to focus on more important tasks?

Try the code below. It uses the **Ticker** class, which calls a callback function at regular time intervals. Essentially the whole program is now managed by interrupts. We don't even need the while loop in the main function.

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led3(LED3);

Timeout button_debounce_timeout;
float debounce_time_interval = 0.3;

InterruptIn button(USER_BUTTON);

Ticker cycle_ticker;
float cycle_time_interval = 0.1;
```

(continues on next page)

(continued from previous page)

```
void onButtonStopDebouncing(void);

void onButtonPress(void)
{
        led1 = !led1;
        button.rise(NULL);
        button_debounce_timeout.attach(onButtonStopDebouncing, debounce_time_
↪interval);

}

void onButtonStopDebouncing(void)
{
        button.rise(onButtonPress);
}


void onCycleTicker(void)
{
        led3 = !led3;
}


int main()
{
        button.rise(onButtonPress);
        cycle_ticker.attach(onCycleTicker, cycle_time_interval);

        // Even more important code could be placed here

}
```

Note that the `main` function could still access the state of the button or LEDs at any time.

The section on dynamic memory allocation is most relevant to students tackling the second variant of the activity, where you don't know at the start how many colours to record in the sequence.

CHAPTER 4

Activity 2: I^{2}C bus and sensors

4.1 Learning objectives

In this second activity, you will learn about a very common aspect of device programming: monitoring, logging and transferring sensor data. It builds up on the skills you learned in the first activity - you will get to use arrays and interrupts again!

This activity will teach you how to interact with a large class of devices communicating with the ubiquitous I^{2}C bus and protocol. You will also have to polish your soldering skills, and learn how to navigate a complex data-sheet to find the information you need.

4.2 Task to complete

The data-sheet of your sensor is available `here`.

Using the sensor and your micro-controller, you will have to:

- record a temperature value every second in an array that will contain the last minute of data (older data is replaced by new data once the array is full).
- if the temperature goes above a threshold value of 28 degree Celsius, get the sensor to trigger an interrupt that will get the LEDs on the microcontroller to flash an alarm signal (for you to imagine), and transmit all the data available in the array to your computer by USB serial communication, so that the log may be analysed.
- optional: if you are keen, you may try to capture and plot the temperature data on your computer using a python script.

4.3 What you may need to learn

To complete the task, you may need to learn the following elements

- What are I^{2}C devices and how to communicate with them?

- Soldering and testing your sensor.
- Configuring the sensor using its internal registers
- Responding to hardware interrupts from the sensor

Before getting started with the main task, you are invited to learn about the prerequisites mentioned above. Please take them in the right order. This should give you the background knowledge needed to tackle the activity. It may take you 2-3 hours to go through it.

4.3.1 I²C bus and devices

What is I²C?

I²C is probably the most common protocol used to exchange information between microcontrollers and sensors, displays or actuators. I²C is for instance the protocol used under the hood during IDP to link your different boards together.

In this section, you will find a brief introduction to I²C, with probably just enough information to get you started. Not everything will be crystal clear, but hopefully most of the relevant sections of the examples code will make sense, and you will manage to modify them to get them to do what you want.

Start with the following intro to I²C, from NXP, the company who developed it a while ago, and also the manufacturer of the sensor you will use (https://www.youtube.com/embed/qeJN_80CiMU).

Of course you could go much deeper into the subject if you want. A few external links are pasted below for reference, but reading them can wait for now.

You may need to refer to them later on to better understand the code examples.

I²C bus specification: short and extended.

Connecting an I²C device to your microcontroller

You need first to identify how to connect I²C devices to the microcontroller board. I²C sensors need at least 4 connections, two for power (VCC and GND), and two for communications (SCL for the clock signal and SDA for the data).

The `user manual` of your microcontroller should allow you to identify the default pins for these communications. You can also find the description of the pins on the spec sheet online.

Task

Identify the different I2C connections available on your microcontroller. We will use the default I2C pins, called I2C1_SCL and I2C1_SDA. Can you identify them on the board? We will also need to power the sensor using the 3.3V pin to VCC, and GND. Look also for these pins on the board. They may appear multiple times. We will use the female headers to connect to the make jumper wires.

Most of the microcontrollers will come with libraries and tutorials to quickly get your system to communication through I^2C. Here is the link to the mbed documentation: mbed I2c class. Note that the protocol is simple enough that with a bit of experience it is easy to understand what signals microcontroller and attached devices are exchanging.

The data sheet of an I^2C device

For any device you wish to connect using I^2C, you will need to:

- identify or set the device address;
- configure the device if needed, by setting the value of different registers controlling the operation of the device;
- read and send data to the device.

All this information can be found in the data sheet of the device you are planning to use. Make sure you check the availability of a good documentation before selecting a device. We will work with a sensor produced by NXP, with a rather good documentation to support the work of the developer.

The data sheet of the LM75 sensor you will use is available `here`.

This is daunting document! It probably contains many technical words you are not familiar with. However, you probably don't need to understand it all to get to use the device in your application. Have a quick scan through it at first. If you know what you are looking for, you will get this information out of it quickly enough afterwards.

Most of the time, there will also be sample codes available to you online. The sensor we use here is fairly popular, and the mbed compiler even contains a fully functional template! Of course, we will use this to make sure that we have an easy start.

Task

Look at the data-sheet of the sensor. What is the address range? How to set it? Your sensor is already soldered to a breakout board. Look at the back of the breakout board, and try to understand how to set the address of the device. The next section will show you how to implement it.

4.3.2 Soldering

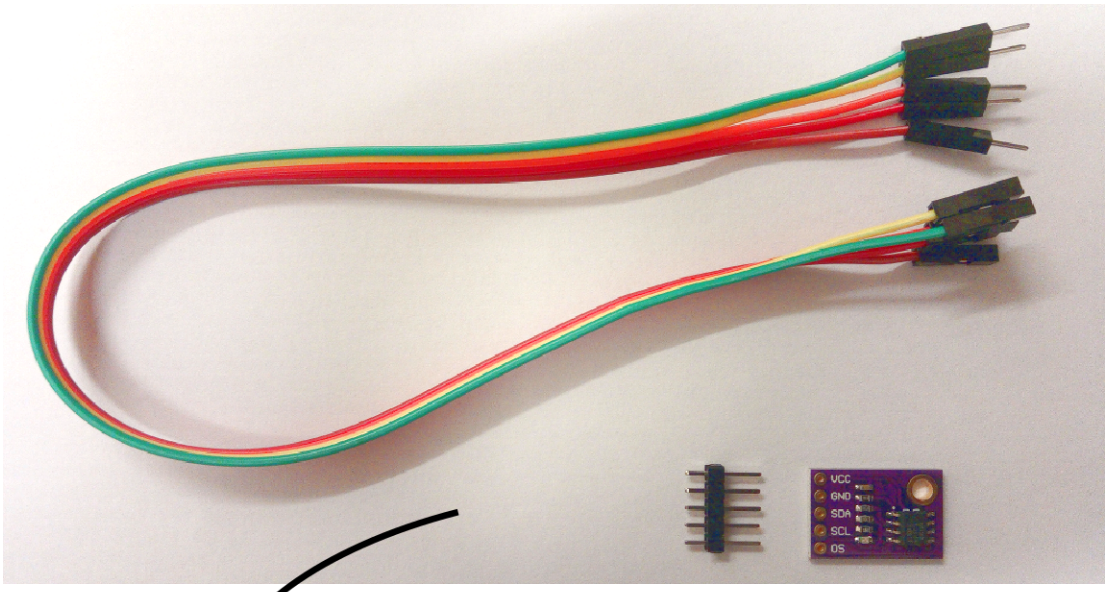
You need to solder the header pins to the breadboard, and set the address of your device by soldering each pin A0-A2 to ground (or VCC if you would like to use a different address).

Soldering is not difficult, but it is nonetheless a basic skill to learn. In this section, you will find various videos and links to get you started with soldering if you are not familiar with it already.

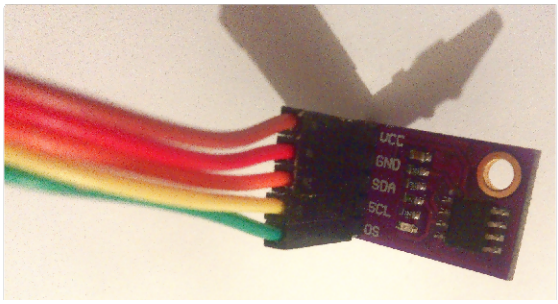
General information about soldering

Here is a good introductory video (https://www.youtube.com/embed/f95i88OSWB4), and a `comic` that will teach you what you need to get started.

This is a more advanced video (https://www.youtube.com/embed/t9LOtOBOTb0) that will show you many more soldering tips, certainly beyond what is needed here.



A bit of soldering required...



DIY in the EIETL

Task

Solder the headers and set the I2C address of your device

Where to find the soldering stations, and what equipment will you find there?

Youtube link 1: https://www.youtube.com/embed/OauRHzzIPMk

Youtube link 2: https://www.youtube.com/embed/77X4XcjcQV0

Note the safety warning:

- The soldering iron will be hot!
- Make sure to start the fume extraction.
- Wear safety goggles.
- Use lead-free solder where possible. If using solder containing lead, make sure to wash your hands afterwards.

Soldering the components

Youtube link 1: https://www.youtube.com/embed/knjjIf9SR9c

Youtube link 2: https://www.youtube.com/embed/1SItk_6e-mc

> **Danger:** Be very careful not to shortcut VCC and GND when you connect the address pads. It is easy to link the three of them together if you are not paying attention. Such a shortcut will prevent the system to work, and would probably damage your board and/or sensor. So check that the soldering is fine, and use an R-meter to make sure that VCC is not connected to GND.

Have you made a mistake? Don't panic, and read below!

Removing solder

Youtube link 1: https://www.youtube.com/embed/mcc2kdqpvKg

Youtube link 2: https://www.youtube.com/embed/pWi0EyEDnEU

The Science of soldering

Did you know that the composition of Lead-based solder is determined to match a eutectic transition? You can use your knowledge of Materials to understand how solder works!

Have a look at this nice page from DoITPoMS - Cambridge University Materials Science on the science of soldering. The Wikipedia page on solder is very informative too.

Task

Look at the lead-tin (Pb-Sn) phase diagram on your "teach yourself phase diagrams" handouts, and check on the Wikipedia page that the composition of lead-based solder used in electronics is eutectic. Find evidence that the composition of lead based solders used in plumbing tend to be set away from the eutectic point.

Link to the relevant file of Moodle (2017): teach yourself phase diagrams

4.3.3 Connecting and testing the device

Connecting the device

You were previously asked to identify the default I2C pins on your micro-controller, called I2C1_SCL and I2C1_SDA in the pin layout; they are names D14 and D15 on the board, on the top-right section of the board.

While your board is disconnected from the computer, use the male end of the jumper wires to connect VCC to the 3.3V pin of the board, and GND to GND. Then connect SDA and SCL to the pins D14 and D15, respectively, in order to use the default I2C pins.

Testing that the device is able to communicate and check its address

The code below will help you test communications with your device. It scans all I^2C addresses on the bus connected to the default I^2C pins. For each address, it tries to read something. If it manages to find a device, it returns the address on the serial port so that you can catch it on your terminal. It also uses led flashes to communicate visually the address.

Task

Paste the code below in a new project on the mbed compiler. Compile it and place it on your board. Check the that device address returned matches to address set to the device after soldering the pads.

```
#include "mbed.h"

// Create an I2C object on the defeault pins D15 and D14
I2C i2c(I2C_SDA, I2C_SCL);
// There are other pins that can be used for I2C the above is the default bus
// See the board pinout on the mbed webpage for more details.


// Use the built in serial bridge to output
Serial pc(USBTX, USBRX);


// Some flashy lights
DigitalOut green(LED1);
DigitalOut blue(LED2);
DigitalOut red(LED3);


// Create a timer so we can time the bus scan
Timer t;

// Buffer for read data
char read_data[2];

// A variable for counting
```

(continues on next page)

(continued from previous page)

```
unsigned int i=0;

// For recording the address of the last device found
unsigned int address=0;

int main()
{
    green=0;
    blue=0;


        // Make sure you set your serial terminal to match this
    pc.baud(9600);

    // Most I2C devices can cope with 400 kHz bus speed.
    // If you have any problems reduce it to 100 kHz.
    // Try changing the I2C speed and seeing what the effect on the speed of the␣
↪program is.

    i2c.frequency(400000);
    pc.printf("Starting Bus Scan....\r\n");

    // Reset and start timer
    t.reset();
    t.start();

    // Address '0' is all call and it is undefined how this would work
    // so we run from address 1 to 127

    for(i = 1; i < 128 ; i++)

        {
        // Read one byte from whatever the default read register is from every I2C␣
↪address.
        // i2c.read returns the I2C ACK bit sent by the slave, if anyone answered the␣
↪call:
        // 0 on success (ack), non-0 on failure (nack)

        // Note that while I2C addresses are from 1-127 we need to left-shift one bit
        // as the address sent on the I2C bus is 8bits
        // with the lowest but indicating if this is a write or read transaction.
        // The operation i<<1 does the bit shifting.

        if(i2c.read(i<<1, read_data, 1)==0)
            {

            // Print the address at which we found a device as a hex and as a decimal␣
↪number
            pc.printf ("I2C device found at address Hex: %x Decimal: %d\r\n",i,i);

            // If we find one device at least light the green LED and save its address
            green=1;
            address=i;
            }
        // Flash the blue LED to show we are scanning - only slow if no devices␣
↪connected
        blue=!blue;
```

(continues on next page)

(continued from previous page)

```
        }

    // Stop the timer and report time to scan
    t.stop();
    pc.printf("Bus scanned in %d ms\r\n",t.read_ms());

    // If device not found flash both red & blue LEDs
    if (address==0)
        {
            red=0;green=0;blue=1;
            while(1)
                {
                red=!red;
                blue=!blue;
                wait(0.25);
                }
            }

    // If we find at least one device
    // Flash address using LEDs:
    // Red flashes first digit and blue second

    red=0;      blue=0;

    while(1)
        {
        wait(2);
        for (i=0;i<(address/16);i++)
            {
            wait(0.25);
            red=1;
            wait(0.25);
            red=0;
            }
        wait(0.5);
        for (i=0;i<(address%16);i++)
            {
            wait(0.25);
            blue=1;
            wait(0.25);
            blue=0;
            }
        }
}
```

Getting your first temperature measurements

Task

Start a new project, and select the basic template *"mbed OS Blinky LED Hello World"*. **Replace the content of main.cpp with the code below. Compile it and try it on your board. Hold the sensor between your fingers, and monitor the evolution of the temperature.**

You will need to catch the serial output to read the temperature, as explained in the corresponding tutorial section: *Using the serial port to monitor and debug programs*

Use this code as it is, without necessarily trying to understand it at this point. The next section will describe in detail how to communicate with the sensor and extract relevant information from the data sheet, if you are not familiar with this yet.

```
#include "mbed.h"

#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address

I2C i2c(I2C_SDA, I2C_SCL);

DigitalOut myled(LED1);

Serial pc(SERIAL_TX, SERIAL_RX);

volatile char TempCelsiusDisplay[] = "+abc.d C";

int main()
{

        char data_write[2];
        char data_read[2];

        /* Configure the Temperature sensor device STLM75:
        - Thermostat mode Interrupt
        - Fault tolerance: 0
        */
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        int status = i2c.write(LM75_ADDR, data_write, 2, 0);
        if (status != 0) { // Error
                while (1) {
                        myled = !myled;
                        wait(0.2);
                }
        }

        while (1) {
                // Read temperature register
                data_write[0] = LM75_REG_TEMP;
                i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
                i2c.read(LM75_ADDR, data_read, 2, 0);

                // Calculate temperature value in Celcius
                int tempval = (int)((int)data_read[0] << 8) | data_read[1];
                tempval >>= 7;
                if (tempval <= 256) {
                        TempCelsiusDisplay[0] = '+';
                } else {
                        TempCelsiusDisplay[0] = '-';
                        tempval = 512 - tempval;
                }

                // Decimal part (0.5°C precision)
                if (tempval & 0x01) {
                        TempCelsiusDisplay[5] = 0x05 + 0x30;
```

(continues on next page)

(continued from previous page)

```
                } else {
                        TempCelsiusDisplay[5] = 0x00 + 0x30;
                }

                // Integer part
                tempval >>= 1;
                TempCelsiusDisplay[1] = (tempval / 100) + 0x30;
                TempCelsiusDisplay[2] = ((tempval % 100) / 10) + 0x30;
                TempCelsiusDisplay[3] = ((tempval % 100) % 10) + 0x30;

                // Display result
                pc.printf("temp = %s\n", TempCelsiusDisplay);
                myled = !myled;
                wait(1.0);
        }

}
```

4.3.4 I²C communication with the LM75 sensor

In this tutorial, we assume that the device is connected and returns already a meaningful temperature, as introduced in the previous section. We will in particular analyse in detail the sample code providing temperature measurements: *Getting your first temperature measurements*

We will provide more details about how the device is configured, linking the code with the relevant sections of the datasheet.

Most of the information provided here is based on the section 7 of the datasheet. Please glance through it, and make sure to have it visible while reading this page.

Device registers

Section 7.4 of the datasheet lists the four data registers present in the sensor (see table 5 in datasheet). The configuration register controls the different modes of operation of the device; you can read or write on it, although you would most likely simply write on it to set the desired behaviour of the sensor. The temp register contains the last temperature reading; it is read only, as expected. The other two registers contain the information needed to control the threshold temperatures, and will be further discussed in the next section of the tutorial.

All interactions with the device involve writing and reading the content of these registers, so this is what we will look at next.

Reading and writing on the registers

Each register has an address. There is a special register in the device called pointer register that sets which data register will be involved in the following reading or writing operation. We do not need to worry to much about the details, as communications will be handled by special functions in the mbed I2C library. But it is important to understand the sequence of typical I²C communications to be able to use properly these high level functions.

A typical sequence to write in the data registers consists in sending through I²C the device address (7 bits and R/W bit set to W), followed by the value of the pointer register, to indicate which data register we want to write on, and the data to store on this register (see figs 7 and 11 in the datasheet).

Note the the value of the data line (SDA) changes when the clock is low, and must not change when the clock (SCL) is high, as this is when it would be read. However there are two exceptions, which are particular signals to indicate

the start and end of communications. To indicate a start of I^{2}C communication, the master would take SDA from high to low while the SCL is high. To indicate the end of the communication, the master would take SDA from low to high while the SCL is high. You will spot these as START and STOP in figs 7 to 12.

Note also that data is only sent one byte at a time, followed by the acknowledgement bit.

To read, a similar precess is followed, but two steps are needed (See figs 8 and 11 in the datasheet). First, we send to the I^{2}C bus the device address (7 bits and R/W bit set to W), followed by the value of the pointer register to indicate what we would want to read. The microcontroller would then send another start signal. The next part involves sending again to the I^{2}C bus the device address, but this time with the R/W bit set to R. The master (microcontroller) would continue to control the clock, but this time the slave (sensor) would control the data line, and send, one byte at a time, the data requested.

I^{2}C library functions

We will use here the write and read functions of the mbed library to performs a complete write/read transactions. Both functions handle the start, stop and acknowledgement signals for us.

Write to a slave device

```
int     write (int address, const char *data, int length, bool repeated=false)
```

Read from a slave device

```
int     read (int address, char *data, int length, bool repeated=false)
```

Let's look at what these parameters are, and shed some light on the sample code provided by the arm-mbed environment.

The address:

The mbed documentation says: "address: 8-bit I^{2}C slave address [addr | 0]". The address should therefore be passed as 8 bits, including the 7 bits for the address, followed by the read/write bit (as the least significant bit). The value of this bit actually doesn't matter as it is overridden by the library. So how to determine this value from the datasheet?

From section 7.3 of the datasheet, we know that the seven bit address should be 1001000 (the last three bits depends on how you soldered the address pads on the chip). This corresponds to 72. To turn it into the 8 bits information needed for the mbed library, we need to add an extra bit at the end, that we can set to 0 or 1. Let's set it to 0: 10010000 = 144 in base 10, or 90 in hexadecimal. Hexadecimal numbers are typed with the prefix 0x, for instance 0x90 would be equivalent to 144. Hexadecimal numbers are commonly used to represent integer data covering 1 byte (2 hexadecimal digits) or 2 bytes (4 hexadecimal digits).

This is why in the code the address is defined as:

```
#define LM75_ADDR     (0x90) // LM75 address
```

The data buffer:

Whether we need to write or read, we need a bit of memory to handle this information. A byte array of the right size is therefore needed.

- To control the config register, we need two bytes, one to store the register pointer, and one of the register value.
- To write on any of the three temperature registers, we need three bytes, one for the register pointer, and two for the temperature value.
- To read any of the three temperature registers, we need to write one byte for the register pointer, and then read two for the temperature value.

The following lines in the sample code define the relevant buffers:

```
char data_write[2];
char data_read[2];
```

The buffers then need to be manipulated to contain the relevant information. This would set the value of the configuration buffer:

```
#define LM75_REG_CONF (0x01) // Configuration Register

data_write[0] = LM75_REG_CONF;
data_write[1] = 0x02;
```

Repeated start:

By default, the read and write commands would complete the transaction with the STOP signal (repeated=false). See for instance:

```
int status = i2c.write(LM75_ADDR, data_write, 2, 0);
```

However, to read data, we need two steps, one to indicate, with a write command which register we want to read, followed by a read. The write call should in this case be sent with `repeated=true`.

```
data_write[0] = LM75_REG_TEMP;
i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
i2c.read(LM75_ADDR, data_read, 2, 0);
```

Returned values:

0 on success (ACK), non-0 on failure (NACK). The sample code uses this returned value to signal any error in the communication:

```
if (status != 0) { // Error
    while (1) {
        myled = !myled;
        wait(0.2);
    }
}
```

Converting the raw data into a temperature

Transforming data buffers into floating point temperature, and vice-versa, is a tricky task. You may not need to create such code, and could reuse the relevant sections of the examples code provided, but it helps to understand how they work.

The way temperatures are stored on the registers is defined in section 7.4.3 and 7.4.4. Have a look at it first. This is the content of the buffer data_read at the start:

These 11 bits represents the whole temperature range, with a 0.125 degree Celsius precision, i.e. 1/8 of a degree. The binary value of each bit, including sign, is detailed in the table below:

The sign convention follows an approach called two's complement. Table 10 of the datasheet shows examples of temperature values and their equivalent representation in bits.

The gist of what follows consists in manipulating the bit array to extract the exact value of the temperature. You may want to learn a bit about bitwise operations in C++ if you never encountered this before.

The mbed example code for the LM75 sensor does something really complicated and long winded to build the temperature as a text. The appendix below explains what it does, but we are going to explain here a much simpler method.

The representation we get from the sensor, stored in data_read, is not too far from the representation of a 16-bit signed integer:

If we were to paste the 16 bits of data_read in a 16-bit integer, we would get a number that is the temperature scaled by a factor 256, since the bit corresponding to 1 celsius in the sensor data corresponds to 256 in the 16-bit int. This may be a good strategy to follow.

To use precisely defined integer types, we will use the header file stdint.h:

```
#include "stdint.h" //This allow the use of integers of a known width
```

To declare a 16-bit signed int called i16, we would type:

```
int16_t i16;
```

How to fill i16 with the relevant bits stored in data_read? This is where bitwise operations are handy!

```
i16 = data_read[0];
```

would create this :

To place the bits D10 - D3 at the right place, we need to shift them bitwise using the left-shift operator "<<":

```
int16_t i16 = data_read[0] << 8
```

To complete the number, we need to add the bits D2-D0 contained in data_read[1]. This is done using the bitwise OR operator, "|", between data_read[0] << 8 and data_read[1].

```
int16_t i16 = (data_read[0] << 8) | data_read[1];
```

To get the temperature in degree Celsius, we need to divide this number by 256, making sure the output is a floating point number. To indicate to the compiler that we want the floating point division, we write 256 with a decimal point, 256.0. The conversion code therefore becomes:

```
int16_t i16 = (data_read[0] << 8) | data_read[1];
float temp = i16 / 256.0;
```

Overall, the code with the new conversion function would be:

```
#include "mbed.h"
#include "stdint.h" //This allow the use of integers of a known width

#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address

I2C i2c(I2C_SDA, I2C_SCL);

DigitalOut myled(LED1);

Serial pc(SERIAL_TX, SERIAL_RX);

int main()
{

        char data_write[2];
        char data_read[2];

        /* Configure the Temperature sensor device STLM75:
        - Thermostat mode Interrupt
        - Fault tolerance: 0
        */
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        int status = i2c.write(LM75_ADDR, data_write, 2, 0);
        if (status != 0) { // Error
                while (1)
                {
                        myled = !myled;
                        wait(0.2);
                }
        }

        while (1)
        {
                // Read temperature register
                data_write[0] = LM75_REG_TEMP;
                i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
                i2c.read(LM75_ADDR, data_read, 2, 0);

                // Calculate temperature value in Celcius
                int16_t i16 = (data_read[0] << 8) | data_read[1];
                // Read data as twos complement integer so sign is correct
                float temp = i16 / 256.0;

                // Display result
                pc.printf("Temperature = %.3f\r\n",temp);
                myled = !myled;
```

(continues on next page)

(continued from previous page)

```
                wait(1.0);
        }

}
```

Comments regarding the sample code provided through the mbed compiler

Feel free at this stage to look again at the sample code provided with the mbed compiler: *Getting your first temperature measurements*

You will recognise similar operations to transform the buffer into a number. However, because the code uses int (32 bits by default) instead of int16_t, the sign bit is not at the right position, and the conversion has to be done carefully as a result.

Moreover, the mbed code only uses 9 bits on the data, as the shift "tempval >>= 7" destroys the values of D1 and D0, hence the 0.5 degree precision, most likely to ensure compatibility with older sensors operating with 9-bit precision.

Note that the mbed code creates the string array digit by digit rather than using the printf function. A string is an array of bytes representing text characters according to what is called the ascii table. The characters "0" to "9" corresponds to values 30 to 39 in hexadecimal representation. So "k + 0x30" represents the ascii value of the character corresponding to the digit value k, with 0<=k<=9.

We encourage you to use the method explained above (using the 16-bit integer) to record and display temperature data.

In the next (and final) section, you will be given a code to test the interrupt mode of the sensor.

4.3.5 LM75 sensor and interrupts

Hardware interrupt

Read section 7.1 of the datasheet. The OS pin of the sensor is useful to let the device monitor temperature in the background, and send a signal when the temperature exceeds a threshold, Tos. A hysteresis temperature, Thyst, is also defined to avoid noisy signal in the interrupt pin. Fig 6 of the datasheet shows how the OS pin would be controlled for a certain temperature input. There are two different modes of operation for the interrupt in the device, defined in the second bit (B1) of the Config register; in what follows, we use the interrupt mode.

The code below illustrates the use of interrupts. Start a new project, and paste this code in. The sensor needs to be connected as previously, with in addition the OS pin linked to the pin D7 of the microcontroller.

The code contains a couple of new elements:

- the address of the registers TOS and THYST are introduced, as well as some code to set the interrupt and hysteresis temperatures. The code essentially does the opposite of what we did to read the temperature. There are only 9 meaningful bits for these registers; the operation " & 0xFF80" is a bitwise AND operation on the 16-bit of data and the binary number "1111111110000000"; it essentially makes sure that we set to 0 the 7 least significant bits of i16.
- The interrupt pin is active when its value is low, so we should trigger the interrupt when OS goes from high to low. We therefore set the interrupt using the function "fall" rather than "rise" as introduced in the previous activity.

When the code is running, you should be able to raise the temperature enough with your fingers to trigger the interrupt and turn the blue led on. As the sensor cools down, as new interrupt is triggered once the temperature goes below 26 degree Celsius, turning the blue led off. Each time an interrupt is triggered, a red led should also flash on the sensor, indicating the state of the OS pin (led is on when OS is low).

```
#include "mbed.h"
#include "stdint.h" //This allow the use of integers of a known width
#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address

#define LM75_REG_TOS (0x03) // TOS Register
#define LM75_REG_THYST (0x02) // THYST Register



I2C i2c(I2C_SDA, I2C_SCL);

DigitalOut myled(LED1);
DigitalOut blue(LED2);

InterruptIn lm75_int(D7); // Make sure you have the OS line connected to D7

Serial pc(SERIAL_TX, SERIAL_RX);

int16_t i16; // This variable needs to be 16 bits wide for the TOS and THYST␣
↪conversion to work - can you see why

void blue_flip()
{
        blue=!blue;
        // The instruction below may create problems on the latest mbed compilers.
        // Avoid using printf in interrupts anyway as it takes too long to execute.
        // pc.printf("Interrupt triggered!\r\n");
}

int main()
{
        char data_write[3];
        char data_read[3];

        /* Configure the Temperature sensor device STLM75:
           - Thermostat mode Interrupt
           - Fault tolerance: 0
           - Interrupt mode means that the line will trigger when you exceed TOS and␣
↪stay triggered until a register is read - see data sheet
        */
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        int status = i2c.write(LM75_ADDR, data_write, 2, 0);
        if (status != 0)
        { // Error
                while (1)
                {
                        myled = !myled;
                        wait(0.2);
                }
        }

        float tos=28; // TOS temperature
        float thyst=26; // THYST tempertuare
```

(continues on next page)

(continued from previous page)

```
        // This section of code sets the TOS register
        data_write[0]=LM75_REG_TOS;
        i16 = (int16_t)(tos*256) & 0xFF80;
        data_write[1]=(i16 >> 8) & 0xff;
        data_write[2]=i16 & 0xff;
        i2c.write(LM75_ADDR, data_write, 3, 0);

        //This section of codes set the THYST register
        data_write[0]=LM75_REG_THYST;
        i16 = (int16_t)(thyst*256) & 0xFF80;
        data_write[1]=(i16 >> 8) & 0xff;
        data_write[2]=i16 & 0xff;
        i2c.write(LM75_ADDR, data_write, 3, 0);

        // This line attaches the interrupt.
        // The interrupt line is active low so we trigger on a falling edge
        lm75_int.fall(&blue_flip);

        while (1)
        {
                // Read temperature register
                data_write[0] = LM75_REG_TEMP;
                i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
                i2c.read(LM75_ADDR, data_read, 2, 0);

                // Calculate temperature value in Celcius
                int16_t i16 = (data_read[0] << 8) | data_read[1];
                // Read data as twos complement integer so sign is correct
                float temp = i16 / 256.0;

                // Display result
                pc.printf("Temperature = %.3f\r\n",temp);
                myled = !myled;
                wait(1.0);
        }

}
```

CHAPTER 5

Activity 3: Actuation and feedback

5.1 Learning objectives

In this third activity you will learn about actuation, which transforms the logical/mathematical capabilities of your micro-controller into actions that have effect on the real world. The activity relies on the skills that you have learned in the first and second activities - you will have to use interrupts, read sensors, and solder again!

You will implement a temperature controller. The controller will acquire data from the temperature sensor and will drive a Peltier cell to achieve the desired temperature.

5.2 Task to complete

- Adjust the brightness of a LED and of a halogen bulb.
- Use a Peltier cell to transfer a constant amount of heat.
- Temperature regulation: use temperature sensor and Peltier cell to regulate the temperature of a given test-surface to a desired temperature.

5.3 What you may need to learn

- Learn about Pulse-Width-Modulation and how to use it to adjust the brightness of a LED.
- Learn what a H-Bridge is and how to use it to adjust the brightness of a halogen bulb.
- Learn what a Peltier cell is and how to drive it to transfer heat.
- Learn how to implement a feedback control loop: read sensor, compute the right control action, drive the actuator, and back.

These activities are progressive, and each relies on the former one. Please take them in the right order.

5.3.1 Pulse-Width-Modulation and LED brightness

What is Pulse-Width-Modulation ?

Pulse-width-modulation (PWM) is probably the most common way to produce an average analog voltage in electrical circuits via fast switching. The idea is simple: if you switch the voltage of a circuit between ON (V_{ref}volts) and OFF (0 volts) sufficiently fast, you will generate an average voltage in the circuit proportional to the ratio between the length of time the circuit was ON and the length of time the circuit was OFF.

Let's consider a few examples. Take a PWM frequency of 10kHz; this means that the ON/OFF cycle repeats 10000 times a second).

- A 50% duty cycle means that the circuit is ON for half of the cycle and OFF otherwise (first row in the image below). Thus, the generated averaged voltage is about $0.5V_{ref}$.
- A 25% duty cycle means that the circuit is ON for a fourth of each cycle and and OFF otherwise (last row in the image below). The generated voltage is about 0.25 V_{ref}.

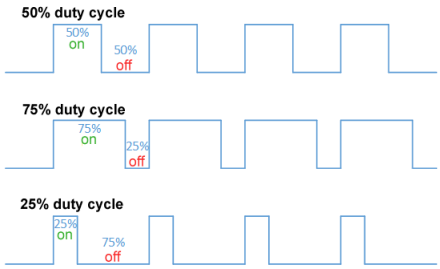


Fig. 1: PWM. Source: Wikipedia

This technique is so popular that you can find a full page on Wikipedia (https://en.wikipedia.org/wiki/Pulse-width_modulation) dedicated to it and several tutorials available on internet like Arduino and Sparkfun. You may find this video useful:

The PWM library

The MbedOS PWM library simplifies the use PWM on the microcontroller.

First, you need to declare your PWM output

```
PwmOut pwmled(LED2);
```

The instruction above sets LED2 as pwm output. Any other digital output compatible with PWM would work. Check your microcontroller manual to know which pins are compatible with PWM.

Then, you need to set the PWM period

```
pwmled.period_us(1000);
```

The instruction above sets a ON/OFF cycle every 1000 microseconds, that is, 1000 cycles each second.

Finally, the duty cycle is defined by

```
pwmled.write(0.1f);
```

The argument of the write function must be a float between 0 and 1. The instruction above sets the duty cycle at 10%.

You can also read the current PWM duty cycle through the instruction

```
pwmled.read();
```

which returns a floating-point value. For more details, please refer to the PwmOut API.

LED brightness through PWM

By now the following code should be quite readable.

```
#include "mbed.h"

PwmOut pwmled(LED2);

int main() {

        pwmled.period_us(1000);
        pwmled.write(0.1f);
        printf("pwm set to %.2f %%\n", pwmled.read());
}
```

The code switches ON and OFF the LED 10000 times a second. Within each cycle the LED is ON only for 10% of the time. Your eyes cannot see such fast frequencies and you will perceive the overal switching pattern as low brightness.

Try different duty cycles to adjust the brightness of the LED. Do you see a linear relation between duty cycle and brightness?

Tasks

- Modify the code to change brightness levels by pressing the button.
- Modify the code to make brightness slowly pulsating from low brightness to high brightness and back.

5.3.2 H-bridge and power

What is a H-bridge?

Microcontrollers are logical devices that function with very modest power. They are not very good at driving loads, like actuators, which typically require power at their inputs. The problem can be solved through an interface capable of injecting power to the actuator as a function of the logical signals of the microcontroller. This is the role of a H-bridge.

A H-bridge is a simple device. It is just a switching circuit connected to a large power generator, as shown in the figure below. The four switches S1-S4 route power to the load, represented in the figure by an encircled M (a motor, in this example).

Fig. 2: H bridge. Source: Wikipedia

The switches are typically controlled by the microcontroller.

- If S1 and S4 are closed, and S2 and S3 are open, a positive voltage is applied to the left side of the load (the motor spins in one direction).
- If S2 and S3 are closed, and S1 and S4 are open, a positive voltage is applied to the right side of the load (the motor spins in the other direction).
- If S1 and S3 are closed, and S2 and S4 are open, the same voltage is applied to both sides of the load (the motor does not want to move). The same happens if S2 and S4 are closed, and S1 and S3 are open.
- If either S1 and S2 are closed, or S3 and S4 are closed, the generator is short circuited and terrible things will happen...

So, by using a H-bridge, you can apply a voltage to your load without strong restrictions on the amount of power. Selecting the configuration of the four switches S1-S4 (ON/OFF) you can also decide the direction of the driving voltage to the load. Finally, to modulate the intensity of the voltage, just use PWM, opening and closing S1-S4 at high frequency to generate a suitable average voltage.

MAX14870 Driver

The MAX14870 Driver is a H-bridge that can be used with voltages between 4.5 V to 36 V and can supply up to about 1.7 A continuously, and 2.5 A peak. Documentation and a several additional resources can be found on the Pololu website.

The image below, from the website, shows control connections and power connections on different sides of the board. The driver is shipped with two 1x5 pin breakaway that you will have to solder.

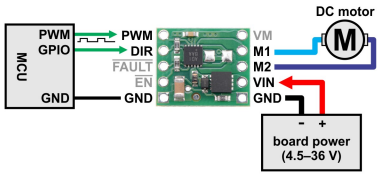


Fig. 3: H bridge connections. Source: Pololu website

- Connect the 9-volts power supply to the pins **VIN** and **GND**, paying attention to NOT reverse polarity (see figure).
- The load will be connected to the pins **M1** and **M2**.
- Connect the two **GND** pins and the **EN** pin, to a common ground.
- The MAX14870 resolves internally the configuration of the switches S1-S4 of previous section. You will need just two signals to control your load: a *direction signal* for the **DIR** pin (ON / 3.3V = M1 > M2; OFF / 0V = M1 < M2), and a *PWM signal* for the **PWM** pin (ON = voltage supplied, OFF = load is at ground).

Typically, the microcontroller sets the **DIR** pin low or high, depending on the direction of the desired voltage supplied to the load. The microcontroller also sends a PWM signal to the **PWM** pin with a given duty cycle. As a consequence, the voltage between **M1/M2** pins will be an average value between 0V and 9V proportional to the PWM duty cycle. Overall, the MAX14870 Driver essentially operates as an amplifier.

Regulation of the brightness of a halogen bulb

Connect **PWM** and **DIR** pins of MAX14870 to the microcontroller pins **PA_7/D11** and **D5**, respectively. Connect the two pins of the halogen bulb to pins **M1** and **M2**.

The following code regulates the brightness of the halogen bulb. Edit the duty cycle to change the brightness.

```
#include "mbed.h"

PwmOut pwmbulb(PA_7);
DigitalOut DIR(D5);

int main() {

        DIR = 1;
        pwmbulb.period_us(1000);
        pwmbulb.write(0.1f); //NEVER go above 0.5f.
        printf("pwm set to %.2f %%\n", pwmbulb.read());

        wait(2);
        pwmbulb.write(0.0f);
}
```

The code is very similar to the LED one. In fact, we have just added an amplifier to drive more consistent loads. The last two lines of the code turn off the driver after two seconds, to avoid termal issues (the bulb drains a lot of current, which makes both bulb and driver becoming hot very fast).

5.3.3 Peltier cells

Preparation of a cell

Peltier cells use the Peltier effect to pump heat from one plate to another of the device. The flux of heat is roughly proportional to the current passing through the peltier cell. The image below provides a schematic representation of a Peltier cell.

Fig. 4: Peltier cell. Source: Wikipedia

- One of the two plates of the Peltier cell is engineered to be hot. The other to be cold. Please attach the thermal adesive on both sides, then attach the heat sink to the cold side. To identify the hot side please refer to the `datasheet`. For the particular model provided, place the cell on the bench/desk with the black cable on the right/down and red cable on the left/down, then the top plate is the hot plate.
- Connect the Peltier cell to the MAX14870 Driver. The red cable to the **M1** pin and the black cable to the **M2** pin (but first be sure that your duty cycle is at zero!)
- Place the Peltier cell on your desk/bench with the heat sink in contact with the bench surface and the hot side exposed to the air. Then place the temperature sensor on the hot side (fix it with standard tape).

Constant heat transfer and sensor readings

The following code drives the Peltier cell with a small voltage and monitors the temperature of the cold side of the Peltier cell by cyclically reading the temperature sensor. Temperature sensor reading is made available to the user through serial.

The code to drive the Peltier cell is similar to the one for LED and bulb. Setting and reading of the temperature sensor is realized through minor adaptations of the code you have developed in Activity 2. In fact, you will need to connect the temperature sensor to the pins **D14** and **D15** as in Activity 2.

The code uses the Ticker interface to set up recurring interrupts. Recurrent interrupts allow to read sensors and send information on the serial port at precise time intervals.

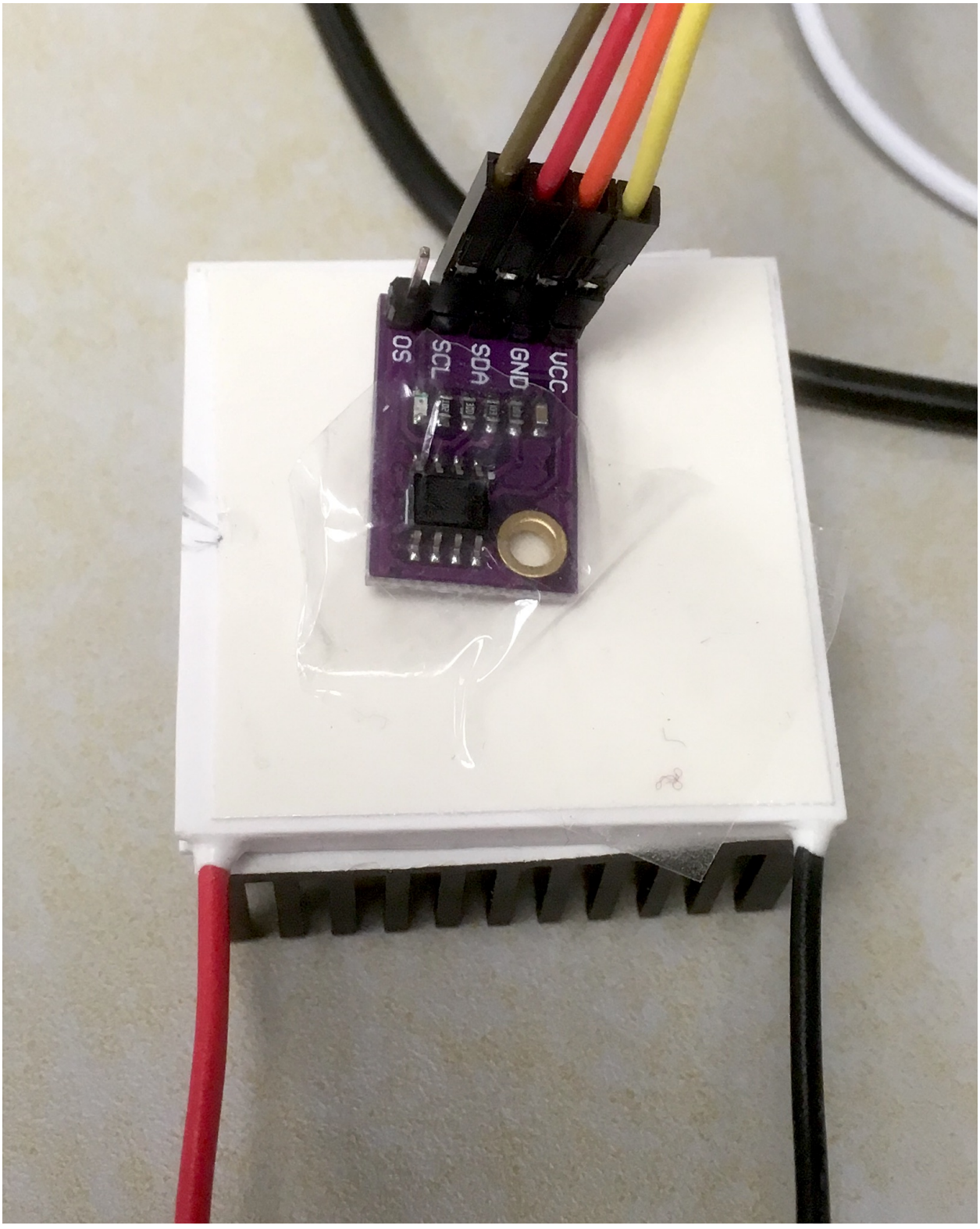


Fig. 5: Peltier bench realization.

```
#include "mbed.h"
#include "stdint.h"

// *** PWM driver: pins and variables
PwmOut peltierpwm(PA_7);
DigitalOut DIR(D5);

// *** Temperature sensor: pins and variables
#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address
I2C i2c(I2C_SDA, I2C_SCL);  //D14 and D15
Ticker dT_input;
volatile int read_input = 0;

// *** Serial communication: variables
Serial pc(SERIAL_TX, SERIAL_RX);
Ticker dT_serial;
volatile int update_serial = 0;

// *** Interrupt functions
void sensing() {
        read_input = 1;
}

void serial_com() {
        update_serial = 1;
}

// *** General functions
float read_temperature() {
        // Read temperature register
        char data_write[2];
        char data_read[2];
        data_write[0] = LM75_REG_TEMP;
        i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
        i2c.read(LM75_ADDR, data_read, 2, 0);

        // Calculate temperature value in Celcius
        int16_t i16 = (data_read[0] << 8) | data_read[1];
        // Read data as twos complement integer so sign is correct
        float temperature = i16 / 256.0;
        // Return temperature
        return temperature;
}


int main() {

        //*** temperature sensing configuration
        //Sensor configuration
        char data_write[2];
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        i2c.write(LM75_ADDR, data_write, 2, 0);
        //variables
        float temperature = 0;
```

(continues on next page)

(continued from previous page)

```
        //*** PWM drive configuration
         DIR = 1;
         peltierpwm.period_us(1000);
         peltierpwm.write(0.1f); // NEVER GO ABOVE 0.5f!
         printf("pwm set to %.2f %%\n", peltierpwm.read());

        //***  Interrupt configuration
        dT_input.attach(sensing, 0.01);
        dT_serial.attach(serial_com, 0.25);

        while(1) {
                if (read_input == 1) {
                        read_input = 0;
                        temperature = read_temperature();
                }
                if (update_serial == 1) {
                        update_serial = 0;
                        pc.printf("Pwm set to %.2f, Temperature = %.3f\r\n ",
↪peltierpwm.read() * 100, temperature);
                }
        }
}
```

The code in detail

The initial part of the code is about setting pins and defining variables.

```
// *** PWM driver: pins and variables
PwmOut peltierpwm(PA_7);
DigitalOut DIR(D5);
```

This is about settings for the PWM driver. Please check that your MAX14870 Driver is connected to the right microcontroller pins.

```
// *** Temperature sensor: pins and variables
#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address
I2C i2c(I2C_SDA, I2C_SCL);  //D14 and D15
Ticker dT_input;
volatile int read_input = 0;
```

This code is about settings for the temperature sensors (please refer to Activity 2). The ticker variable `dT_input` is used to trigger an interrupt at constant intervals of time. You will see that, as a consequence of the interrupt, the variable `read_input` will flip from 0 to 1 to inform the main routine that a sensor read must be performed. This variable is declared as `volatile` to inform the compiler that this is a sensitive variable whose state may change at any moment (therefore the compiler will not apply any optimization that could cause a delay in detecting its status).

```
// *** Serial communication: variables
Serial pc(SERIAL_TX, SERIAL_RX);
Ticker dT_serial;
volatile int update_serial = 0;
```

This code is about setting for the serial comunication. Please note that the ticker variable `dT_serial` is used

to trigger an interrupt at constant intervals of time, to request serial comunication. When the volatile variable `update_serial` is set to 1, the main routine is informed that a serial comunication must be done.

```
void sensing() {
        read_input = 1;
}
```

The function `sensing()` is called when the ticker `dT_input` triggers an interrupt. The function flips the `read_input` variable to 1, informing the main code that a sensor reading must be done as soon as possible.

```
void serial_com() {
        update_serial = 1;
}
```

The function `serial_com()` is called when the ticker `dT_serial` triggers an interrupt. The function flips the variable `update_serial` to 1, informing the main code that a serial comunicatoon must be done as soon as possible.

```
float read_temperature() {
        // Read temperature register
        char data_write[2];
        char data_read[2];
        data_write[0] = LM75_REG_TEMP;
        i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
        i2c.read(LM75_ADDR, data_read, 2, 0);

        // Calculate temperature value in Celcius
        int16_t i16 = (data_read[0] << 8) | data_read[1];
        // Read data as twos complement integer so sign is correct
        float temperature = i16 / 256.0;
        // Return temperature
        return temperature;
}
```

The function `read_temperature()` returns a temperature read from the sensor in Celcius. Please refer to Activity 2 for details.

We now go into the details of the main routine.

```
//*** temperature sensing configuration
//Sensor configuration
char data_write[2];
data_write[0] = LM75_REG_CONF;
data_write[1] = 0x02;
i2c.write(LM75_ADDR, data_write, 2, 0);
//variables
float temperature = 0;
```

This code initialize the temperature sensor and define the float variable `temperature` which will contain the sensor last read.

```
//*** PWM drive configuration
DIR = 1;
peltierpwm.period_us(1000);
peltierpwm.write(0.1f); // NEVER GO ABOVE 0.5f!
printf("pwm set to %.2f %%\n", peltierpwm.read());
```

This code set the Peltier PWM duty cycle at 10%. You are encouraged to try different duty cycles but please never go above 50% to avoid termal issues with the cell (the cell may break).

```
//*** Interrupt configuration
dT_input.attach(sensing, 0.01);
dT_serial.attach(serial_com, 0.25);
```

This code set the interval of the recurring interrupts. The first line sets a recurring interrupt every 0.01 seconds, which calls repeadetly the function `sensing()` to request a sensor reading. The second line sets a recurring interrupt every 0.25 seconds, which calls the function `serial_com()` to request serial comunication.

You will notice that serial comunication happens at much slower rate than sensor reading. The reason for these differences will be clear later, when we will design a more complex actuation mechanism. The idea is that sensing and comunication with the user can occur at different rates. Typically, sensing and actuation need a very fast rate to avoid issues but comunication with the user (serial) can be done at a slower rate to save computational resources.

Finally, the while loop constantly monitors the two variables `read_input` and `update_serial`. A sensor read is performed when `read_input` is detected equal to 1. Consequently, `read_input` is set to 0, in preparation for the next interrupt. Temperature and PWM status are comunicated to the user when `update_serial` is detected equal to 1. After that, `update_serial` is set to 0, in preparation for the next interrupt.

```
while(1) {x
          if (read_input == 1) {
                  read_input = 0;
                  temperature = read_temperature();
          }
          if (update_serial == 1) {
                  update_serial = 0;
                  pc.printf("Pwm set to %.2f, Temperature = %.3f\r\n ",
↪peltierpwm.read() * 100, temperature);
          }
}
```

Tasks

- Why does the cold side become colder as the duty cycle increase?
- Can you set the temperature of the cold side to a desired value by a suitable selection of the duty cycle?

5.3.4 Temperature controller

Feedback control

With a constant duty cycle, the Peltier cell moves heat from the cold side to the hot side. Heat is then slowly removed from the hot side by the heat-sink, and dissipated through convection. For this reason, the cold side temperature of the Peltier cell becomes colder than the room average temperature. Increasing the duty cycle further reduces the cold side temperature. However, you do not have much control on the actual temperature of the cold side. That depends on a number of factors. Even if you spend time to estimate the relationship between duty cycle and cold side temperature (at steady state), that relationship will dependent on the room average temperature, airflow within the heat sink, humidity, and many other factors.

To overcome these limitations and control precisely the cold side temperature you need feedback. Feedback is effective against uncertainties. The idea is simple: if the cold side temperature is above the desidred temperature, then the PWM duty cycle must be increased to remove more heat. If the cold side temperature is below the desired temperature, then the PWM duty cycle must be set to zero to reduce the extraction of heat. This basic feedback approach, summarized in the figure below, is extensively used in industry.

Fig. 6: Feedback loop. Source: Wikipedia

In our setting, the "system" is the Peltier cell and the "sensor" is the temperature sensor. In principle, the microcontroller can computes the difference between the cold side measured temperature and the desired/reference temperature, and proportionally adjusts the duty cycle to reach the desired temperature.

Feedback control algorithm

The code below implements a feedback control algorithm. Sensor reading and serial comunication are as in the previous code. A new Ticker variable is introduce to trigger updates of the PWM duty cycle at constant interval of times. The function `update_PWM()` adapts the PWM duty cycle accordingly to the difference between current and desired cold side temperatures. The function implement the simplest form of feedback control: proportional feedback.

```
#include "mbed.h"
#include "stdint.h"

// *** PWM driver: pins and variables
PwmOut peltierpwm(PA_7);
DigitalOut DIR(D5);
Ticker dT_output;
volatile int write_output = 0;

// *** Temperature sensor: pins and variables
#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address
I2C i2c(I2C_SDA, I2C_SCL);  //D14 and D15
Ticker dT_input;
volatile int read_input = 0;

// *** Serial communication: variables
Serial pc(SERIAL_TX, SERIAL_RX);
Ticker dT_serial;
volatile int update_serial = 0;

// *** Interrupt functions
void sensing() {
        read_input = 1;
}

void actuation() {
        write_output = 1;
}

void serial_com() {
        update_serial = 1;
}

// *** General functions
float read_temperature() {
        // Read temperature register
        char data_write[2];
        char data_read[2];
        data_write[0] = LM75_REG_TEMP;
        i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
```

(continues on next page)

(continued from previous page)

```
        i2c.read(LM75_ADDR, data_read, 2, 0);

        // Calculate temperature value in Celcius
        int16_t i16 = (data_read[0] << 8) | data_read[1];
        // Read data as twos complement integer so sign is correct
        float temperature = i16 / 256.0;
        // Return temperature
        return temperature;
}

void update_PWM(float temperature) {
        // Read temperature register
        float ref = 30;
        float kp = 0.3;
        float duty_cycle;

        duty_cycle = kp*(ref-temperature);
        if (duty_cycle <= 0) {
                peltierpwm.write(0.0f);
        }
        if (duty_cycle >= 0.50) {
                        peltierpwm.write(0.50f);
        }
        if (duty_cycle >= 0 && duty_cycle <= 0.50) {
                        peltierpwm.write(duty_cycle);
        }
}

int main() {

        //*** temperature sensing configuration
        //Sensor configuration
        char data_write[2];
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        i2c.write(LM75_ADDR, data_write, 2, 0);
        //variables
        float temperature = 0;

        //*** PWM drive configuration
         DIR = 1;
         peltierpwm.period_us(1000);
         peltierpwm.write(0.0f);
         printf("pwm set to %.2f %%\n", peltierpwm.read());

        //***  Interrupt configuration
        dT_input.attach(sensing, 0.01);
        dT_output.attach(actuation, 0.01);
        dT_serial.attach(serial_com, 0.25);

        while(1) {
                if (read_input == 1) {
                        read_input = 0;
                        temperature = read_temperature();
                }
                if (write_output == 1) {
                        write_output = 0;
```

(continues on next page)

(continued from previous page)

```
                        update_PWM(temperature);
                }
                if (update_serial == 1) {
                        update_serial = 0;
                        pc.printf("Pwm set to %.2f, Temperature = %.3f\r\n ",
↪peltierpwm.read() * 100, temperature, ref);
                }
        }
}
```

Feedback control algorithm in detail

Let's discuss only the new elements.

```
// *** PWM driver: pins and variables
PwmOut peltierpwm(PA_7);
DigitalOut DIR(D5);
Ticker dT_output;
volatile int write_output = 0;
```

In this code, the ticker variable `dT_output` is used to trigger an interrupt at constant intervals of time. You will see that, as a consequence of the interrupt, the variable `write_output` is set to 1. This will trigger an update of the duty cycle driving the Peltier cell.

```
void actuation() {
        write_output = 1;
}
```

The function `actuation()` is called when the `dT_output` ticker triggers an interrupt. The function changes the variable `write_output` to 1.

```
void update_PWM(float temperature) {
        // Read temperature register
        float ref = 30;
        float kp = 0.3;
        float duty_cycle;

        duty_cycle = kp*(ref-temperature);
        if (duty_cycle <= 0) {
                peltierpwm.write(0.0f);
        }
        if (duty_cycle >= 0.50) {
                        peltierpwm.write(0.50f);
        }
        if (duty_cycle >= 0 && duty_cycle <= 0.50) {
                        peltierpwm.write(duty_cycle);
        }
}
```

The function `update_PWM()` adapts the duty cycle driving the Peltier cell:

- `ref` is the desired temperature. This is set by the user.
- `duty_cycle = kp*(ref-temperature)` adjusts the duty cycle proportionally to the difference between the reference temperature `ref` and the actual measured temperature `temperature`. The proportional gain `kp` can be adjusted by the user.

The rest of the code normalizes the duty_cycle within safety bounds, for compatibility with the physical limits of Peltier cell and MAX14870 driver:

- if `duty_cycle <= 0`, the measured temperature is already below the reference temperature and the best action is to turn off the Peltier cell by setting `peltierpwm.write(0.0f);`
- if `duty_cycle >= 0.50` then the actual duty cycle is normalized to the maximum value `peltierpwm.write(0.50f)` to avoid large currents within the Peltier cell.

The initial part of the main code is used for initialization of the controller. For instance,

```
//*** PWM drive configuration
EN = 1;
peltierpwm.period_us(1000);
peltierpwm.write(0.0f);
printf("pwm set to %.2f %%\n", peltierpwm.read());
```

sets the initial PWM duty cycle at 0. Also

defines a recurrent interrupt every 0.01 seconds, which call the function `actuation()`.

Finally, within the main while loop, the code

```
if (write_output == 1) {
        write_output = 0;
        update_PWM(temperature);
}
```

triggers an update of the PWM duty cycle whenever the variable `write_output` is detected equal to 1. After that, `write_output` is set to 0, in preparation for the next interrupt.

Actuation and sensing are updated every 0.01 seconds, a very fast rate. Serial updates to the user are just four each second, a much slower rate (enough for monitor reading).

Task

- what is the difference between small and large proportional gain kp?
- Set the reference temperature through buttons.
- The proportional control you have implemented in the code above is non optimal for temperature control: it works but it is not precise. There is always a small error near the desired temperature. The best approach is the so called proportional + integral control. The role of integral control is to estimate the exact amount of energy that the system needs at steady state to keep the temperature at the exact desired value (no small error). If you have mastered this lecture so far and you want to explore a more challanging control algorithm, please read the linked wikipedia page and add an integral component to your proportional controller. A few additional resources can be found here:

PID Cookbook Mbed

What is a PID Controller?

What are PID Tuning Parameters?

Document license and copyright

Contact: ajk61@cam.ac.uk

Documentation repository

These documents are managed at:

https://github.com/CambridgeEngineering/PartIB-Computing-Device-Programming