Embedded Systems Development Using mbed Tools

The purpose of this class is to get the student acquainted with embedded systems using mbed.

Embedded systems are all around us every day, from the clock in your car to the controller in the microwave and the blender. mbed is a highly productive architecture and set of tools that focus on developer productivity and efficiency. The entry point is http://developer.mbed.org - the developer community. There you'll find hardware platforms, component libraries and a web-hosted IDE and compiler. This means that it works on all operating systems (Linux, Mac and Windows). This is free for developers and the software is available under the commercially friendly Apache 2.0 license.

In this workshop we will cover:

- The mbed online tools and developer community.[IA1]
- Programming in C/C++ : variables, loops and functions.
- Using the terminal for output and debug.
- Digital vs Analog - what's the difference and when to use each one.
- The mbed SDK APIs:
    - DigitalIn and DigitalOut.
    - AnalogIn and AnalogOut.
    - Wait.
    - Ticker.
    - Pulse Width Modulation (PWM).

Pre-Class

- Install driver (Windows only) http://developer.mbed.org/handbook/Windows-serial-configuration
- Update board firmware (optional) http://developer.mbed.org/platforms/

How to Use the Website

First things first: the majority of the mbed platform is online. This means you can use it on any operating system - Windows, OSX, Linux and Unix - It doesn't matter; as long as it has a web browser you're good to go. Go to https://developer.mbed.org/ and follow these steps:

1. Register for an account.
2. Select your board on the Platforms page (top left).
3. Import a program into the online compiler (see http://developer.mbed.or/teams/mbed_example/).
4. Compile code.
5. Download binary to computer (automatically on each compile).
6. Load binary onto board (drag and drop to USB).

Programming Starter: Variables, Loops and Functions

There are three basic programming concepts that are key to programming: the variable, the loop and the function. Don't worry if this section is a little overwhelming, there are lots of good sample programs on the website to get you started.

Variables

Variables are a thing that you can store a value to or read a value from. In the C language there are many different variable types, but the four we will focus on in this class are int, bool, float and char.

Integers (int)

- Whole numbers between 0 - 2^31.
- Numbers like 0, 1, 2, 3, 4, 111, 222 and 333 are all integers.

- Integers do not contain anything after the decimal point. All numbers are rounded down to the lowest whole number. For example, 4.56 would be saved as 4 and 2.99 would be saved as 2.
- Int was once defined differently depending on the hardware, which was a big problem. Then a language revision happened and it's now an explicit option. As of C99 (an old C version) you can specify the number of bits an integer takes up:

int	ambiguous size
int8_t	8 bits
int16_t	16 bits
int32_t	32 bits

Floating Point (float) [IA1]

- Numbers before and after the decimal point. Fractions are a great example of what floating point numbers are used for.
- Numbers like 0.45, 7.88 and 0.001 are floats.

Boolean (bool)

- Contains two values: true (1) or false (0).

- Booleans are useful for things that can only have two states, high or low, on or off, true or false. It happens often enough to justify having its own type.

Characters (char)

- Characters are anything you can type on your keyboard surrounded by quotation marks. For example, "A", "4" and "~" are all valid representations of characters.

- A bunch of characters put together is called a **string**. For example, "*mbed rules!*" is a string.

- There are special characters, such as the new line "\n" and return characters "\r" and even the tab character "\t", that move the cursor around the screen. There are also special characters to change the text color, insert tabs, make the bell sound and so on. Google "ASCII special characters in C" for more information.

Example: Variables

```
int xInt;               // no initialization, value is random
int yInt = 0;           // initialized to 0

float xInt;             // no initialization, value is random
float yfloat = 0;       // Initialized to 0
float zfloat = 3.14;    // Initialized to 3.14

bool xBool;             // no initialization, value is random
bool yBool = true;      // initialized to true, has int value 1
bool zBool = false;     // initialized to false, has int value 0

char xChar;             // no initialization, value is random
char yChar = "A";       // initialized to "A"
```

This code snippet is an example of declaring different variable types.

Loops

Loops are a piece of code that executes more than once. There are many kinds of loops, some go for a certain number of iterations, some go on forever, and some go until something happens (a condition is met). The most common type of loop is the **while** loop.

While Loops

The **while** loop executes a piece of code *while* a condition is **true**. The syntax for a while loop looks like this:

```
while( expression ){
 // execute some code.
}
```

The above code executes over and over again while the expression is true. In C “true” is any number greater than 0, or the boolean value **true**.

Example: While Loop

```
int x = 10;

// wait for 10 seconds
while(x){
    wait(1) // wait for 1 second
    x = x-1;
}
// continue to execute other code
```

For example this code waits for x seconds and then continues.

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, main(), and all the most trivial programs can define additional functions:

```
rtype FuntionName(vtype vname){
	// function code goes here
}
```

The *rtype* is the type of the variable the function returns. The *vtype* is the type of the variable passed to the function. To keep it simple we are only going to use functions that are passed and return variables of type **void**, which is a fancy way of saying nothing.

Every program has a special function called **main**(). This function name is special and tells the compiler to start running the application from that function. Applications therefore start with main(), and if or when the code gets to the end of main() it stops running.

Using the Terminal for Output and Debugging

It would be nice if everything just worked, unfortunately this is rarely the case. Sometimes things go wrong in the program. This is always the programmer’s fault. Well, not always - but you should operate under this assumption. Blaming tools will solve less than 0.1% of your problems. When something really bad happens you will get the siren lights telling you the microcontroller has crashed. Luckily we have the ability to print messages out to the computer telling it what's going on.

To do print statements you need two things:

1. **printf** statements in the code to send the data from the microcontroller to the computer.
2. A terminal program on the computer listening to the microcontroller and displaying the data.

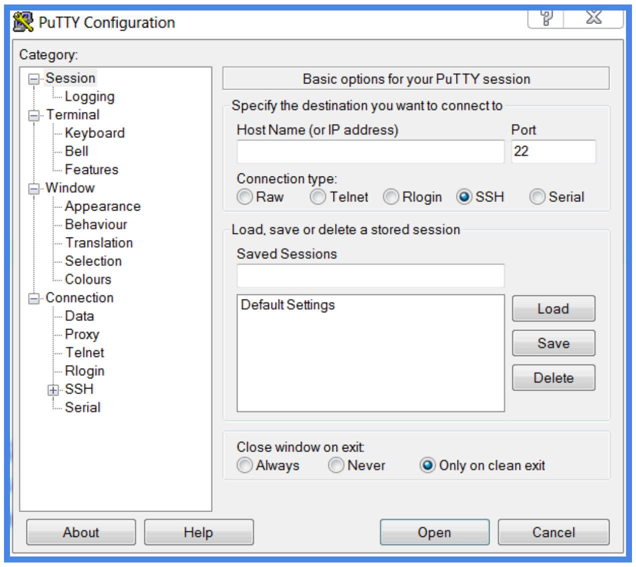
Note: If you're on Windows, make sure you install the serial driver or the microcontroller will not show up.

PuTTY

Wwhat is PuTTY? There's no intro or definition.

To set up Putty:

1. Open putty:



Specify the destination you want to connect to

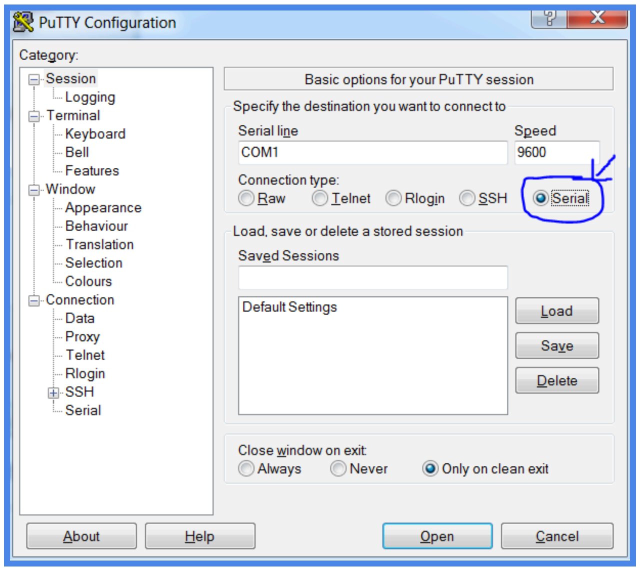
Host Name (or IP address):

Port: 22

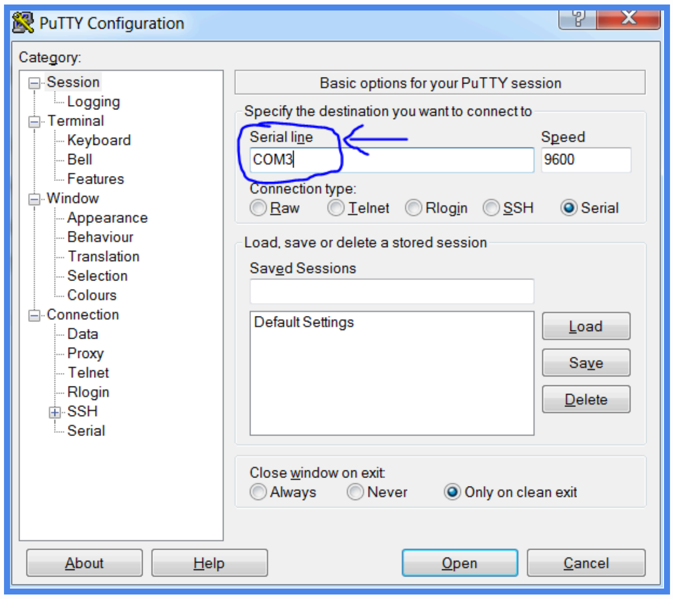
Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Close window on exit: Always Never [✓ Only on clean exit]

2. Select the **Serial** connection type:

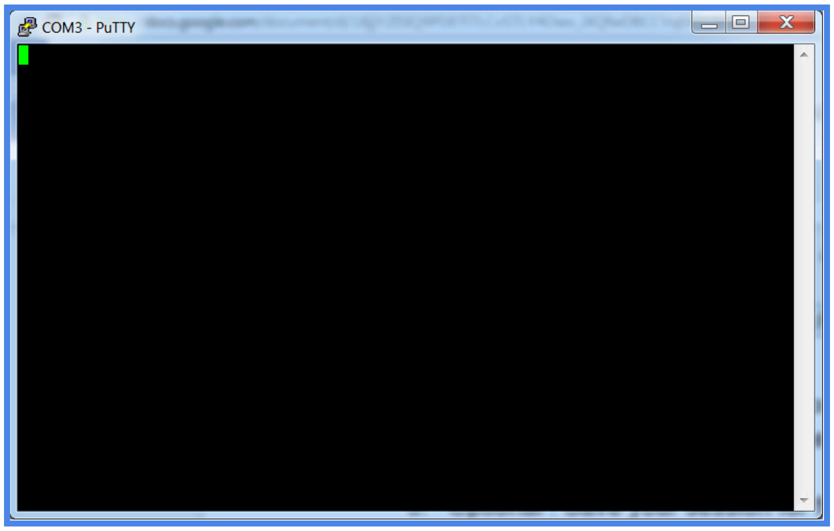


3. Modify the **serial line** parameter to match the device's connection to your computer:
   a. **Windows**: Go to Start -> Control Panel -> System -> Hardware -> Device Manager -> Ports. The name will be "mbed Serial Port (COM**x**)", where "**x**" is the number of the COM port allocated to the device.
   b. ***Mac OS X***: Use the command ls /dev/tty.usbmodem*
   c. **Linux:** Use the command ls /dev/ttyACM*



4. Check the **Speed** parameter. All mbed microcontrollers default to 9600, and this is also the default on PuTTY, so you shouldn't have to edit this parameter.

5. If you want, you can save your session for future use. This is useful if you use the connection more than once a day or use multiple microcontrollers (as each will have its own COM/tty port).
6. Click **Open** to access the terminal. It will look like this:



Great! Now we have an open terminal, but no information on it. We need to send information from the microcontroller to the terminal using **printf** statements.

Add the **Example_printf** program to your workspace, compile and load it to the board. For reference the code is provided below and at developer.mbed.org/teams/TechShop/code/Example_printf.

If this process seems too hard, here is a tool to do this automatically for you (windows only): http://developer.mbed.org/users/sam_grove/code/open_mbed/

Example

Print ‘Hello World’ with a new line:

```
#include "mbed.h"

int main() {
    printf("Hello World!\n\r");
    while(1){
    // Do nothing
    };
}
```

This snippet of code will print “Hello World” followed by a new line to the terminal.

To print the value of variables to the terminal you use a special character, the ‘%’, followed by a letter for what variable type you are trying to print.

- **int** = ‘%d’
- **float** = ‘%f’
- **char** = ‘%c’
- **string** = ‘%s’
- **bool** = ‘%d’ (0 or 1 are integers so it maps)
- **hex** = ‘%x’

For example, lets create an integer variable ‘x’ and give it the value 5, then print it.

```
#include "mbed.h"

int main() {
    int x = 5;
    printf("%d",x);
    while(1){
    // Do nothing
    };
}
```

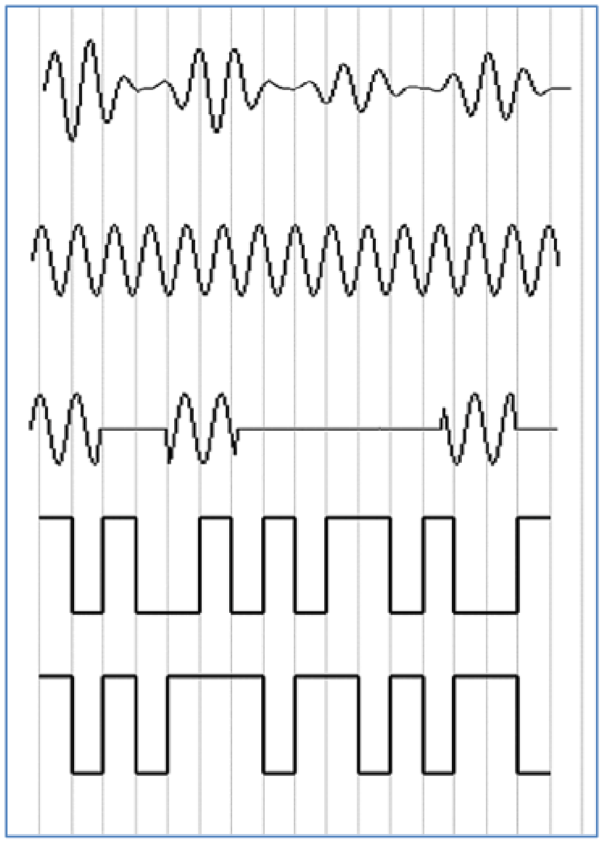
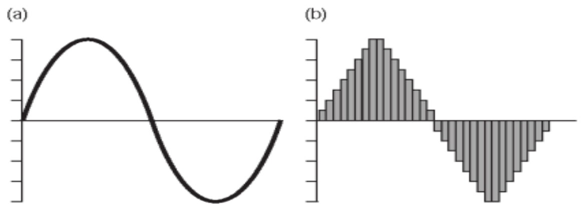
If you run the above snippet multiple times you will see that each value appears directly after the previous. To get a new line you have to print the newline characters “\n\r” after the value. Try it.

The **printf** statement can print any number of characters. The catch is **printf** is slow, so if you put many instances of it in your program it will slow it down. There is also a function called **scanf** that lets you take input from the terminal so you can create interactive programs; this is beyond the scope of this class but is something to check out on your own.

Analog vs Digital

There are two types of signals, analog and digital. Analog signals are processed by converting them to a value captured at a single point in time. The range of numbers is determined by the precision of the analog to digital converter (2^8, 2^12, etc). These are represented as a raw value (int) or normalized as a percentage of VCC / GNC (float)

Digital signals have two distinct values, 0 or 1. The voltage represented by this signal is defined by the voltage of a circuit usually referred to as VCC. Digital is really good for turning things on and off, communication and control. The problem is most things are not digital, most things we want to interact with are analog. Sound, temperature, and pressure just to name a few. All analog things exist on a sliding scale with infinite precision, but for ease of use we break them up into finite blocks so we can measure them.

	Analog is squiggly with an infinite smoothness. To use analog in a computer we approximate the smooth line into a whole bunch of skinny rectangles. This x axis of the rectangle is the amount of time it takes to record a sample.
	
	Digital Signals are square, they are either on or off, 1 or 0. I2C, SPI, UART are all digital communication protocols.

mbed SDK APIs

Let’s start doing things! What follows are some of the more useful APIs that mbed has built up. For a full list see http://developer.mbed.org/handbook.

Digital In and Out

The **DigitalIn** and **DigitalOut** types are used for declaring a pin a digital input or output. In general you write to digital outputs and read from digital inputs. A common example of a digital input is a button. A common example of a digital output is a LED. Try playing with the code snippets below.

Example: button developer.mbed.org/teams/mbed_example/code/DigitalIn_HelloWorld_Mbed

```
#include "mbed.h"

DigitalIn enable(p5);
DigitalOut led(LED1);

int main() {
    while(1) {
        if(enable) {
            led = !led;
        }
        wait(0.25);
    }
}
```

Blink LED1 when enable signal is high.

Example: flash LED (developer.mbed.org/teams/mbed_example/code/DigitalOut_HelloWorld)

```
#include "mbed.h"

DigitalOut myled(LED1);

int main() {
    while(1) {
        myled = 1;  // turn myled on
        wait(0.2);
        myled = 0;  // turn myled off
        wait(0.2);
    }
}
```

Flash LED1 on and off

Challenge: How could you simplify the above program?

Analog In and Out

The **AnalogIn** and **AnalogOut** types are used for declaring a pin an analog input or output. In the mbed SDK API analog values are read in a range from 0.0 to 1.0 . A common example of an analog input would be a slide potentiometer. A good example of an analog output would be the intensity of a buzzer or the volume of a speaker. Try playing with the code snippets below.

Example: flash LED based on input voltage
developer.mbed.org/teams/mbed_example/code/AnalogIn_HelloWorld_Mbed/

```
#include "mbed.h"

AnalogIn ain(p19);
DigitalOut led(LED1);

int main() {
    while (1){
        if(ain > 0.3) {
            led = 1;
        } else {
            led = 0;
        }
    }
}
```

Turn on LED if analog voltage is > 30% of supply voltage. Remember the mbed API uses a percentage from 0 to 1 so .3 is 30%. The voltage on this pin would be VCC * 0.3 or 3.3 * 0.3.

Example: sawtooth wave
developer.mbed.org/teams/mbed_example/code/AnalogOut_HelloWorld_mbed/

```
#include "mbed.h"

AnalogOut signal(PTE30);

int main() {
    while(1) {
        for(float i=0.0; i<1.0; i+=0.1) {
            signal = i;
            wait(0.1);
        }
    }
}
```

Create a sawtooth wave by ramping up the analog output from 0% to 100% by 10% increments, then drop to zero and repeat. This will create a sawtooth wave. Note that PTE30 is board-specific and not all boards have analog output. To see if your board supports PTE30, see the Platforms page of your board.

Wait

Sometimes you need to wait on something and do nothing else, for a determined period of time. There is a really nifty function called **wait()** that does just that. The **wait()** function is a busy wait (see below). That means it spins its wheels while waiting and does nothing else but wait. The **wait()** function takes a floating point number that represents the amount of time to wait in seconds. So 1.0 is a 1 second wait and 0.001 is a 1 millisecond wait. The **wait()** function is designed to be used for times in the milliseconds and seconds range. The maximum time you can wait is 2^31 microseconds or about 30 minutes. For longer times consider using the **timer()** or real time clock functionality. For some examples of the **wait()** function see the code snippets below.

Example: wait five seconds between each LED flash

```
#include "mbed.h"

DigitalOut led(LED1);

int main(void)
{
    while(1){
        led = 0;
        wait(5);
        led = 1;
        wait(5);
    }
}
```

This code snippet is a good example of how to use the wait function to wait a determined amount of time between actions.

Busy Wait

Busy Wait is a concept in programming that waits for something and does nothing else until the thing being waited on happens. This is done with while loops. During a busy wait nothing else is being done, the chip is spinning its wheels wasting energy and processing time.

Example: wait for button press before continuing

```
#include "mbed.h"

DigitalIn button(USER_INPUT1);
DigitalOut led(LED1);

int main(void)
{
    int input = button; // read button

    // loop forever
    while(1){
       // busy wait flash led until button press
        while(input == 0){
            led != led;
```

This code snippet will busy-wait until a button is pressed.

```
            wait(.2);
        }
        printf("Button Pressed!\n\r");

    }
}
```

Ticker

The **Ticker** class is really handy for setting up events that happen at a fixed interval. The ticker class is a ‘set it and forget it’ type thing; once set up it will run on its own and the program can go off and do something else. The ticker works by taking a floating point time variable in seconds and a function to call when the counter reaches 0. The ticker will automatically reset itself and call the function each time. See the following code snippets for a good example.

Example: flash LEDs using a ticker
http://developer.mbed.org/teams/mbed_example/code/Ticker_HelloWorld/

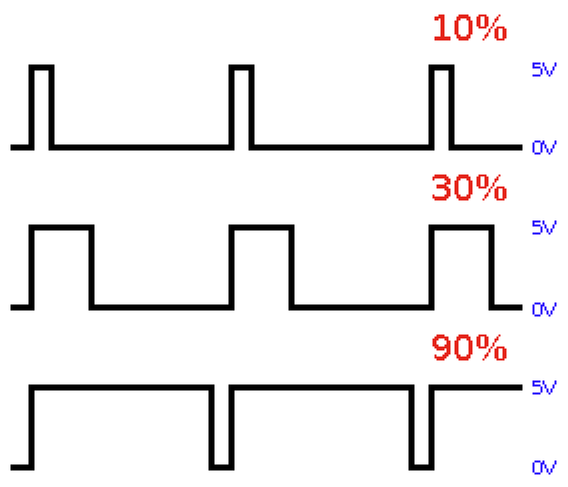
```
#include "mbed.h"

Ticker flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED2);

void flip() {
    led2 = !led2;
}

int main() {
    led2 = 1;
    // init callback and time (2s)
    flipper.attach(&flip, 2.0);

    // spin in a main loop.
    //flipper will interrupt it to call flip
    while(1) {
        led1 = !led1;
        wait(0.2);
    }
}
```

This code snippet initializes a callback (flip) to be called every two seconds. The Ticker object is called flipper. To get a ticker object to call a function you have to attach the object to the function.

Note: Ticker events are called in interrupt context. That is outside the context of this lecture, but you should remember that they should be executed as quickly as possible. Wait() and printf are discouraged.

PWM : Pulse Width Modulation

PWMs are a special kind of digital waveform. PWMs are characterized by the period of the signal and the duty cycle. The duty cycle is the percentage of the cycle that the signal spends in the high state (aka 1). The period of the PWM is the time that a single cycle of waveform takes (two consecutive matching edges). The frequency of the PWM is how many times the period happens in a second, measured in Hertz (Hz).

	This is an example of PWM waves with different duty cycles. Notice that the duty cycle is the amount of time the signal spends in the high (or on) state.

PWMs are often used to transfer average power. For example if you have a 5V pin and you send a PWM signal with a 20% duty cycle it would transfer an effective one volt across the line. If you run a 5v line with a 100% PWM you would get the full 5v transferred across the line. This technique can be used for running motors faster or slower and for dimming LEDs by providing average power over time. See the code snippets below for how to use the PWM APIs.

Note: Not all pins are 5v tolerant. Make sure the voltage levels are compatible, otherwise things *will* break, blow up, sizzle and emit the magic blue smoke, not necessarily in that order.

Example: PWM hello world

http://developer.mbed.org/teams/mbed_example/code/PwmOut_HelloWorld/

`#include "mbed.h"`
 	
`PwmOut led(LED1);`
	This code will flash a LED on for two seconds

```
int main() {
    // specify period first
    led.period(4.0f);      // 4 second period
    led.write(0.50f);      // 50% duty cycle, relative to period
    //led = 0.5f;          // shorthand for led.write()
    //led.pulsewidth(2);   // alternative to led.write, set duty
                           // cycle time in seconds
    while(1);
}
```

and off for two seconds.

The PWM can be specified by duty cycle % of period time or by absolute duty cycle time in seconds.

More Cool Stuff

All of the information included in this workshop can be found in various forms on the mbed.org website. Below are some key points to jump off from.

- Handbook page - http://mbed.org/handbook/Homepage
    - This page has a good listing of the APIs and getting started help

- Training Code example - http://mbed.org/teams/mbed_example
    - This page has all the code examples for the official API's
- For common questions see http://mbed.org/questions/
- See the youtube page for videos on using mbed

Also try using the search bar; you can find a lot of really cool projects by searching for it. Leverage the large community of code on mbed.org to minimize development time!