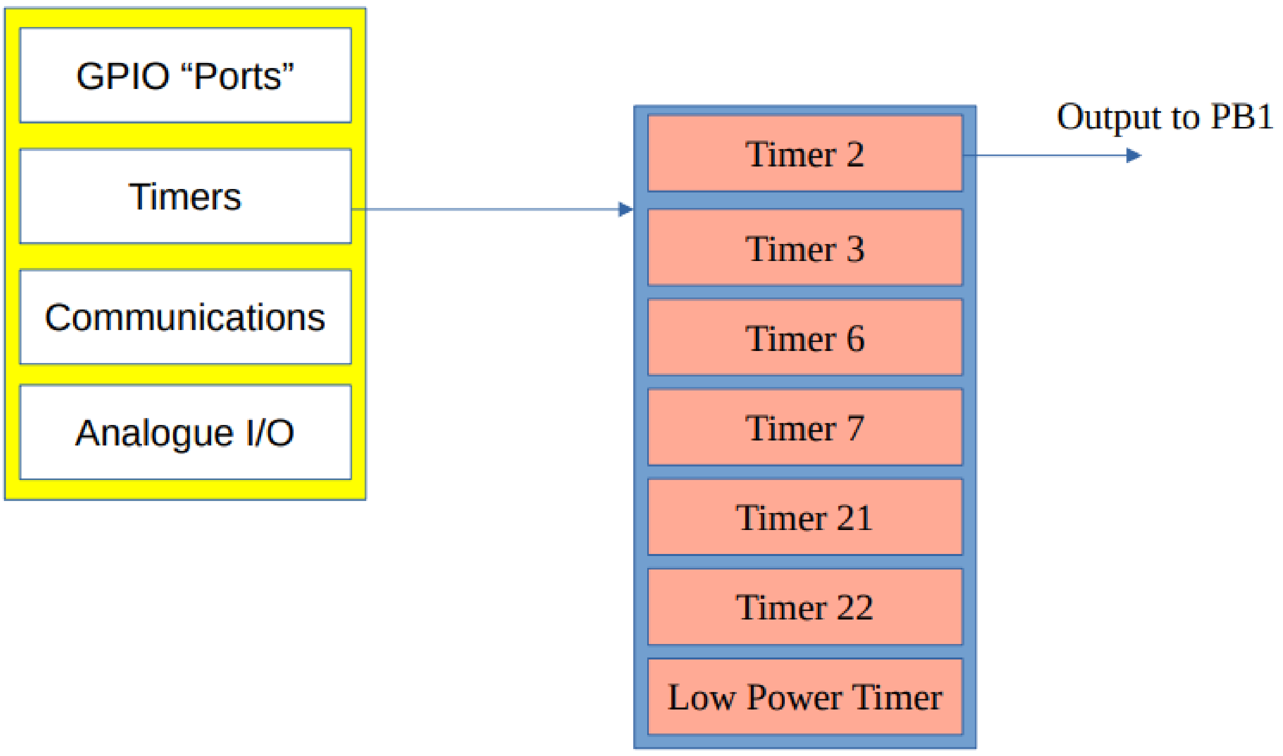
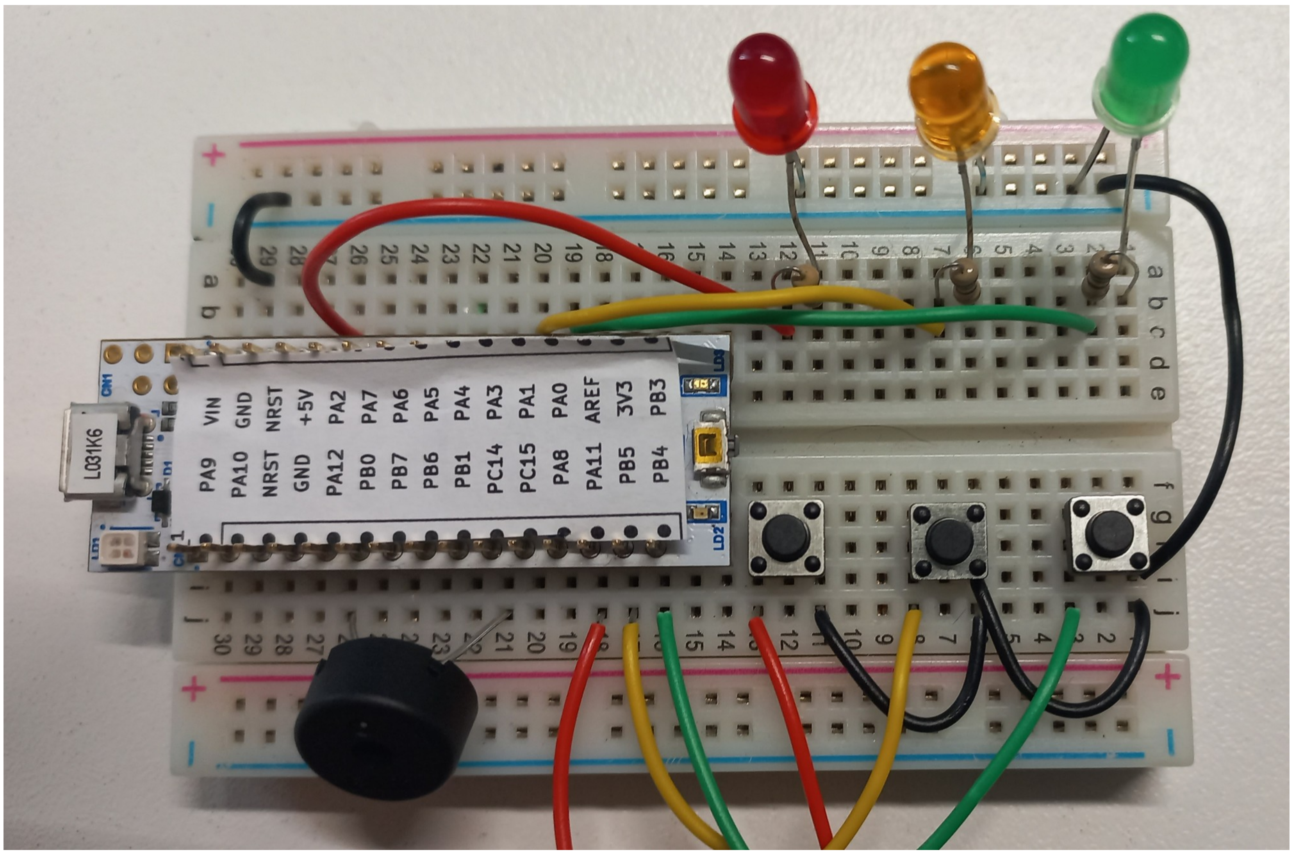
Lab 2: Timing and sound.

Introduction
This lab makes use of a hardware timer/counter within the STM32L031 to generate different sounds. The STM32L031 has a number of timer/counter devices built into its package. They are numbered in a slightly odd way. Instead of being called Timer 1,2,3 etc. they have the names shown below. The reason for this is that Timer 2 is a particular timer design which is different to Timer 6. STMicroelectronics have a number of off-the-shelf timer designs that they include in their various MCU's and each of these has a particular design number. Each of the different designs has different capabilities. We will be using Timer 2 which can send a waveform out on GPIOB bit 1 (PB1). We will attach a small Piezo electric speaker to this pin.

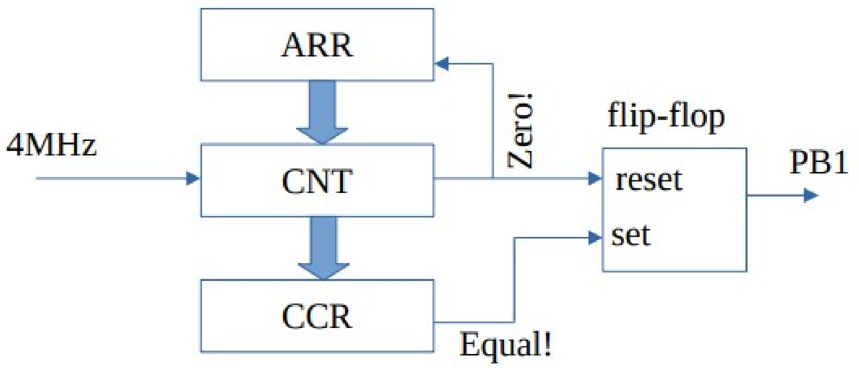


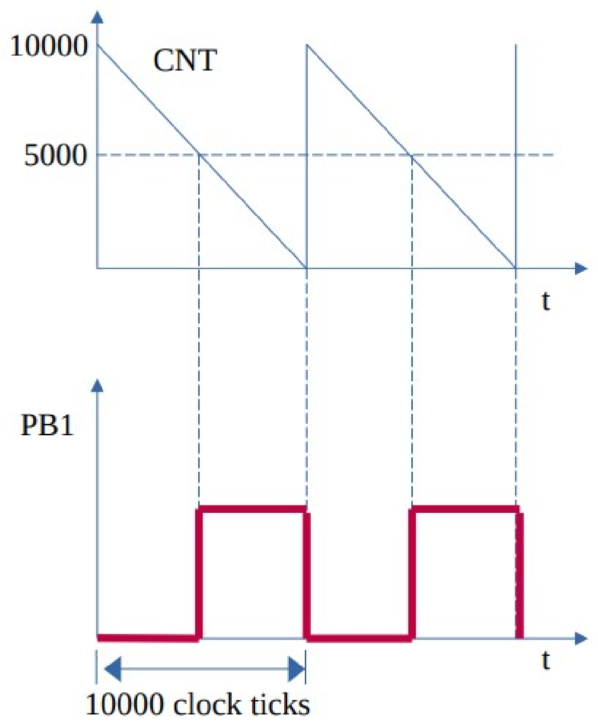
Construction
Photos of the completed board are shown below. It builds on the work in lab 1 with the same arrangement of LED's etc. A second button is added as shown which connects to PB5. The buzzer is connected between PB1 and Gnd.



Operation

A function called **initSound** is provided which connects PB1 to Timer 2, configures Timer 2 and also sets the entire MCU running at a calibrated 16MHz. This is necessary if we are to output predictable sound frequencies. The function **playNote** provides a mechanism for generating sound. It takes as single parameter: the desired output sound frequency. If you pass it a value of 0 it will turn off sound output. How is sound generated? Timer 2 is configured to count down down from a value at a predictable rate. In this case the counter counts at a rate of 4 million ticks per second or 4MHz.



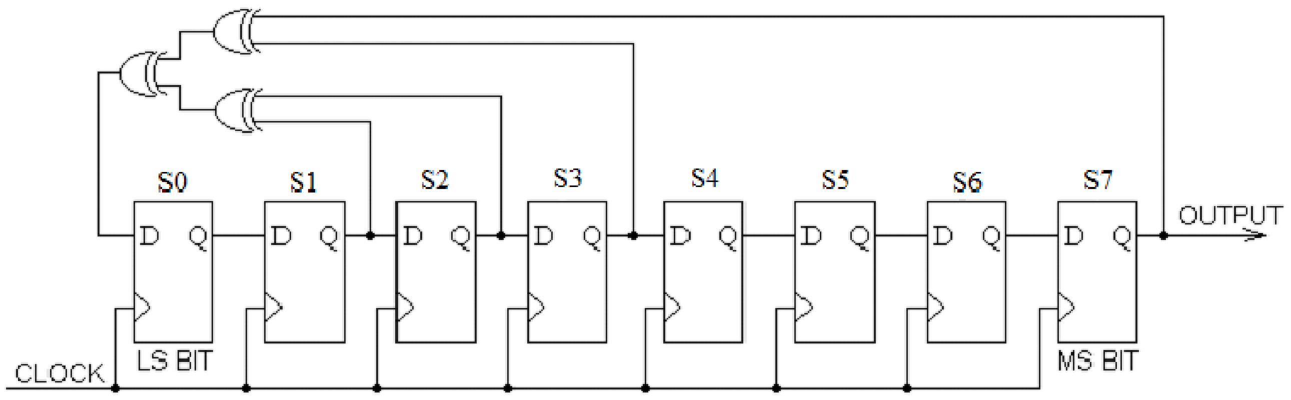


The output signal has a period of 10000 clock ticks. Each clock tick lasts 1/4000000 seconds so the output period is 10000/4000000=0.0025seconds. If we invert this we get the equivalent frequency of 400Hz. To double the output frequency to 800Hz we would have to halve the values in ARR and CCR. The **playNote** function calculates the correct value for the ARR and CCR registers based on the frequency of the output sound we ask it to produce.

The copycat game.
The idea behind this game is that the microcontroller continuously selects a random (see Appendix A) LED and turns it on. The player is then required to press the corresponding button. The starter code on Brightspace is incomplete. It will randomly turn on an LED each pass through a main loop but does not contain code to initialize or test the buttons the player interacts with.

Tasks:
1) Complete the wiring as shown above.
2) Test all LED's are working by running the starter code. You should see each LED light randomly
3) Write code to initialize the three buttons (don't forget the pull-up resistors!)
4) Write helper functions e.g. redPressed(), yellowPressed() etc which will return 1 if that button is pressed (0 otherwise)
5) Implement the game logic as follows:
    if the user presses the correct button, a 1000Hz short tone is generated (**playNote** function)
    if the user presses the wrong button, a 400Hz short tone is generated.
    Wait for the player to let go of the button before performing next pass through the main loop.
    Prevent cheating: Check to see if the player has kept all buttons pressed!
6) To add a layer of difficulty, maybe shorten the loop delay over time requiring the player to work faster.

Appendix A
Making random numbers in a deterministic computer system requires a little bit of trickery. What we typically do in fact is generate a pseudo random number which hopefully looks random enough that the player won't notice. This is often done using a shift register with feedback such as shown here:



The value in the shift register will change as 1's and 0's are shifted in from the left. The system is periodic but with sufficient bits the period will be so long that the observer will not notice.

Unless otherwise arranged, the starting value for the shift register will always be the same leading to the same sequence of values each time the system is used. This is not great for gaming. What we need is a way to initialize the system with a truly random value. The code in this lab uses electrical noise measured using the analogue to digital converter to seed the shift register with a random value when the program starts. This is done in the function **randomize**. The **random** function can then be used to return pseudo random values during gameplay. Code is shown on the next page.

```
uint32_t random(uint32_t lower,uint32_t upper)
{
    return (prbs()%(upper-lower))+lower;
}
uint32_t shift_register=0;
void randomize(void)
{
    // uses ADC noise values to seed the shift_register
    while(shift_register==0)
    {
        for (int i=0;i<10;i++)
        {
            shift_register+=(readADC()<<i);
        }
    }
}
uint32_t prbs()
{
            // This is an unverified 31 bit PRBS generator
            // It should be maximum length but this has not been verified
            unsigned long new_bit=0;
            new_bit= ((shift_register & (1<<27))>>27) ^ ((shift_register & (1<<30))>>30);
            new_bit= ~new_bit;
            new_bit = new_bit & 1;
            shift_register=shift_register << 1;
            shift_register=shift_register | (new_bit);
            return shift_register & 0x7fffffff; // return 31 LSB's
}
```