1. Introduction

The purpose of this lab is to gain some familiarity with the timer/counter peripheral and time division multiplexing on an AVR microcontroller. Note that with timers we will be discussing time and period of a cycle which are measured in seconds (s) or milliseconds (ms), which is one thousandth of a second. Frequency is the measure of how many periods of a cycle occur per second, and is measured in Hertz (Hz). Frequency is the reciprocal of period.

Another important point to be aware of before starting this laboratory is that AVR microcontrollers support only signed and unsigned integers. For this laboratory therefore, any variables that you use for calculations should be of the type uint8_t, uint16_t, int8_t or int16_t, and not of type float or double. The 8 and 16 refer to whether the integer is 8 or 16 bits respectively, and the u prefix is for unsigned integers.

2. Getting started

2.1. Boot Linux and log in.

2.2. Open a bash shell and navigate to the uckf4 directory that you downloaded last week.

2.3. Just like last week, list the directory contents using ls -a.

2.4. Like last week, you will be working in the ence260-ucfk4/labs directory.

3. Flashing an LED

You did this last week, right? Well, rather than use a button, this week we are going to do something similar but at a slightly more precise rate whilst giving your finger a rest.

Microcontrollers, such as the ATmega32u2, have built in timer/counter peripherals and we are now going to learn how to control them. Your task is to flash an LED at a specific rate (or frequency). We will use timer/counter1 of the AVR microcontroller. This is a 16-bit timer/counter peripheral with three **8**-bit control registers TCCR**1**A[1], TCCR**1**B, and TCCR**1**C. The bits of these registers are used to control some advanced functionality of this timer but, at this stage, we are only interested in the lower three bits of TCCR**1**B. These three bits are responsible for controlling the prescaler's clock divider, which governs how fast the timer runs.

3.1. Open the file, lab2-ex1/lab2-ex1.c and you will see a partially completed program that will flash an LED.

3.2. Before we can successfully run this LED program we need to do some initialisation. For example, let's use timer/counter1 and program a flash rate into this timer. In the initialisation section of lab2-ex1.c, type in the following:

```
TCCR1A = 0x00;

TCCR1B = 0x05;

TCCR1C = 0x00;
```

[1] Timer/Counter1 Control Register A.

The above code will tell the AVR microprocessor to divide the main 8 MHz clock frequency by 1024 for Timer/Counter1. This is performed by placing 0x05 in TCCR1B, however there are many other frequency options for Timer/Counter1. To see these, refer to Table 16.5 (Page 133) of the ATMEGA32U2 datasheet located on Learn. Note that in Table 16.5, *n* refers to the timer/counter number, i.e., *n*=1 in our case since we are using timer/counter1. Also note that if TCCR1B = 0x00, timer/counter1 will not work!

3.3. To get the value of the timer/counter, the AVR microcontroller must read the TCNT1 register. This is a 16-bit register that counts from 0 to 65535 (2^{16}-1) automatically, i.e., you don't have to increment this, and when it reaches the maximum count it rolls back to 0 again. To reset the timer at any time along the way, a value of zero can be written into this register.

3.4. In the sections of code where a delay is required, start by resetting the timer/counter and then write a while loop that waits until the value of TCNT1 gets to a certain value. We want each delay to last 500 ms (0.5 seconds). This will make the LED flash at approximately 1 Hz (one cycle per second). To calculate the value of TCNT1 needed to get this delay we need to first consider the frequency a 10-bit (2^{10}=1024) prescaler would give us. Based on a CPU frequency f_{Clk} = 8 MHz clock, the frequency of the timer-clock f_{timer} is given by

$$f_{timer} = \frac{f_{Clk}}{prescaler} = \frac{8 \times 10^6}{1024} = 7812.5\ Hz$$

TCNT1 will increment every Δt seconds, where Δt is the clock-timer resolution. We can calculate Δt as the reciprocal of the timer-clock frequency

$$\Delta t = \frac{1}{f_{timer}} = \frac{prescaler}{f_{Clk}} = \frac{1024}{8\times10^6} = 0.000128\ \text{s} = 0.128\ ms = 128\ \mu s$$

3.5. We get the value of TCNT1 by dividing the period by the resolution ie

$$TCNT1(Ticks) = \frac{Delay}{\Delta t} = \frac{0.5}{0.000128} = 3906$$

The value 3906 is the value we have to wait until TCNT1 reaches before we can exit the while loops for the LED being on, and off again.

3.6. Note that you should try and use the 16 bit unsigned integer data type uint16_t for any variables you use to calculate the number of counts.

3.7. Compile and program your code the same way you did in the previous lab. You should see the LED flashing slowly.

4. Pulse width modulation

What do you think would happen if we increase the flash rate above 50 Hz? And what would happen if we reduce the duty cycle (the ratio of the period that the LED is on to the total flash period)? Let's find out.

4.1. Edit the lab2-ex1/lab2-ex1.c file and modify it so that the LED flashes at 100 Hz? What happens?

4.2. Now change the duty cycle, so that the LED is on for one quarter of the flash period and off for three quarters of the flash period. What do you observe?

Altering the duty cycle is called *pulse width modulation* (PWM). It is a technique for varying the average output voltage. Note that in this laboratory that we are achieving PWM via software by changing the duty cycle of PC2, rather than through hardware.

5. Roll your own timer module

We saw in the previous lab that it is preferable to create modules for handling specific functionality (It hides the gory details behind an abstraction). We will now create a timer/counter module.

5.1. Open the lab2-ex2/lab2-ex2.c file. You will see a program that flashes the LED but this time uses timer/counter functions.

5.2. Open the lab2-ex2/timer.c file and implement the functions timer_init() and timer_delay_ms(). Note that the timer_delay_ms() function takes an argument of the delay period. This argument is in milliseconds so you will need code to convert the value in milliseconds into a value for the TCNT1 register.

5.3. Compile your code and program it onto your UCFK4 board. You should see the same behaviour as the previous exercise. Note how much easier it is to get the desired delay when you have a function that takes an argument in a real time unit!

6. Paced loops

In the first exercise our intended period of flashing was one second. This was, unfortunately, not quite the case. The functions to turn on/off the LED take time (albeit small) to run. These contribute to the delay of the loop making the period ever so slightly longer than one second. As your program runs (especially at high flash rates), this timing error grows and will eventually lead to the system being out of time. This will be exacerbated if a more time consuming operation were performed. We will now create what is known as a paced loop. This makes sure that the task is repeated with accurate timing, independent of the duration of the task[2].

Previously, in the timer_delay_ms() function, we reset the timer just before the delay loop. In the case of a paced loop, we reset the timer just after the delay. This means that the timer is counting up while other code is executing in the loop and this time is counted towards the delay.

6.1. Open the lab2-ex3/lab2-ex3.c. You will see a program that flashes the LED but, this time, it uses a paced loop to do it.

6.2. Edit the lab2-ex3/pacer.c file and implement the empty functions. The code for the pacer_init() function will be mostly the same as in the delay module from the previous exercise but will also need to take the pacer frequency value and calculate the counter value from this. The delay (in seconds) is 1/pacer_frequency (in Hz). The pacer period in ticks is then given by:

[2] Provided that the task duration is shorter than the repetition period.

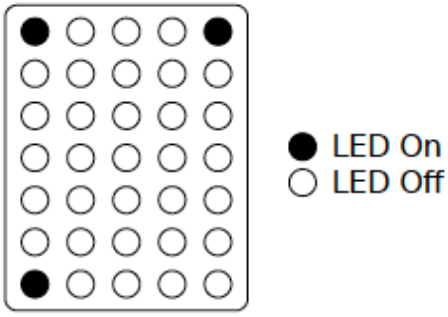
$$Ticks = \frac{Delay}{\Delta t} = \frac{1}{pacer\_frequency \times \Delta t} = \frac{f_{clk}}{pacer\_frequncy \times prescaler}$$

6.3. The pacer_period_ticks variable is stored in a static variable so the pacer_wait() function can access it. Take care writing the pacer_wait() function. Make sure you reset the timer in the correct place.

6.4. Compile and program your code. You should see the LED flashing again but this time the period will be more accurate[3].

7. The three corner problem

Your goal here is to turn on all but one of the corners of the LED matrix. This is a bit of a problem as you can only drive complete rows and columns at a time, making it easy to drive all four corners at once but not just three. The trick to achieve this is called time division multiplexing! The diagram below shows the desired pattern.



7.1. Open the lab2-ex4/lab2-ex4.c file. Here is a basic paced loop program in which you will need to drive the LED matrix. Note that the frequency of the paced loop is high (100Hz). This is so that when we drive alternating rows and columns of the LED matrix, we get the illusion that the LEDs are constantly on. You need to include your pacer module from the previous exercise (the pacer module in the lab2-ex4 directory has empty but compilable functions).

7.2. Start by initialising the row and column pins using the functions in pio.h, as you did last week. Remember these need to be set high to be in their off state.

7.3. In the paced loop we need to toggle between two pin configurations. This can be achieved with a variable and a conditional statement. In one of the pin configurations, set the first and last row pins to low, and all others high. Also set only the first column pin to low. In the second pin configuration, set only the first row pin to low. Also set the final column pin to low, all others high.

7.4. Compile and program your UCFK4 board. Each time the microcontroller goes through the loop, it should display only one of the configurations and it should be fast enough to trick the eye into seeing the three corner LEDs on constantly. If you see your LEDs flashing, increase the pacer frequency. Note that because we are toggling between 2 states each of which is occurring at 100Hz, we have an overall frequency of 50 Hz alternating between the 2 states.

[3] You would require an oscilloscope to check this.

7.5. This technique relies on persistence of vision. The momentary flash of each LED persists on the retina such that it seems on longer than it actually is. If the LED comes on again before the eye registers it as off, it appears that it is constantly on. Try moving the UCFK4 board rapidly back and forth in front of you while looking past it with one eye. This results in different parts of the retina being stimulated over time and you should be able to see broken lines where the LEDs move across your vision. The dark parts of these lines are when the LED is off!

8. Arbitrary pattern

Now we'll try something a little more complicated. We are going to build a program that can display any generic pattern on the LED matrix. To do this, we will need to iterate through arrays containing reference to the rows and columns and, at each point, read a value from a bitmap containing the shape we wish to display.

8.1. Open the file lab2-ex5/lab2-ex5.c. You will see arrays of the PIO definitions for the rows and columns. You will also see an array of bitmap data along with a function waiting to be filled.

8.2. First, in the main() before the while(1) loop, you need to initialize the LED pins. You can use a for loop over rows and cols to do this. The initial state should be set at PIO_OUTPUT_HIGH so that the LED array elements are initially off.

8.3. Secondly, in the empty function, write a for loop to iterate over each row in the considered column (you are looping over the 7 row bits of a given column). This loop will need to take the row bit mask and identify which rows need to be set low (active). The following code snippet may be useful.

```
if ((row_pattern >> current_row) & 1)
{
    pio_output_low (rows[current_row]);
}
else
{
    pio_output_high (rows[current_row]);
}
```

8.4. Next, this function must set the proper column each time it is called. We will need a static variable that remembers the previous column so that it can be set high. The current column number is passed as an argument and should be set low.

8.5. A physical phenomenon that you may encounter is that of *ghosting*, where you may notice that some LEDs of the previous column are faintly illuminated. In order to avoid this ghosting, you should avoid changing the row pattern while the previous (or current) column is active.

8.6. Compile and program your code. If the function you just implemented is working correctly, what do you see on the LED matrix?

8.7. Try making your own patterns by editing the bitmap array.