# Assignment 1, Part 2
More Blinking Lights
**Due Friday, 29 January 2016**

## 1 Summary

In the first part of the assignment, you turned LEDs on and off, either in a systematic fashion or in response to pressing pushbuttons on the LaunchPad. The method you used is referred to as "polling," where the port is repeatedly queried to find it's status. For time-sensitive processes (i.e. those where something has to happen immediately upon a given condition being realized) or in low-power operations, polling is undesirable. There is a large *latency* in polling, unless the processor is fully restricted to the polling operation (in which case, the process blocks while waiting for the condition). In low-power operation, the process of polling prevents putting the CPU to sleep, increasing the power draw of the system.

In order to avoid these issues, processors allow us to use *interrupt-driven* code. The processor has separate lines to the control system which override normal operation of the program, and jump execution to separate code blocks based on which line is triggered. The triggers are *interrupts* and the code which handles an interrupt is an *interrupt handler.* There are two ways of generating interrupts: hardware interrupts and software interrupts. A hardware interrupt is triggered by an external condition, usually a pin state changing. A software interrupt is a way of forcing code execution to the interrupt handler, followed by return to the normal execution point. Older systems (DOS) used software interrupts for most of their I/O routines: to write a character to the screen, for instance, the code for that character was loaded into a register and in interrupt invoked.

Timing via delay loops, as you did in Part 1, is difficult to tune and subject to change when operating conditions change (a new processor may have a slightly different clock frequency, or any of a myriad of other effects may change the loop timing). So we prefer to use timer-based code when possible, as it is more predictable (although it can still vary if the oscillator frequency drifts, e.g. due to temperature changes).

In this part of the assignment, you will learn the basics of the MSP432 clock system and how to handle interrupts. Your resulting program will be a "Dolan clock."

## 2 Interrupts on the MSP432

Fortunately, interrupts on the MSP432 are simple to get started with (life gets a lot harder when multiple interrupts can occur nearly simultaneously- in the words of my old officemate, "we'll blow up that bridge when we get to it.") The MSP432 has a set of memory locations which contain pointers to the interrupt handlers (this is described in the datasheet around page 80). In order to add an interrupt handler, you create the interrupt handler function and then set the value in the address table to point to that function. In order to avoid random voltage drifts constantly diverting the processor, interrupts are disabled by default, and must be turned on using the *nested vector interrupt controller*, or NVIC. This process is not particularly difficult, but it is made easier by the files CCS generates for each new project.

### 2.1 Using NVIC Interrupts

The signature of the interrupt handler should be

```
void InterruptHandler(void);
```

Write the code for the interrupt handler. Then, in the file **msp432_startup_ccs.c**, add a declaration for this function (after line 57 in the auto-generated file):

```
extern void InterruptHandler(void);
```

so the compiler knows that the function is defined in another module. Starting in line 63 of the auto-generated file, you find a declaration:

```
#pragma RETAIN(interruptVectors)
#pragma DATA_SECTION(interruptVectors, ".intvecs")
void (* const interruptVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_END),
                                            /* The initial stack pointer */
    resetISR,                               /* The reset handler         */
    nmiISR,                                 /* The NMI handler           */
    faultISR,                               /* The hard fault handler    */
    defaultISR,                             /* The MPU fault handler     */
```

where all of the interrupts are listed. At the interrupt you want to handle, simply put the name of the interrupt handler function in.

You also need to enable the interrupt overall. In your startup code (either in **main** or one of the initialization routines called from it), include the line

```
NVIC_EnableIRQ(IRQEnum);
```

where *IRQEnum* is the value of type **enum IRQn** for your given interrupt. This enum type is defined in **msp432p401r.h** starting in line 84.

### 2.1.1 Example

We want to handle interrupts generated by pressing the pushbuttons on the LaunchPad. We know these to be connected to Port 1, so we want to use our handler as the handler for the interrupt PORT1_IRQn. In our initialization code, we have the line

```
NVIC_EnableIRQ(PORT1_IRQn);
```

We have somewhere the handler defined, e.g.

```
void PortOneInterrupt(void)
{
/* somebody pushed a button */
}
```

In **msp432_startup_ccs.c**, we add the line

```
extern void PortOneInterrupt(void);
```

and then change the line which reads

```
    defaultISR,                             /* PORT1 ISR                 */
```

to read

```
    PortOneInterrupt,                       /* PORT1 ISR                 */
```

When an interrupt is generated for Port 1, control will be passed to the function **PortOneInterrupt**.

## 2.2 Enabling interrupts

There are multiple levels at which we need to enable interrupts for our interrupt handler to be called. It is possible to disable all user interrupts (fault interrupts will still be called); on the MSP430 series, it is necessary to actively enable them globally. Each interrupt which we wish to use must be enabled as seen above. Finally, the subsystem which triggers the interrupt can enable or disable specific interrupts related to that subsystem. There is a flag register which contains information about the specific condition which raised the interrupt. (It's easier to do than to describe simply.)

We want to trigger interrupts when a pushbutton is pressed. We know that the pushbuttons are on P1.1 and P1.4. The I/O ports have interrupt enable registers **PxIE**. We want to enable the pushbuttons, but no other lines on this port. So we set this register to have values of 1 in bits 1 and 4, and values of 0 at all other pins by placing into our initialization code the line

```
P1IE=(BIT1|BIT4);
```

There are two ways an interrupt can be triggered on an I/O port: when the port transitions from a low state to a high state, or when the port transitions from a high state to a low state. Which one triggers the interrupt is governed by the **PxIES** register. The pushbuttons on the LaunchPad are held high via the pull-up resistor, and forced low when the button is pressed, so we want to trigger on the high-to-low transition. This direction is selected when the appropriate but of the **PxIES** register is set to 1, so we include the line

```
P1IES|=(BIT1|BIT4);
```

in the initialization code. Generally it is better to configure before enabling, to avoid unexpected interrupts.

## 2.3 The Interrupt Handler

The interrupt handler contains the code which determines the response to the interrupt. In general, we have to determine which of several possibilities raised the interrupt. In the case of the pushbuttons, we need to determine which button was pressed. In general on the MSP432, there is an interrupt vector register which contains that information. For the port interrupt, this register is **PxIV**. It is a 16-bit register on the MSP432, but most of the bits are unused. This register will contain a value of twice the number of the *highest-priority* line raising the interrupt. The status of the interrupt flags can be read via the interrupt flag register **PxIFG**, which contains a 1 in a given bit if the corresponding line has an interrupt raised. The interrupt flag needs to be cleared when the interrupt is handled. Otherwise, when control is passed out of the interrupt handler, the interrupt will still be present, and control will immediately return to the handler.

Reading the value of the interrupt vector register will clear the flag for the highest-priority interrupt. If the two events can be handled sequentially, this is the easiest way to work: handle the highest-priority interrupt first, then return control and let the interrupt handler be called again with the next value in **PxIV**. Otherwise, querying and resetting the flags in **PxIFG** must be called.

### 2.3.1 Example: Toggling the Red LED

In the simplest case, we want to toggle the red LED (P1.0) whenever either button is pushed. We don't care which button, but we need to clear the interrupt flag. Our interrupt handler would then be

```
void PortOneInterrupt(void)
{
  unsigned short iflag=P1IV;
  P1OUT^=BIT0;
}
```

Since we are toggling the LED inside the handler, we don't need to do it in **main**. Nor do we need the delay loop to fix issues with the button staying down, as there is only one transition per button press (although it appears there is some "bounce" to the pushbuttons). Our **main** function thus gets even simpler:

```
void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;              // Stop watchdog timer
 /* Initialize the I/O port */
    InitializeLED();
    InitializePushButton(1);
    InitializePushButton(4);
    P1IE=(BIT1|BIT4);
    NVIC_EnableIRQ(PORT1_IRQn);
 /* Now we enter the loop */
    for(;;)  //idiomatic infinite loop; while(1) is also usable here
    {
    }
}
```

In fact, our main loop is empty. Once we initialize the LED and the pushbuttons, we enable interrupts on port 1 (for lines 1 and 4, the pushbuttons) by setting the corresponding bits in the **P1IE** register. Then we enable the port 1 interrupts for the program through the NVIC.

# 3 Timers

For many purposes, we want events to happen at specific times. The MSP432 has 4 16-bit timers, 2 32-bit timer, and a real-time clock that we can use for timing operations. For now, we'll use one of the 16-bit timers, which the MSP432 documentation refers to as Timer_A. The Timer_A features are discussed in Section 17 of the MSP432 Family User's Guide.

Timers are essentially simple devices. They take a clock signal and use it to count- in this case, by incrementing the value of a register every time the clock signal goes high[1]. The period of the timer is determined by a value set in a control register. When the counter register is equal to this value, the timer resets[2] We can thus control the period of the timer by writing an appropriate value into the control register.

The simplest use of a timer, then, is to raise an interrupt every time the timer resets. As an example, we will use this to blink our red LED on or off every second. To do this, we want the timer to have a period of 1 second.

## 3.1 The Timer Registers

There are several timer registers which determine the base counting frequency, mode of operation, and behavior of a timer. There is also a register which contains the actual count (i.e. the timer value) and registers related to timer-based interrupts. Like most things on the MSP432, there is nothing hard about using timers, but a little attention to detail is required. In this example, I will use the timer Timer_A0 (the other 16-bit timers are Timer_A1, Timer_A2, Timer_A3, and Timer_A4, with the obvious change in register nomenclature).

### 3.1.1 The Timer Control Register

The timer control register, **TA0CTL**, is a 16-bit register which determines which system clock signal is used to determine the count frequency, sets the counter frequency to either that frequency or a lower one obtained by dividing the frequency by a power of two, and selects the mode of operation. It also allows us to disable the timer, enable interrupts, and reset the timer. The details of this register are explained in Figure 17-15 and Table 17-4 on page 603 of the User's Guide.

The highest 6 bits are marked *reserved*, meaning they have no specific purpose in the MSP432 but may be used in future versions of the chip. We should basically leave them alone, as writing to them may have unpredictable results. Bits 9 and 8 (marked **TASSEL** in the table) determine which clock signal is used to determine the input frequency. There are two internal clocks which can be selected, **SMCLK** and **ACLK** (we'll look at these next), and two external clock inputs which could be selected. There are good reasons to use external clocks at times, but we don't need to concern ourselves with that for now.

Once the clock is configured (*vide infra*), we select the input clock and set the divider using the **TA0CTL** register. This 16-bit register controls the selection of the input clock and frequency divider, the mode (and disabling) of the timer, and allows for resetting the timer and enabling an interrupt when the timer reaches the top of its range (timer overflow). It also has a bit to indicate whether an interrupt signal is pending for the timer overflow.

We want to configure the timer to run off **ACLK** with no extra divider. Since we will set the **ACLK** clock to run at 128kHz, this will make the timer increment 128000 times per second. The clock is selected by writing a value of 01b to the **TASSEL** field (bits 9 and 8). The internal divider is set to 1 by setting the **ID** field (bits 7 and 6) to 0. Until everything else is set, we disable the timer by setting the **MC** field (bits 5 and 4) to 0, and we leave the other fields set to 0. So we use the line

[1]The MSP432 16-bit timers have a mode- the Up/Down mode- where this is not true. Instead, the timer counts up to some designated value and then counts back down.

[2]Again, this is an oversimplification. In the up/down mode, the timer counts up to the register value, then down to zero, then back up, and so on. In the continuous mode, the counter counts until the counter overflows.

```
TA0CTL=0x0100;
```

to establish the timer. We will enable it, reset it, and enable the verflow interrupt when we are done setting the period (trying to establish all the settings, particularly with interrupts enabled, on a running timer is a bad idea). To do this, we will set the **MC** field to choose a count-up timer (01b), enable the overflow interrupt by writing a value of 1 to the **TAIE** field (bit 1), and reset the timer by writing a 1 to the **TACLR** field (bit 2). The line for this is

```
TA0CTL=0x0116;
```

The period of the timer in count-up mode is determined by the value in the **TA0CCR0** register. We need to tell the processor how to interpret this field first, however, by setting the value of the **TA0CCTL0** register. Each Timer_A timer has 6 capture/compare registers (CCRx) which can be used to either capture the time of an external event or to trigger an external event when the timer register compares equal to the value in the capture/compare register. We need to configure the capture/compare register to be capture or compare, to resolve some details about how the clock resolves when a capture event occurs, to determine is an external output is directly driven by the clock event, and to control whether an interrupt is raised when the capture or compare event occurs. These parameters are set via the **TA0CTLx** registers. For now, we will only use the **TA0CCTL0** and **TA0CCR0** registers to set the frequency of the timer. The **TAxCCTLn** registers are described in Figure 17-17 and Table 17-6 of the User's Guide.

We want to use this register in compare mode, so we set the **CM** field (bits 15 and 14) to 0. Since we are not capturing, the input source is unimportant to us; we set it to GND by setting the **CCIS** field (bits 13 and 12) to 10b. Similarly, the **SCS** and **SCCI** fields relate to how the clock is latched on capture; we set the corresponding fields to the default values of 0. We set the value of the **CAP** field (bit 8) to 0 to select compare mode. The **OUTMOD** field (bits 7-5) selects the output mode. This output is mapped to an output line of the chip and can be used for direct control of, e.g. a motor. For this case, we are not using the output, and it is of no particular interest to us. We set it to be driven by the output pin by setting this field to 0. The **OUT** field (bit 2) thus directly controls the output state; we set it to 0. The **CCIE** field enables the compare interrupt. For the particular case of the **CCR0** register in count-up mode, this offers no real benefit over using the overflow interrupt. We thus leave the interrupt disabled by setting the **CCIE** field (bit 4) to 0. The **CCI** field (but 3) is read-only, so we use a value of 0 for this bit. The last two bits tell interrupt status, and we set them to 0 (they will be set when interrupt conditions are met, even if interrupts are disabled). The resulting value of the register thus has the binary representation
00100000 00000000
with the hex representation 0x2000. We set this register with the line

```
TA0CCTL0=0x2000;
```

The value we put in the **TA0CCR0** register will now control the period of the timer. We want to have the LED blink once per second, so we want the interrupt to be raised every half-second. Since the clock counts at 128000 counts per second, we want to set a value of 64000 into this register to achieve this. We can do this with the decimal value directly

```
TA0CCR0=64000;
```

I often use hex values for registers; the hex representation of 64000 is 0xFA00. One reason for converting to hex directly is to ensure that the value will fit into the register- the compiler will likely not issue a diagnostic if you try to write too big a number. The highest value attainable for a 16-bit counter is 65535.

Our timer initialization routine then looks like

```
void ConfigureTimer(void)
{
  TA0CCTL=0x0100;
  TA0CTL0=0x2000;
  TA0CCR0=0xFA00; //or TA0CCR0=64000
  TA0CTL=0x0116;
}
```

### 3.1.2 The Clock Input

The two internal clocks are described in Section 5 of the User's Guide. For now, it is easier to use the **ACLK** signal to drive the timer; this clock can be run off any of three oscillators as described on page 292 of the User's Guide. The first choice, **LFXTCLK**, is an external oscillator input. The second choice, **VLOCLK**, is the very-low frequency clock, which runs at a maximum of 10kHz. This oscillator is useful if you want to run an infrequent cycle. The final choice is to use the **REFOCLK**, which can run at a rate of up to 128kHz. This clock is what we will use.

To configure the **ACLK** signal, we need to set the clock control registers to link **ACLK** to the **REFOCLK** oscillator, and ensure that **REFOCLK** is running at 128kHz. The process for doing this is established in Section 5 of the User's Guide.

Before we can set any of the clock control registers, we have to unlock them. Unlocking the registers is accomplished by writing a specific value (0xA596) to the **CSKEY** field of the **CSACC** register:

```
CSKEY=0x695A ;
```

When we are done adjusting the clock control registers, we reset this value to avoid inadvertant resetting of clock parameters:

```
CSKEY=0xA596 ;
```

Reading the **CSKEY** field will always yield the value 0xA596.

Once we have unlocked the control registers, we can set the values we need to control the **ACLK** signal. Doing so requires us to set two fields in the **CSCTL1** register (Figure 5-7 and Table 5-5 on pages 309-310 of the User's Guide). We will set the **DIVA** field (bits 26-24) to 0. We could use these to lower the **ACLK** frequency by dividing by the selected power of two. We select the **REFOCLK** signal as the source for **ACLK** by setting the value of the **SELA** field (bits 10-8) to 2 (010b). For now, we can set the rest of the bits to the default values indicated in the table (all 0 except for bits 5 and 4 of the **SELS** field and bits 0 and 1 of the **SELM** field. Our resulting value of the register thus has the binary representation
00000000 00000000 00000010 00110011
or, converting to hex: 0x00000233 (each 4 bits in the binary representation yield one digit in the hex representation).

Finally, we need to set the **REFOCLK** frequency and enable the **ACLK** clock. These tasks are accomplished by setting the appropriate fields of the **CSCLKEN** register (Figure 5-10 and Table 5-8 on page 314 of the User's Guide). The **REFOFSEL** field (bit 15) selects the frequency of the **REFOCLK** signal. If it is 0, the clock runs at 32.768kHz, and if it is 1, the clock runs at 128kHz. For our purposes, it is more convenient to choose the higher frequency. The **REFO_EN** field (bit 9) enables the **REFOCLK** oscillator. If this bit is 0, the oscillator is enabled if the **REFOCLK** oscillator is used as an input source for a clock (e.g. **ACLK**). If this bit is 1, the **REFOCLK** oscillator is always active. Since we are using this oscillator, it will be enabled either way. In the spirit of low-power operation, we leave this bit 0, so that if we chose to disable the timer, the oscillator would not be driven (and hence not consuming power). The last field of interest to us is the **ACLK_EN** field, which must be 1 for the **ACLK** clock to be enabled. We put the other fields in their default settings as indicated in Figure 5-10 and Table 5-8); the binary representation is
00000000 00000000 10000000 00001111
and the hex value is 0x0000800F.

We thus have a four-line function to establish the clock settings for our timer:

```
void SetClockFrequency(void)
{
  CSKEY=0x695A ;
  CSCTL1=0x00000233 ;
  CSCLKEN=0x0000800F ;
  CSKEY=0xA596 ;
}
```

We invoke this function from **main** before we configure Timer_A0.

### 3.2 Using the overflow interrupt

There are two interrupt signals which can be raised by Timer_A0. One is raised when the compare/capture interrupt on **TA0CCR0** is triggered, *viz* the **TA0_0** interrupt. The other is raised for all the other compare/capture interrupt, including the overflow interrupt. This interrupt, which is the one we are using, is the **TA0_N** interrupt. We need to enable it with NVIC:

```
NVIC_EnableIRQ(TA0_N_IRQn);
```

All we want to do is toggle the red LED when the interrupt is raised. Out interrupt handler is thus very simple. We query the interrupt vector register (**TA0IV**, Figure 17-19 and Table 17-8, page 607 of the User's Guide) to ensure that we have an overflow interrupt (**TA0IV**=0x0E) and then toggle the port for the red LED. Reading **TA0IV** resets the interrupt flag.

```
void TimerA0Interrupt(void)
{
  unsigned short intv=TA0IV;
  if(intv==0x0E)
    P1OUT^=BIT0;
}
```

As before, we add the line

```
extern void TimerA0Interrupt(void);
```

to the file **msp432_startup_ccs.c** and set the value of the **TA0_N** interrupt vector to this function (line 94 of the startup file).

Our main program is thus

```
#include "msp.h"

void SelectPortFunction(int port,int line,int sel0,int sel1)
{
        if(port==1)
        {
                if(P1SEL0 & BIT(line)!=sel0)
                {
                        if(P1SEL1 & BIT(line)!=sel1)
                                P1SELC|=BIT(line);
                        else
                                P1SEL0^=BIT(line);
                }
                else
                {
                        if(P1SEL1 & BIT(line)!=sel1)
                                P1SEL1^=BIT(line);
                }
        }
        else
        {
                if(P2SEL0 & BIT(line)!=sel0)
                {
                        if(P2SEL1 & BIT(line)!=sel1)
                                P2SELC|=BIT(line);
                        else
                                P2SEL0^=BIT(line);
                }
                else
```

```
                {
                        if(P2SEL1 & BIT(line)!=sel1)
                                P2SEL1^=BIT(line);
                }
        }
}

void InitializeLED(void)
{
        P1DIR|=BIT0;
        SelectPortFunction(1,0,0,0);
        P1OUT|=BIT0;    //turn the LED on
}

void SetClockFrequency(void)
{
        CSKEY=0x695A;
        CSCTL1=0x00000233;
        CSCLKEN=0x0000800F;
        CSKEY=0xA596;
}

void ConfigureTimer(void)
{
  TA0CTL=0x0100;
  TA0CCTL0=0x2000;
  TA0CCR0=0xFA00; //or TA0CCR0=64000
  TA0CTL=0x0116;
}

void TimerA0Interrupt(void)
{
        unsigned short intv=TA0IV;
        if(intv==0x0E)
        P1OUT^=BIT0;
}

void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;               // Stop watchdog timer
    SetClockFrequency();
    InitializeLED();
    ConfigureTimer();
    NVIC_EnableIRQ(TA0_N_IRQn);
    for(;;);
}
```

### 3.3 Controlling intensity through pulse width

It is also possible to control the intensity of the LED. We do this by pulsing it off and on at a much higher rate than our eyes can distinguish. By varying the fraction of the cycle in which the LED is on, we perceive an intensity difference.

We can use a very similar timer arrangement to that above. We will reset the timer period to a much smaller value; this will determine the length of the off/on cycle. We will turn the LED off every time the overflow interrupt triggers. Changing the third line of **ConfigureTimer** to

```
TA0CCR0=0x0080; //or TA0CCR0=128
```

will cause an interrupt to be raised every millisecond, which should be fast enough for our purposes. It allows us to adjust the intensity from off to fully on in 128 steps, which is more than I can distinguish.

We need to introduce a second compare register which we will use to turn the LED on. We do this just like we did for the main compare register. In this case, we will use capture/compare register 1. The only difference to the capture/compare register 0 case *supra* is that we want to enable the compare interrupt by setting the **CCIE** field of the **TA0CCTL1** register (bit 4) to 1. Then we set the value of the **TA0CCR1** register to the desired value to turn on the LED. The closer this number is to 128, the dimmer the LED will be. Our initialization routine thus adds the lines

```
TA0CCTL1=0x2010;
TA0CCR1=0x0080;
```

Changing the value of **TA0CCR1** changes the intensity of the LED. Setting it to 0 would turn the LED fully on, setting it to 0x007f would give the dimmest setting. Setting it to 0x0080 actually turns it off because the two interrupts fire simultaneously, and the CCR1 interrupt has priority. The interrupt handler gets called twice in succession, the first time turning the LED on and the second turning it off. Neither fires until the end of the next cycle. Our updated timer initialization routine is thus:

```
void ConfigureTimer(void)
{
  TA0CTL=0x0100;
  TA0CCTL0=0x2000;
  TA0CCR0=0x0080; //or TA0CCR0=128
  TA0CCTL1=0x2010;
  TA0CCR1=0x0080;
  TA0CTL=0x0116;
}
```

We also need to change the interrupt handler. Now we want to turn the LED on when the CCR1 CCIFG interrupt is raised, which would set the value of **TA0IV** to 0x02. We turn it off when the overflow interrupt is raised, with **TA0IV** being set to 0x0E. We thus use the code

```
void TimerA0Interrupt(void)
{
        unsigned short intv=TA0IV;
        if(intv==0x02)
          P1OUT|=BIT0;
        if(intv==0x0E)
          P1OUT&=~BIT0;
}
```

This allows us to change the intensity of the LED by changing the value of **TA0CCR1**. It would be nice to be able to control this value without changing the program. It is easy enough to accomplish this task.

To change the intensity within the program, we need only reset the value of **TA0CCR1**. I will illustrate this by making the LED intensity ramp from dimmest to brightest. We need two static variables in the interrupt handler; one to count the number of on/off cycles and one to control the intensity setting. Since we want to start from the dimmest possible, we initialize the intensity to 0x0080, or fully off, and count down to 0, or fully on. We change the intensity every 10 cycles, or every 10 milliseconds. The code is

```
void TimerA0Interrupt(void)
{
        static int intCycles=0;
        static unsigned short intensity=0x80;
```

```
        unsigned short intv=TA0IV;
        if(intv==0x0E)
        {
                if(++intCycles==10)
                {
                        intCycles=0;
                        if(intensity==0)
                                intensity=128;
                        else
                                intensity-=1;
                        TA0CCR1=intensity;
                }
                P1OUT&=~BIT0;
        }
        else if(intv==0x02)
                P1OUT|=BIT0;
}
```

# 4 Assignment 1, parts 2, 3, and 4

Modify your colored blinking light program from part 1 to use interrupts.

Use the timer to change the color of the RGB LED in two different ways. In the first case, step through the 7 basic colors in sequence, so that each color is displayed for 1 second at a time, and then the LED is off for 1 second.

In the second case, use the timer to control the intensity of the component LEDs to explore the colors which you can distinguish. Decide on 9 colors with different RGB intensities and cycle through those colors. Identify each color with a triple (RGB) of timer values: (e.g. {0x10,0x50, 0x78}). Use the register **TA0CCR1** for the red intensity, **TA0CCR2** for green, and **TA0CCR2** for blue. The values of **TA0IV** will be 0x02 (red), 0x04 (green) and 0x06 (blue): turn the corresponding LED on when the interrupt is called. Turn them all off when the overflow interrupt (**TA0IV**=0x0E) is raised.

Your programs should be submitted via Moodle by class time on Monday, 1 February.