Gustavo Da Costa

ENCM 369 PIC Activity 5

ENCM 369 ILS Winter 2021 Version 1.1

Activity Objectives

- Understand significance of timing in an embedded system
- Implement the Timer peripheral on the PIC to provide a variable time function
- Use arrays to hold light patterns and index this array to implement the patterns on your LEDs
- Practice following coding requirements and code review of a peer

Deliverables

- Light pattern design plan
- Check-in of you code to Github
- Completed rubric review of someone's code in your learning community

Background Information

Timing is very often critical to an embedded system and has many implications in hardware, firmware design/operation, and even power consumption. We can just scratch the surface here and we'll focus this activity on using a hardware feature of the PIC that is an independent timer that you can configure and then let run without any attention from the main processor. This keeps the main processor available to run other parts of code or catch up on its sleep. When the timer is done counting, it will tell the processor much like your alarm clock ringing in the morning to wake you up.

Perhaps the most important concept to take from this activity is the concept of "blocking" which typically refers to a task or function in a program that takes 100% of the processor time such that no other tasks or functions can execute. Though we so far only have one task running in our system (aka UserApp), the activities have encouraged you to ensure the task executes quickly and returns to main. If your embedded system is running an operating system, then there is much less concern with what each task is doing and how long it is taking, since the OS can put the task on hold to service another task that is waiting. In our system there is no such luxury of an OS scheduler running and ensuring every task gets some processing time, so it becomes immensely important to design and write code that won't hog the processor. Systems without an OS are called "bare metal" systems for some reason. Of the hundreds of billions of embedded devices out there, it is safe to say that the majority are bare metal.

In this activity, you will write a non-blocking function that can be used to set a variable timer to clock a period between 1us and 65535us (can you guess the variable type of the function argument already?). This is far better than the library "delay" function that a few people discovered during previous activities. That function is blocking, and you also have 0 idea how it works.

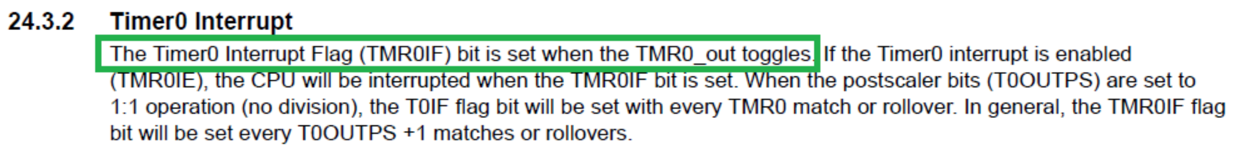
Hardware Design

There is no new hardware required for this activity, though you might want to change the color of some of your LEDs to make life more interesting. If you do change colors, do you think you should change the LED resistors, too? Why or why not?

Timer0 Module

Timers are very useful so most microcontrollers will include several of them. The PIC18F27Q43 has three timer peripherals/modules called Timer0, Timer1, Timer2. Each is slightly different and offers various features. You will use Timer0 for this activity.

Since you don't know anything about using Timer0, you need to read the PIC user guide to learn how to use it. Read pages 377-379 and focus on the basic operation and required registers for the timer. You should be able to figure out what a prescaler and postscaler are, and what registers the timer value lives in. We will use Timer0 in 16-bit asynchronous mode. A very important piece of information is in 24.3.2 that tells you that the TMR0IF bit will be set when the timer overflows.



24.3.2 Timer0 Interrupt

The Timer0 Interrupt Flag (TMR0IF) bit is set when the TMR0_out toggles. If the Timer0 interrupt is enabled (TMR0IE), the CPU will be interrupted when the TMR0IF bit is set. When the postscaler bits (T0OUTPS) are set to 1:1 operation (no division), the T0IF flag bit will be set with every TMR0 match or rollover. In general, the TMR0IF flag bit will be set every T0OUTPS +1 matches or rollovers.

This is how you will know that the timed period is up. Take a moment to find what register that bit lives in by searching for it in the user guide. Write down the name of the register.

The most difficult part of setting up a timer is translating the timer's clock source into real time. In other words, if you want the timer to be able to count in the ballpark of 1ms – 1s, how do you set up the timer configurating registers properly? There are several things to consider:

- What clock source will you assign to clock the timer module?
- Do you need to slow the clock down using a prescale or postscale value?
- How much real time actually passes with a single tick of the timer?
- How many timer ticks need to pass to count out the real time period that you want?

To keep things simple, you will clock Timer0 from the same clock source as the processor's clock. As you know, the processor oscillator frequency is 64MHz and executes instructions at Fosc/4 = 16MHz. So how long is one instruction cycle in nanoseconds? Hint: T = 1/f.

The goal of the timing function is to provide a variable timer between 1us and 65,535us. If you clock the timer directly from Fosc/4 and the timer can count up to 16 bits, can you achieve the objective timing using Fosc/4? Hint: How many clock cycles in 1us? How many clock cycles in 65,535us?

Since the timer can only count to 65,535 (because it is 16 bits), you should see that you will need to slow down the timer's clock frequency using a prescaler. The prescaler will divide the input clock frequency that the timer sees down to a slower value. If you're paying attention, you should already know by how much you need to prescale. If you don't know, figure it out now. The postscaler can remain 1:1 as we do not need to extend the timer any further.

So now we have the following information ready:
Timer clock mode: Asynchronous
Timer size: 16-bits
Timer clock source: Fosc/4
Timer prescaler: 1:16
Timer postscaler: 1:1

You'll also want to ensure that Timer0 is enabled. Using this information, look at the Timer0 registers and determine the configuration values you will need for the following registers:

Register	Binary Init Value	Hex Init Value
T0CON0	1001 0000	0x90
T0CON1	0101 0100	0x54

Timer0 Programming

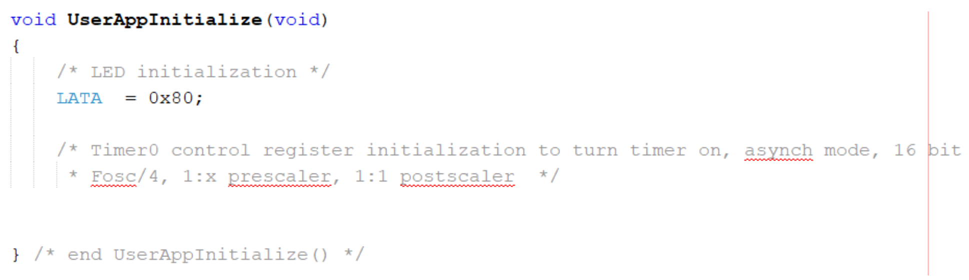
Branch your PIC_Activity3 (NOT PIC_Activity4) to PIC_Activity5. Delete most of the UserAppRun() code except the part that updates the LEDs.

Your LED update code must work as follows:

1. Read LATA to a temporary variable
2. Use a bitmask and bitwise operation to clear the 6 LSBs
3. Use a bitwise operation to update the 6 LSBs to the new value you want
4. Write the temporary variable back to LATA
5. Make sure all of this is done using 8-bit variables.

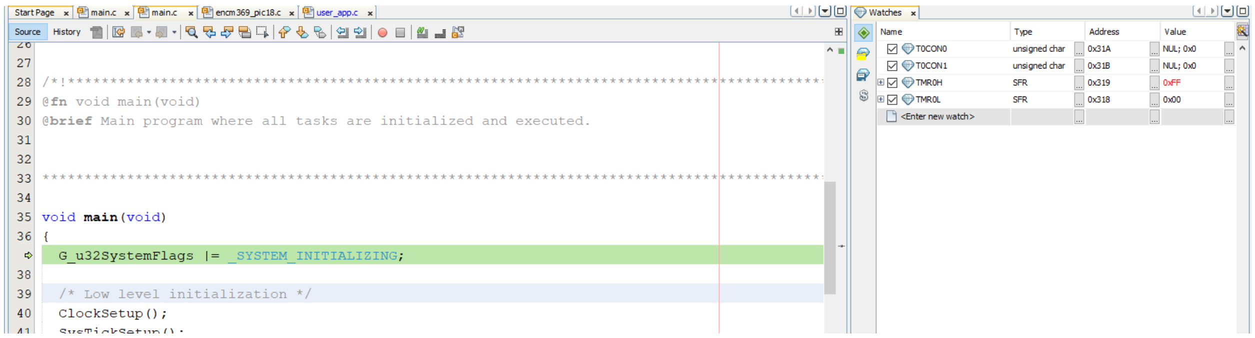
If you didn't write your code this way, update it now and make sure it still works.

Since the UserApp is the one that is using Timer0, write all the timer code there. Start by adding two lines of code to UserAppInitialize() to load the two timer control registers with the values you identified above. Try adding some comments to help guide the reader as to what you're up to.



```
void UserAppInitialize(void)
{
    /* LED initialization */
    LATA  = 0x80;

    /* Timer0 control register initialization to turn timer on, asynch mode, 16 bit
     * Fosc/4, 1:x prescaler, 1:1 postscaler  */


} /* end UserAppInitialize() */
```

Build your code and start the debugger. Setup a Watches window and add the four Timer0 registers so you can easily see them during debugging.

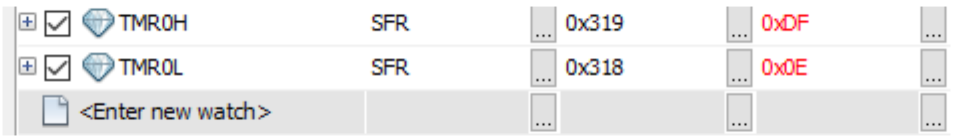


```
/*!**************************************************************************************
@fn void main(void)
@brief Main program where all tasks are initialized and executed.


****************************************************************************************

void main(void)
{
  G_u32SystemFlags |= _SYSTEM_INITIALIZING;

  /* Low level initialization */
  ClockSetup();
  SysTickSetup();
```

Name	Type	Address	Value
T0CON0	unsigned char	0x31A	NUL; 0x0
T0CON1	unsigned char	0x31B	NUL; 0x0
TMR0H	SFR	0x319	0xFF
TMR0L	SFR	0x318	0x00
<Enter new watch>			

Set a breakpoint in UserAppInitialize() just before T0CON0 is written. Run to the breakpoint then STEP through the two lines of code to observe the values being written. This is REALLY important, because sometimes microcontroller peripherals will not let you write their registers until you do something else. So when first working with a new peripheral, it's critical to make sure the processor is doing what you think you are telling it to do.

Now let the code run and pause/run a few times to make sure the TMR0x counter values are indeed changing which shows that the timer is running.



TMR0H	SFR	0x319	0xDF
TMR0L	SFR	0x318	0x0E
<Enter new watch>			

TimeXMicroseconds()

Now that you have a working timer, design and write the function that will set the timer to time out the requested period. The function should take the number of microseconds that the user wants to time as an input parameter. Make sure the parameter is the correct type and exact size you need.

The function itself must do the following in this order:

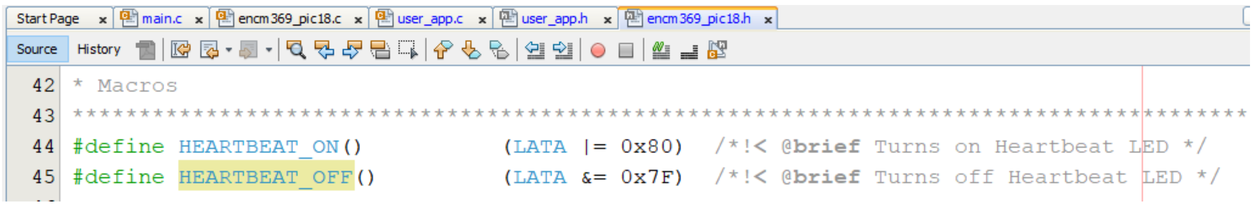
1. Stop the timer. Hint: what bit must you clear to do this? One line of code.
2. Pre-set the TMR0H and TMR0L registers to the correct value. Hint: this is NOT just the function's input parameter, right? Another hint: TMR0x registers are 8-bits wide. How do you split up a 16-bit number into a low and high byte which you can then write to low and high 8-bit counter registers? Two lines of code.
3. Clear the TMR0IF bit (flag) that lives in the PIR3 register. Hint: Make sure you ONLY clear the TMR0IF bit without touching the other bits. One line of code.
4. Start the timer. Hint: if you need a hint here, you're in trouble. One line of code.

Here is the header for the function. Make sure you have this in your code. The steps you need to implement are commented, so make sure you write the code for each step below the comment.

```
/*----------------------------------------------------------------------
void TimeXus(INPUT_PARAMETER_)
Sets Timer0 to count u16Microseconds_

Requires:
- Timer0 configured such that each timer tick is 1 microsecond
- INPUT_PARAMETER_ is the value in microseconds to time from 1 to 65535

Promises:
- Pre-loads TMR0H:L to clock out desired period
- TMR0IF cleared
- Timer0 enabled
*/

void TimeXus(INPUT_PARAMETER_)
{
  /* OPTIONAL: range check and handle edge cases */

  /* Disable the timer during config */

  /* Preload TMR0H and TMR0L based on u16TimeXus */

  /* Clear TMR0IF and enable Timer 0 */

} /* end TimeXus () */
```

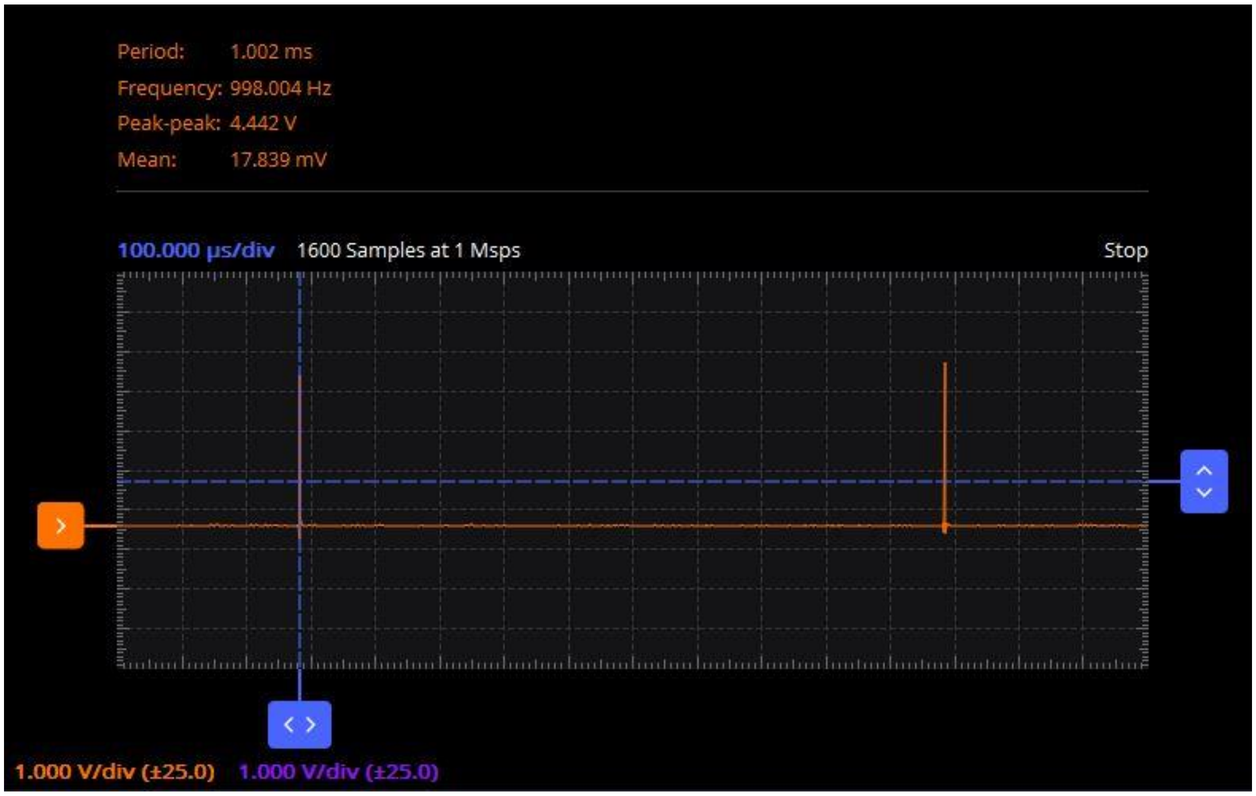
To test your function, add a call to TimeXus() to give a 1ms delay in main.c right after the SystemSleep(); function call. We want the processor to wait here until the 1ms is up, so write one more line of code to wait for TMR0IF to be set.

Lastly, update the macro functions in encm369_pic18.h to toggle the RA7 LED:



```
* Macros
**********************************************************************************************
#define HEARTBEAT_ON()          (LATA |= 0x80)  /*!< @brief Turns on Heartbeat LED */
#define HEARTBEAT_OFF()         (LATA &= 0x7F)  /*!< @brief Turns off Heartbeat LED */
```

Run your new code and use Scopy on RA7 to confirm that you have approximately 1ms between toggles of RA7. This assumes you have removed the delay code in UserAppRun() like you were supposed to. You should get exactly the Scopy output as is shown below.



In the embedded world, we would call this a "heartbeat" or "system tick" signal as it shows that the system is regularly executing at a defined rate. Notice the "ON" (high) time of the heartbeat LED is tiny compared to the 1ms delay time. This is very typical of an embedded system operation where the processor would spend the majority of its time sleeping / doing nothing. We'll get to sleeping the processor later.

Program Design

Take a moment to really, really understand how your system is running. How often is the code in UserAppRun() being executed? You MUST understand that it runs only once every one millisecond (assuming TimeXus() provides the 1ms delay in main.c).

To prove to yourself that you understand this, write code in UserAppRun() to blink RA0 LED at 1Hz by using a static u16 variable that increments by 1 each time UserAppRun() is called. When it reaches 500, this counter should reset to 0 and you should toggle RA0. No other code is required. If you think you need to add a call to TimeXus() inside UserAppRun() you are wrong. So you are using the system timing to blink your LED.

Once this is working successfully, modify UserAppRun() to create an interesting pattern on your LEDs. The pattern MUST be stored in an array of u8 (au8Pattern would be a good name) and the update to the LEDs should occur by indexing the array to get the current LED state to mask in to LATA properly. None of this part of the activity is new to anyone who has taken a C programming course.

Start by capturing the design of your code with a flow chart, and a visual representation of the LED pattern you'll use along with digital values for each part of the pattern. For example, if you are lazy and simply light up each LED in sequence, your design would look like this:

LED Pattern

LEDs	Binary Value	Hex Value
⬜⬜⬜⬜⬜🟥	b'00000001'	0x01
⬜⬜⬜⬜🟥⬜	b'0000010'	0x02
⬜⬜⬜🟥⬜⬜	b'00000100'	0x04
⬜⬜🟥⬜⬜⬜	b'00001000'	0x08
⬜🟥⬜⬜⬜⬜	b'00010000'	0x10
🟥⬜⬜⬜⬜⬜	b'00100000'	0x20

On your flow chart, indicate somewhere what delay you are using so someone can compare your design to how your code actually runs.

What to Hand In / Review

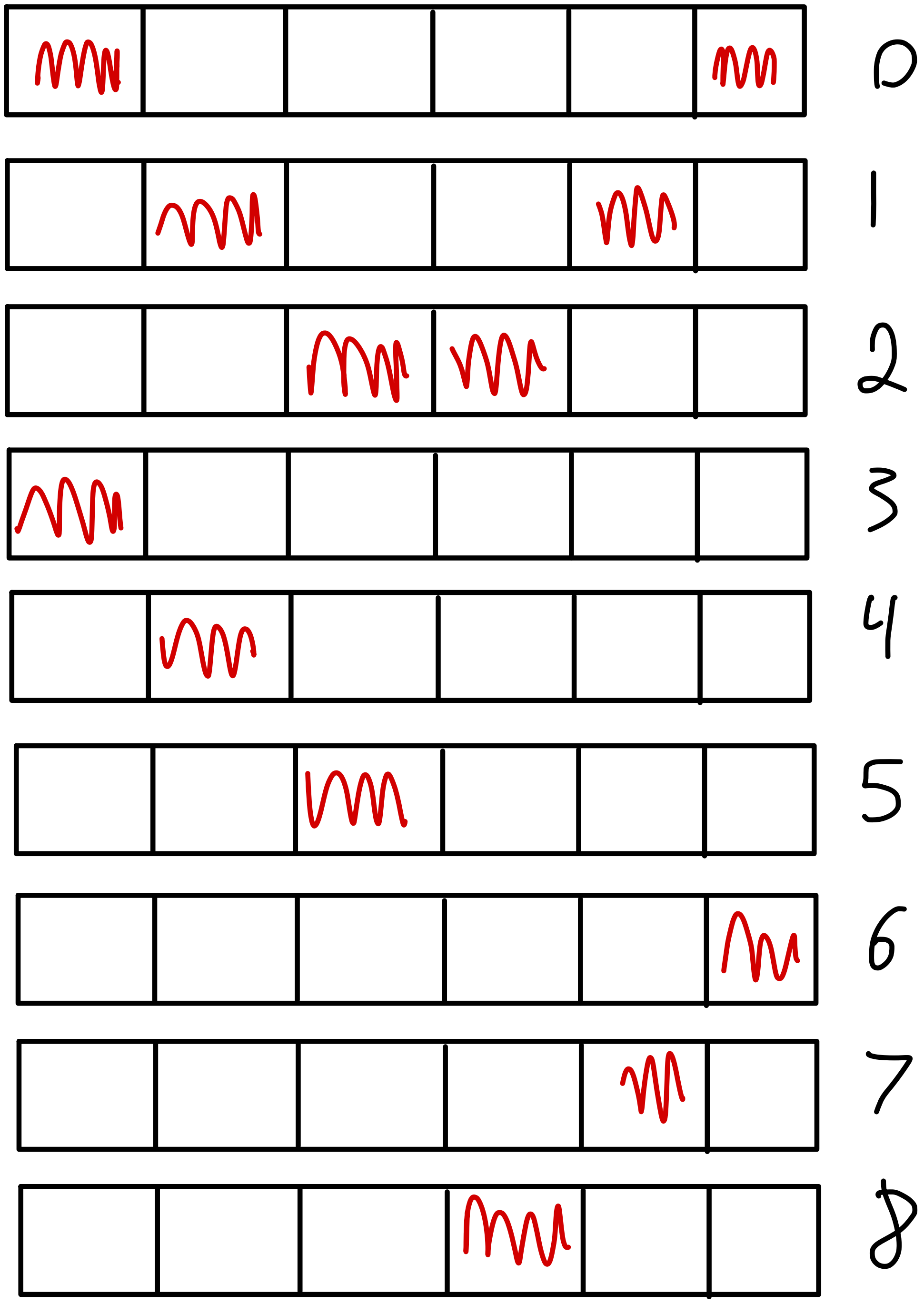
Save your design documentation in your source code folder and commit it and your code. Don't forget to push it to Github. Perform a code review on someone's code from your learning community who you have not already done a code review for. Upload the code review spreadsheet to the D2L Dropbox for the assignment.

More Things To try

OPTIONAL: this is not part of the assignment, but take a look and see if you can think about how you would do the following. Of course is you have extra time, take a stab at coding these things.

1. Use button input to change the speed.
2. Have two or more different patterns in your function. Use button input to change the pattern. Do this with a pointer to select the current pattern – the rest of the code would be identical regardless of the pattern being used.

Array Design Array Pos.



Array Position	Binary Value	Hex Value
0	0010 0001	0x21
1	0001 0010	0x12
2	0000 1100	0x0C
3	0010 0000	0x20
4	0001 0000	0x10
5	0000 1000	0x08
6	0000 0001	0x01
7	0000 0010	0x02
8	0000 0100	0x04

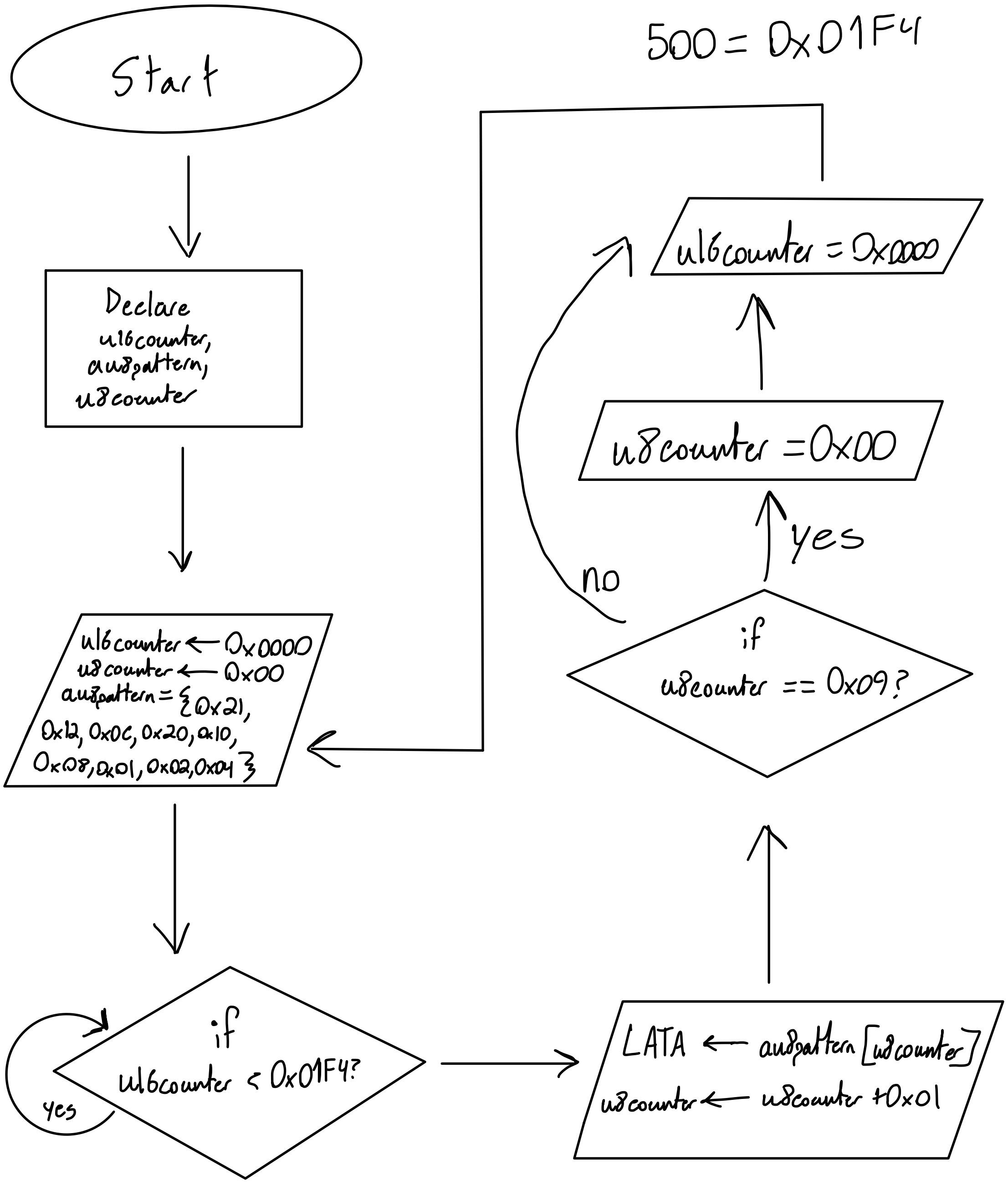
Array will look something like:

```
ua8Pattern[9] = {0x21, 0x12, 0x0C, 0x20, 0x10, 0x08, 0x01, 0x02, 0x04}
```

PIC Activity 5 Design
Flowchart: UserAppRun

Delay: 500 µs

500 = 0x01F4



Start
Declare u16counter, au8pattern, u8counter
u16counter ← 0x0000
u8counter ← 0x00
au8pattern = {0x21, 0x12, 0x0C, 0x20, 0x10, 0x08, 0x01, 0x02, 0x04}
if u16counter < 0x01F4?
yes
LATA ← au8pattern[u8counter]
u8counter ← u8counter + 0x01
if u8counter == 0x09?
yes
no
u8counter = 0x00
u16counter = 0x0000