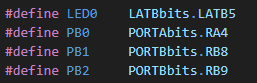
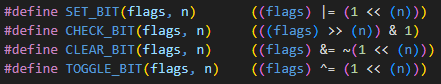
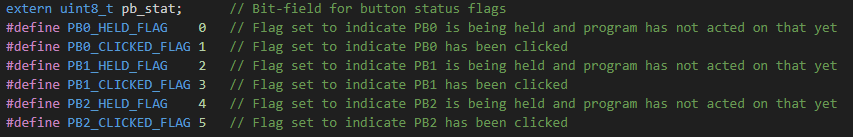
ENCM 511 Assignment 3

Preamble

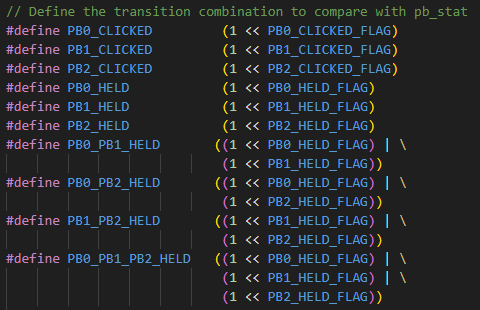
**common.h** contans the following which have been defined to be used throughout the program:

The LED and buttons have been abbreviated as shown.

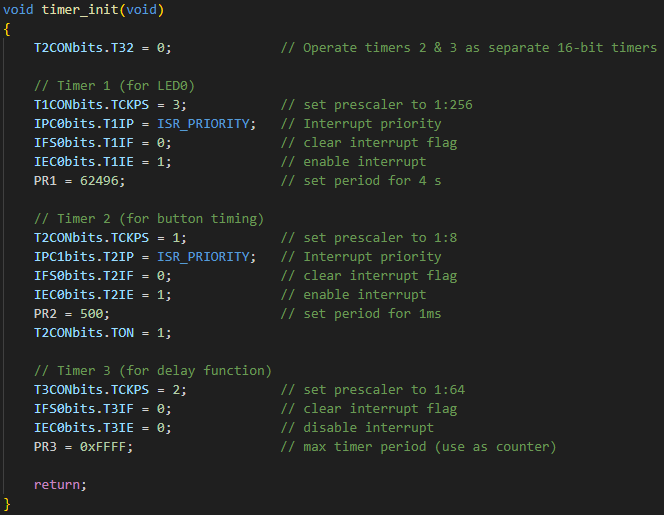
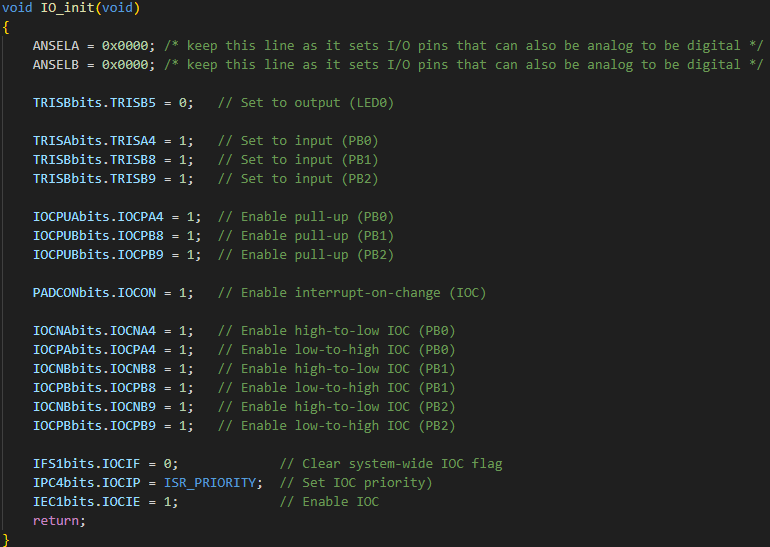
The following bit masks have been defined to allow for enhanced readablity. Several variables are used throughout the program to hold multiple flags (bit-fields). These defines allow those flgas to be accessed easily.

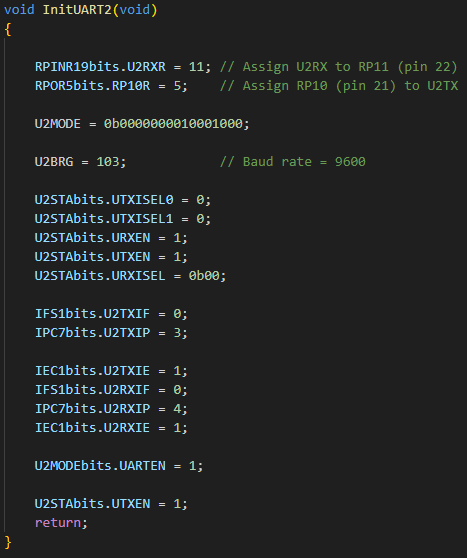
This is one such bit field. Purpose of each flag elucidated by its comment. Since each flag is either a 0 or 1, we considered it would be a waste to use a separate uint8\_t for each flag, as such we used one variable and each position in that variable has a specific button flag which its position is defined in the variable.

Here is an example of this bitfield being updated. In this case the PB0\_HELD\_FLAG of pb\_stat is set to 1.

The following have been created to simplify checking the flags in pb\_stat for commonly used combinations.

Peripherals

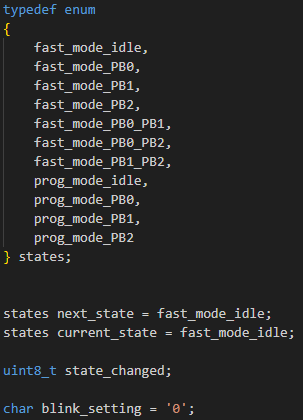
**init\_functions.c**  contains the initailization functions for the IO pins and the timers. Line functionality elucidated by comments. ISR\_PRIORITY is used to prevent ISRs from pre-empting each other.

**uart.c**  contains the initilization for UART2 which is taken from the provided lab files.

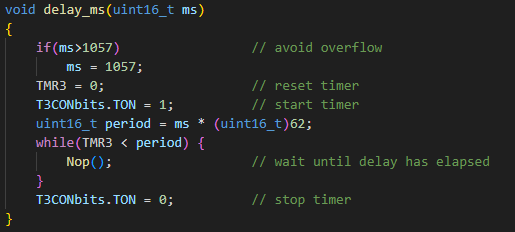
Datatypes and Initialization

The pb\_stat bit-field is initialized to 0.

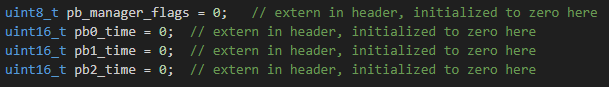
The states enum is used to control the state machine. Initial and next state are initialized to fast\_mode\_idle. The state\_changed variable is used to indicate when the machine has changed states. The blink setting controls the speed of LED blinking in certain states and is initialized to a char 0.



Key Functions

The delay\_ms function is held in the **delay.c** file and is used to for the debouncing of button inputs.

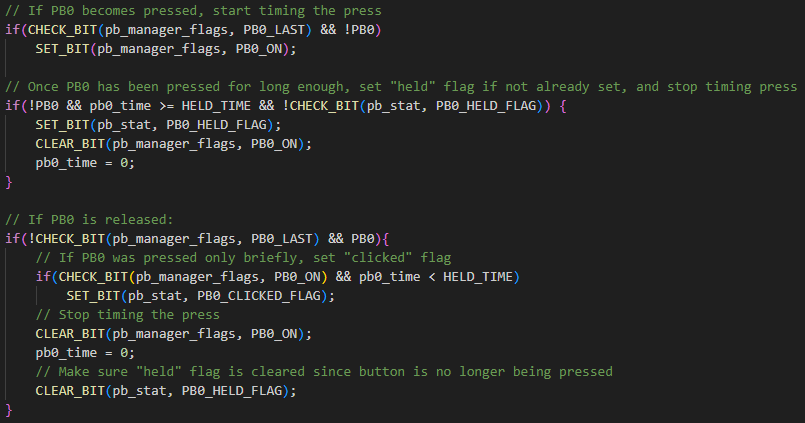
The **buttons.c** file contains the buttons\_update function. The function uses a bitfield pb\_manager\_flags as well as three unsigned 16 bit integers to record the number of milliseconds each button has been held for.

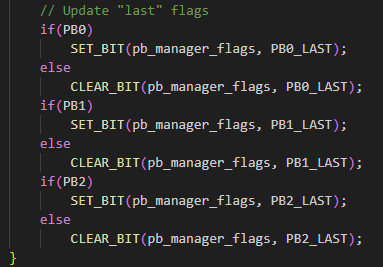


The buttons\_update function is called periodically (every 1ms) by main when the timer 2 ISR occurs or whenever the button ISR occurs.

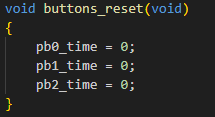
Initially it debounces the buttons.



Next the program evaluates the condition of PB0. This logic is then run for PB1 and PB2. **Note** that we choose to implement a hold to be a press that is longer than 1 second, if you stop holding it down the held flag is then reset to indicate that an active hold is not occurring.

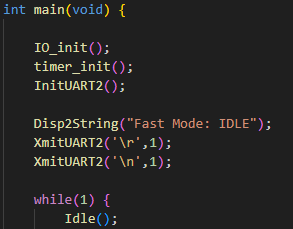
After all buttons have been checked their states are updated to be the new “last” flags.

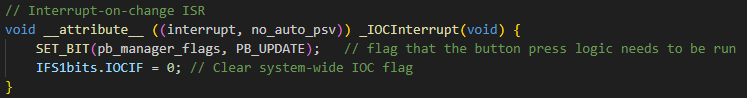
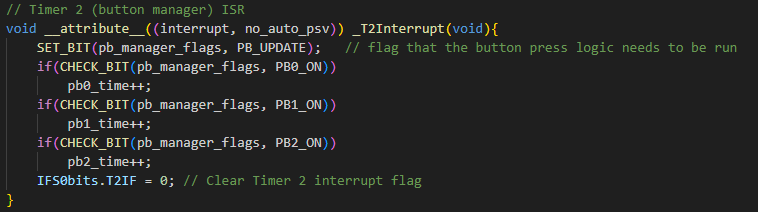
The buttons\_reset function is used to reset the millisecond counter of each button.

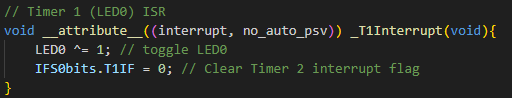


Operation

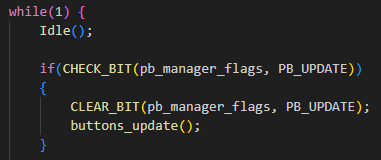
The code begins by calling all three initialization functions and printing the initial state to the terminal before entering its while loop and going into idle.



Assuming this is the first run of the program, i.e. the program is in a state where the LED blink timer is not running. The Idle will be interrupted by either a button press or the button manager ISR timer triggering.

If the program is in a state with the LED timer running there may also be an interrupt generated to toggle the LED.

If one of the ISRs set the PB\_UPDATE flag then main will call buttons\_update to check on the status of the buttons.



Next, the program checks to see if a button flag was raised by buttons\_update, if a flag is set, it will go through the state logic implemented as per the provided state machine drawing. A screenshot is provided to show the state transitions from idle mode. Once in a state which is not idle, the state transitions are simple as only one action can cause a state change back to idle.

A screen shot of a computer program

AI-generated content may be incorrect.

For the modes that are entered via a hold, holding any button can get it back to the idle state which is shown below.

A paper with writing on it

AI-generated content may be incorrect.A computer code on a black background

AI-generated content may be incorrect.

The state machine drawing is provided below.

The next section of code determines if a state transition occurs and raises a flag to switch the case which has been used to implement state functionality. This and the following section are not under the pb\_state flag and happens each time the microcontroller goes out of idle.

A black background with white text

AI-generated content may be incorrect.

If the above code has detected a change of state, it switches the case implemented in the state logic. We used switch statements and each section has its own PR1 value depending on the required blink rate needed. If TMR1>PR1, we reset TMR1 this is done to catch the case in which PR1 is changed from a larger value to a smaller value and TMR has surpassed the new PR1 value in which case the timer will go until it overflows before interrupting.

A screen shot of a computer program

AI-generated content may be incorrect.

The only case that can directly control the next state is prog\_mode\_PB2, as after you push PB2 to confirm your setting value, it should go into idle mode. We have implemented the button logic inside a while loop in RecvUartChar012 and thus once that function has exited, the program returns to prog\_mode\_idle.

A computer screen shot of a program code

AI-generated content may be incorrect.

At the end of the case statements, but before breaking from if(state\_changed), the pb\_state variable and the state\_changed variable are reset.  
A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.Below is attached a screenshot of RecvUartChar012 function. This reuses code from the regular button function found in buttons.c, this time only looking for a click to exit (PB2 click). Since the user has already clicked PB2 the next case should be set prog\_mode\_idle which is done in the case statement for prog\_mode\_pb2.

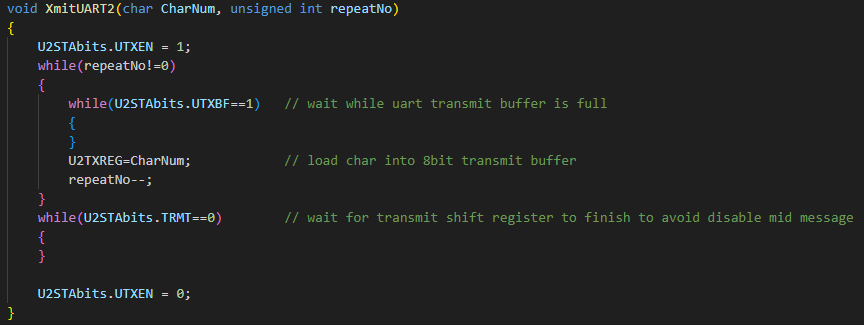
Answers to Lab Questions

*In part 1 (the lab portion), what was the target baud rates? What is the % error of the actual baud rates?*

The target baud rate was the UART standard 9600. The actual baud rate (with U2BRG = 103 and BRGH = 1) was:

The percent error is then:

*What are your comments/explanations for XmitUART2?*

**

*In your assignment part, how do you get the LED to blink while waiting for a character to be input? Is the character receive function a “busy wait,” i.e., does it stop/block the CPU from doing something else?*

The character receive function has a while that is constantly polling if an enter key has been received, inside that while loop there is function that only attempts to read a character when the data in the Rx buffer interrupts occurs that there is, otherwise it is not doing anything. Therefore, this is considered a busy wait as its constantly polling if the received char is an enter and is blocking other sections of code from occurring.