SCC.369 Assessment 2: Clock

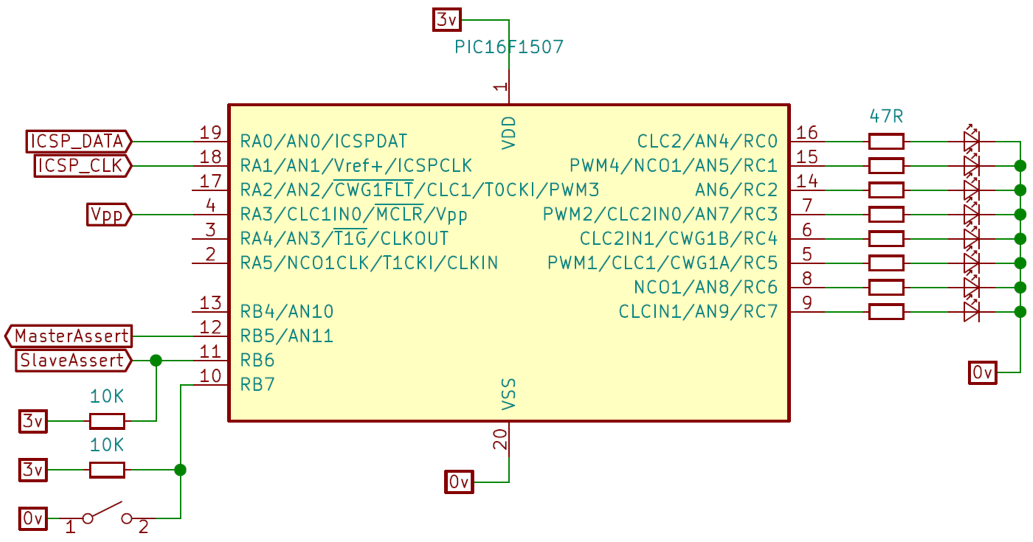
The Task

This exercise builds on the design for assessment 1, Lights and Switches, so you should need few hardware changes, but it is important that you begin with working hardware so please ensure this is the case, even if your code didn't function perfectly.

At the end of this exercise you and a friend should be able to demonstrate a working 24 hour binary clock running 60 times normal speed, i.e. a clock that updates the minute count once per second.

Basic Setup

In assessment 1, you had a set of LEDs on PORTC and switch inputs on PORTB bits 4, 5, and 6. For this exercise, you should have a switch on RB7/ bit 7, and the other switches should be removed. In terms of configuration for PORTB, you will need to configure bits 6 and 7 as inputs and bit 5 as an output. As with the remaining switch on bit 7 (pin 10), bit 6 (pin 11) will need a 10K pull-up resistor connecting it to +3v. You should end up with the following; the purpose of MasterAssert and SlaveAssert will be explained later, but you should ensure you leave easy access for an additional wire.



The following sections give more detail on the task, some background to read alongside the datasheet, an outline of the marking criteria. You can use the Counter.asm example as a starting point.

Have fun!!!

Exercise

As with the first task, there are a number of stages to implement and demonstrate in the marking session. Some background to help you get going with each task is provided later, and this should be read in conjunction with the relevant sections of the datasheet.

Task 1: Cycle counting – achieving more accurate timing

When looking at the datasheet you may already have noticed each instruction is listed with a cycle count; perhaps you've already made use of these, in which case you'll be a good way to completing this first task. Show a repeating 8 bit (0..255) counter on the PORTC LEDs using cycle counting to achieve an accurate one second update. **You will be expected to show your working/ calculations.**

Task 2: Just a Minute

Adapt your solution to task 1 so that the counter resets to zero every sixty seconds. Ensure that you adjust your cycle counts to include any additional instructions. **Again, you will need to show your working/ calculations.**

Task 3: Counting Button Presses

This task introduces interrupt handling, in this case using the Interrupt On Change (IOC) feature to update the counter. You will need to demonstrate that your counter increments (via IOC) every time the button on RB7 is pressed.

Task 4: Hardware Counters

Counters, often (as with the PIC) called timers, count events, whether they be clock pulses, interrupts, etc. The current count value can typically be read by running code, or counter-timers can be configured to generate an interrupt after a set number of events have occurred. For this task, you need to set up an interrupt handler so that the LEDs display an incrementing count every second.

Task 5: A Binary Clock

For the final task you will need to combine the code for tasks 3 and 4 to create a binary clock. You will also need to adapt your IOC code to support MasterAssert and SlaveAssert.

When your code starts it should check whether SlaveAssert has been pulled low. If SlaveAssert is low your code should operate in IOC mode – as task 3, whereas if it remains high, your code should set MasterAssert low and operate in counter mode – as per task 4.

The final bit of coding is to have the IOC code reset when it reaches 24 or 0x18, and the counter code reset once it reaches 60 or 0x3c. So a master would count 0..59, and a slave 0..23. Every time the master resets it should pulse MasterAssert high, then low.

For fun, you can test this with a friend… take two MicroBit-PIC systems connected to the same PC, add a wire between the GND/ 0v on both, then put a wire from MasterAssert on your master to SlaveAssert on your slave. You should have a working binary clock with one board counting seconds and the other minutes. You'll need to demonstrate that your board can correctly operate as either a master or a slave – you can manually manipulate the Assert lines on a single device to do this.

Note: you will need to think how to get the boards started as the slave will need to see the assert line pulled low when it does the check. You can be careful how you power up the boards, but some extra credit if you can demonstrate software handling this in a robust way.

Finally, you should include some means to speed up the count, while the system is running, for debugging and marking.

Background

The following provides some background and pointers to useful starting points in the datasheet, which may refer you to other sections to read. Remember that datasheets should be precise… they tell you what must be done to get the component, in this case a microcontroller, to behave in certain ways; if you miss steps, or incorrectly configure the device it will not work as expected and this can be quite hard to debug, so care and attention to detail is essential and saves a lot of time.

Cycle Counting

You should recall from past modules that each processor instruction takes a number of clock cycles to operate. In other words, for every *n* on..off, pulses of the clock signal, one instruction will be fetched, decoded, and executed. To make this a little more tricky, some instructions will likely take longer than others, and may even have different durations depending on their parameters or processor state. In complex systems, components such as cache, pipeline, and Memory Management Unit (MMU) state can also be a factor. Thankfully, the PIC is fairly simple with each instruction taking one or two instruction cycles, and few special cases.

The number of instruction cycles each instruction takes is given along with the description of each instruction, and associated summary table, in section 24 toward the end of the datasheet.

You should now be asking, how long is an instruction cycle… On the PIC this is one quarter of the clock frequency, Fosc/4, or in other words it takes four clock pulses to process a basic instruction.

You'll see from the diagram on page 43 of the datasheet that the system clock can be generated from a number of different sources, however, in our case the important detail is on page 49, …the speed of the internal oscillator, set by the OSCCON register. If you change this setting while the processor is running you should check the OSCSTAT register to ensure the new clock frequency has stabilised before performing anything time critical.

With this information, you should be able to construct and pad code to take any desired time to execute given a set clock frequency.

Note that the datasheet gives power-up/ default settings for each configuration register – it is worth checking this to avoid lots of unnecessary initialisation code.

Interrupts

Interrupts allow us to handle 'random' or aperiodic events without constant checking or polling for state changes. They are a very efficient way of responding to irregular, relatively low-frequency, input; however, they move us from our familiar linear, step-by-step, code execution to an event model, where different blocks of code are executed in response to external stimuli. In a busy system, the seemingly random jumps between different blocks of code can be confusing and difficult to track, but in this case, things are relatively easy as you only have to handle a small number of interrupt sources.

Initially we are interested in the ability to respond to changes on PB6 or PB7, and for this we need to look at section 12 of the datasheet beginning on page 108.

Notice that IOC detection must be explicitly enabled for each pin we're interested in (IOCBP), and the IOC feature must be enabled (INTCON) before any interrupts will be generated.

Once a change has been detected the change detect flag associated with the pin (IOCBF) is set so your interrupt routing can determine which pin or pins have changed. These flags must be cleared in your code or you risk getting repeated interrupts for the same event.

Finally, the whole interrupt system must be enabled (INTCON) before the microcontroller will respond to any interrupts. The system disables this on entry to an interrupt service routine (ISR), and re-enables interrupts on exit from the ISR, so on the PIC you can’t normally get nested ISR calls.

Timers and Counters

Timer 1 on the PIC, described in section 17 of the datasheet, is a 16bit counter that can be driven from multiple sources, with a divide by *n* pre-scaler to reduce the update rate (T1CON).

The counter can be loaded with any desired 16bit value, held as a low-byte (TMR1L), high-byte (TMR1H) pair, and will generate an interrupt when the count rolls up to zero – this allows plenty of scope for fine tuning.

As with the IOC feature, Timer 1 interrupts must be enabled (PIE1) and the Timer 1 flag (PIR1) reset once you’ve handled the event in your ISR.

Further, Timer 1 is one of the Peripheral Interrupts, and this whole set (INTCON), and again global interrupt handling must be enabled for these interrupts to trigger your ISR.

Marking

Task 1: Cycle counting – achieving more accurate timing

Marks available for maintaining accurate 1s update, identification of system clock and instruction frequency, cycle counting, consistent timing in event of loops and conditionals.

Task 2: Just a Minute

As per task 1, plus correctly resetting once per minute.

Task 3: Counting Button Presses

Correct configuration of IOC, repeated interrupts… once per button press, correct updating and display of counter.

Task 4: Hardware Counters

Correct configuration of Timer 1, repeated interrupts maintaining accurate 1s update.

Task 5: A Binary Clock

Correct assertion of MasterAssert and response to SlaveAssert. Pulsing MasterAssert once per minute when in Master mode. Resetting counter at correct point depending on Master/ Slave operation – due to time constraints, we will use the button help confirm this. Correct operation in both Master and Slave mode. Workable software solution to ensuring correct slave start up when turning on both devices.

In Addition

For each of the above, marks will also be available for understanding and ability to answer marker's questions, code layout, commenting/ documentation – appropriate number and detail of line comments, block comments, and routine header comments.

Normal submission rules and penalties apply.

-- acs 23/10/2019, updated 22/10/2021