Lab 3: Button Controlled Binary LED Counter

EE 234: Section 2

Michael Mentele

Partner: Jonathan Quissenberry

September 18, 2012

*Abstract*

This paper describes the programming of a PIC32MXcK microcontroller using C++ as a counter using the Cerebot on-board LEDs. We use PB1 and PB2 as inputs and the display of the four on-board LEDs as the outputs. When you press PB1 or PB2 you either count up from 0 or down from 0 respectively. However the viable number values for this system cannot be negative so display values loop from 0 to 7 or 7 to 0 depending on the direction. So in our system when the LEDs count up in binary and reaches seven the display becomes zero or off. Similarly when the count is going down and reaches zero it then goes to seven on the next display cycle. You can press these push buttons at any time to change the count direction. In addition when you press both pushbuttons the system freezes or pauses. If you press either pushbutton you then resume counting however if you push both again then it resets the counter and waits for a pushbutton to be pressed to begin counting from zero again.

**Introduction**

In this lab we will use read/write operations to read the state of buttons on Port A to perform different operations for a 4-LED display counting in binary. We initiate the system clock and peripheral clock to gain a basic understanding of how the clock works.

There are four essential cases that would affect the count on the LED display.

1. You press and release PB1 (Count Up)
2. You press and release PB2 (Count Down)
3. You press and release PB1 & PB2 (Stop)
4. Following the first press and release of PB1 & PB2 (Reset)

**Hardware**

Simply connect the USB cable to your PC and to the debug port on the Cerebot board. You will be using push button 1, push button 2 and the four on-board LEDs.

**Software**

**Design**

Because you are using a button as the input you are waiting for a de-assertion to be a triggering event. Implicitly this means you must remember what was asserted and then act *after* it de-asserts. To continually count without waiting in a state for an event (thus having indeterminate state lengths) variables were used to remember what was pressed prior to an event. These variables are stored as the program continually cycles; checking the button states in a loop that takes a second to time out to the counting function. After 1 second the program counts based on what was the last bit asserted in the check states section. This design scheme allows the system to go “up” or “down” from its current state.

This was accomplished by explicitly acting when all the buttons are de-asserted and saving the states of the button(s) that had been asserted to use in IF states when you check for an event. You can see the change in design philosophy by comparing the two flowcharts. One is waiting for assertions whereas the other is checking if they have and storing variables for the next check.

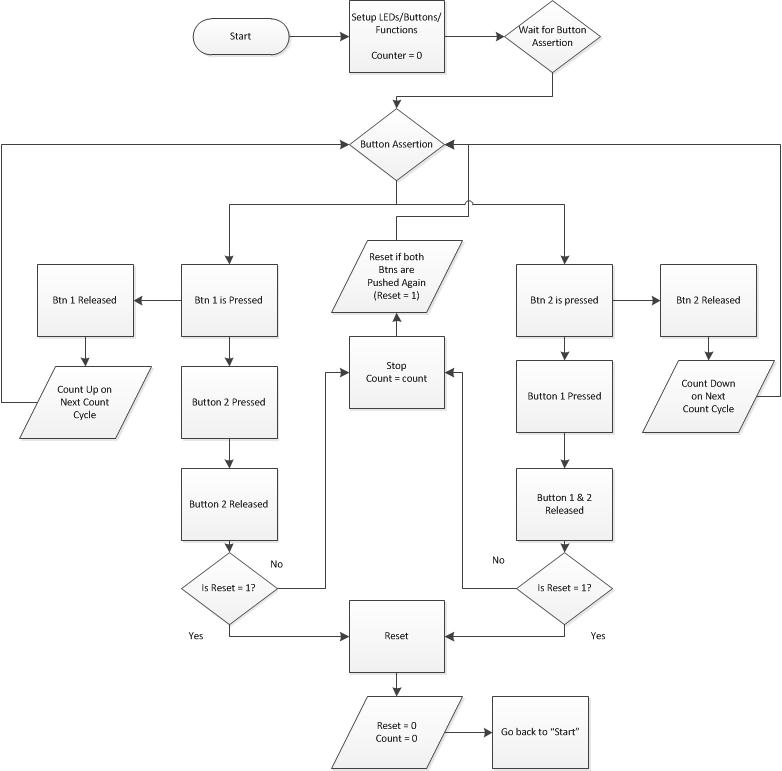


Fig. 1 Initial Design Flow Chart

**Initial Design:**

The problem with the above approach is that the system gets stuck waiting for things to happen instead of performing the operations at predetermined times. In this system you do operations based on events whereas in my redesign you do operations based on time. To make this system work I would need to pass some type of time variable around into loop and have IF statements all over the place to break at certain points in time. It is much simpler to use memory.

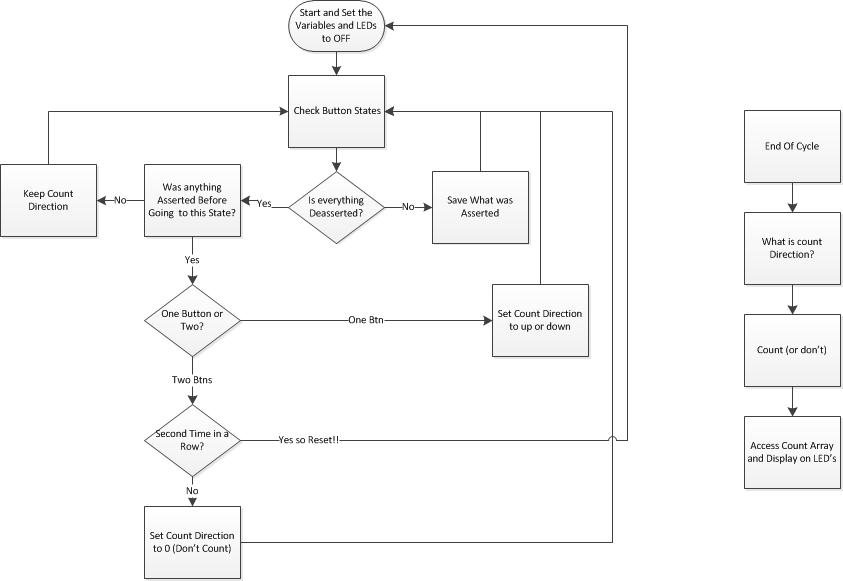


Fig. 2 Redesign

**Redesign:**

This solution is simpler and cleaner and outputs at regular time intervals which is what we wanted. The use of variable to store which buttons were pushed and whether they were pushed or de-asserted together make it possible to perform the correct operation with only one timing loop.

**Test Procedure and Results**

**Methodology**

Testing was performed from a “Top Down” viewpoint where functionality is tested first followed by a debugging phase before finally being tested by an independent user. We also made great use of the step through function in the iterative design project. This greatly reduced the time it took to debug the program before getting a testable prototype.

**Procedure**

1. Test Correct Operation
   1. Test each possible case
      1. Does it count up and down?
      2. Does it stop?
      3. Does it reset?
      4. Do the operations execute in the correct sequence and after the corresponding buttons are pressed?
2. Attempt to Break Current Build
   1. Counting Up
      1. Count Up from Stop
         1. Does it continuously count up after PB1 is pressed?
            1. Hold down PB1. Does anything change?
            2. Assert PB1 several times.
      2. Count Up after Counting down
         1. Does it wait until after the current cycle to count up?
         2. Hold down PB1. Does anything change?
      3. Count Up from Reset
         1. Does it begin counting?
         2. Hold down PB1. Does anything change?
   2. Counting Down
      1. Repeat Similar sequence as for Counting Up
   3. Stopping & Resetting
      1. Hold one button down for a few seconds and then assert the other.
      2. Do the same thing for de-asserting the buttons.
3. Let someone else play it!

Using Murphy ’s Law… and another perspective this is an effective way to achieve unexpected results.

**Results**

When starting the board it waited with the LEDs off until a button was pressed. When button one was pressed it counted up in binary across the LEDs starting on the right. So from the right it counted up like so:

000 -> 001 -> 010 -> 011 -> 100 -> 101 -> 110 -> 111 -> 000

0 1 2 3 4 5 6 7 0

When you click button two from the start state it counts down in the opposite manner from 7 to 0 and back to 7.

When an opposing button is clicked when counting up or down on the next second interval it will count the opposite direction. So you press button one between states while counting down then when it counts next it will count up. Meaning if in the ‘0’ state and you press PB1 while counting down it will go to the ‘1’ state when it counts.

Pressing both buttons freezes the count as expected and pressing them both sets all the LEDs to off. The program meets the design criteria for the lab.

**Notes**

One unexpected timing issue was noticed during phase three of the testing where the LED display would refresh slightly faster. The issue occurred when you held buttons down because the program had to do fewer checks and therefore the loop skips over some if statements thus causing the code to eat less clock time per loop iteration.

One way to reduce this effect is have every IF statement be explicit so as to be not contained within another IF statement. However the most effective method would be to synchronize in some way with the system clock on the Cerebot board.

Something to think about is when holding a button down and asserting and de-asserting the other. This is interpreted as the same as clicking both but with an unspecified time for de-assertion between the buttons. You could add some further functionality for delays here.

**Answers to Questions**

1. **Disassembly Listing**
   1. addiu (v0,zero,1)

The v0 register is a return register and it gets either 0 or 1.

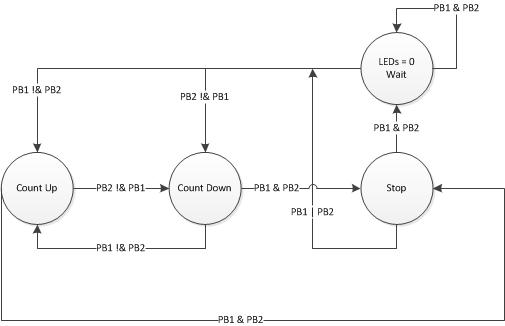
* 1. sw (s8,88(sp))

The s8 register is a save register that is preserved across function calls. It gets the value at the address the stack pointer is pointing to.

* 1. sw (ra,92(sp))

The ra register is the return address register. It gets the value of the that the pointer points to.

1. **Revised State Chart**

****

**Conclusion**

In this lab we realized the importance of timing in every application. But to juxtapose the prior lab we are no longer waiting in a state for an input to drive you to the next. Instead time drives you to the next state. This gives us a new perspective on approaching code design for differing applications.

The results were achieved as expected with a slight exception; when a button is held down the system counts just slightly faster on the LED display. This is because there are a few less checks or sets of instructions that have to be read through on every iteration. However the impact is mild and still meets performance expectations.

The next step would be to identify a method to time regularly. This would involve the exploitation of the onboard clock for synchronization purposes.