Michael Eller

ECE 3430

Lab 7

19 October 2015

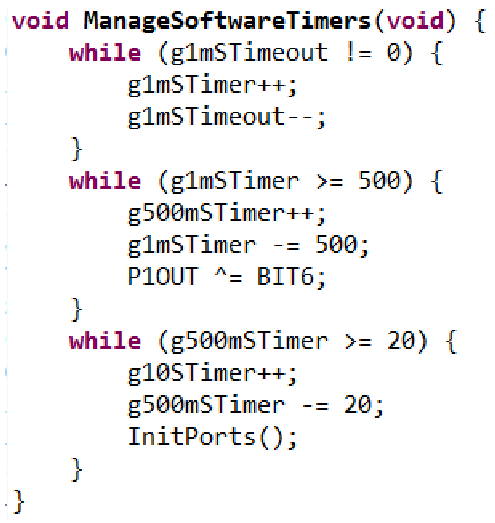
Switch Debouncing

Goals:

The purpose of this laboratory was to debounce the button onboard the MSP430 using a state machine and a software implemented timer. Using the timer, the green LED was toggled every 500 ms, the red LED was toggled every time the button was pressed, and the port directions were reinitialized every 10 seconds.

Program Description:

Software Timers



```
void ManageSoftwareTimers(void) {
    while (g1mSTimeout != 0) {
        g1mSTimer++;
        g1mSTimeout--;
    }
    while (g1mSTimer >= 500) {
        g500mSTimer++;
        g1mSTimer -= 500;
        P1OUT ^= BIT6;
    }
    while (g500mSTimer >= 20) {
        g10STimer++;
        g500mSTimer -= 20;
        InitPorts();
    }
}
```

Figure 1, software timer code

All operations in this lab worked off of a global program timer. This timer was built off of the Timer A interrupt service routine, which simply incremented a variable, g1mSTimeout, every 1 ms. The code above increments the main software generated timer, g1mSTimer, for every increment of g1mSTimout. Ideally g1mSTimeout will either be 0 or 1 for the duration of the program. A 500 ms timer was incremented and the green LED was toggled when g1mSTimer reached 500. The 1 ms timer was then decremented by 500 to keep it in a reasonable range. Similarly, a 10 second timer was incremented and all port directions were reset every time the 500 ms timer reached 20. This timer implementation was more efficient because it allowed for a very short timer interrupt routine.

Debouncer

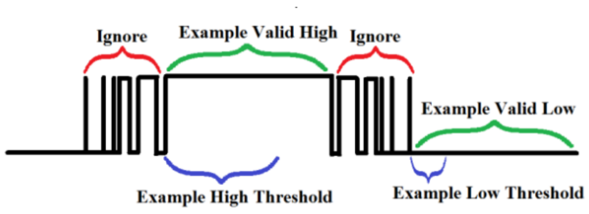
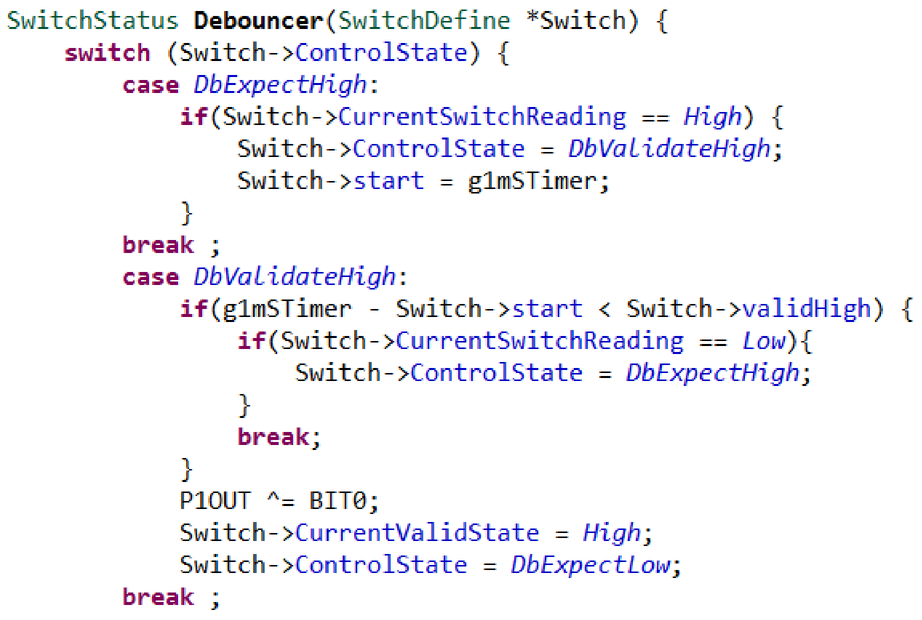


Figure 2, debouncing concept

Debouncing was necessary to normalize button/switch behavior. For example when the button on the MSP430 was pressed, the signal was far from perfect. An example of the unpredictability of the button is depicted above. Using the Debouncer method in this project, these false High's/Low's were effectively ignored, which created a much more stable button interrupt.



```
SwitchStatus Debouncer(SwitchDefine *Switch) {
    switch (Switch->ControlState) {
        case DbExpectHigh:
            if(Switch->CurrentSwitchReading == High) {
                Switch->ControlState = DbValidateHigh;
                Switch->start = g1mSTimer;
            }
        break ;
        case DbValidateHigh:
            if(g1mSTimer - Switch->start < Switch->validHigh) {
                if(Switch->CurrentSwitchReading == Low){
                    Switch->ControlState = DbExpectHigh;
                }
                break;
            }
            P1OUT ^= BIT0;
            Switch->CurrentValidState = High;
            Switch->ControlState = DbExpectLow;
        break ;
```

Figure 3, first two states in Debouncer state machine

The first two states of the state machine implemented for this lab is shown above. The last two states are completely analogous to the first except they expect and validate a logic Low.

Debouncer was continually called in the main ‘while(1)’ loop with another function that read the current switch or button input. The state machine remained in either ‘DbExpectHigh’ or ‘DbExpectLow’ until that switch reading was High or Low respectively. Once it received an High or Low, it moved to a validate state, which used the global 1 ms timer to determine if the High or Low input remained constant for a given time interval specific to each switch. Once a valid High was determined, the red LED was toggled.

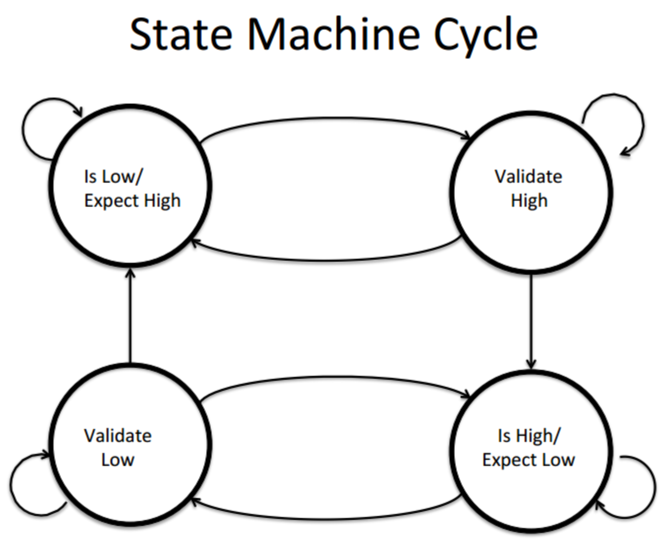
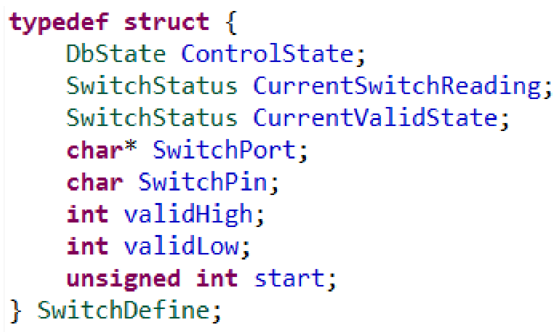


Figure 4, state machine flow chart

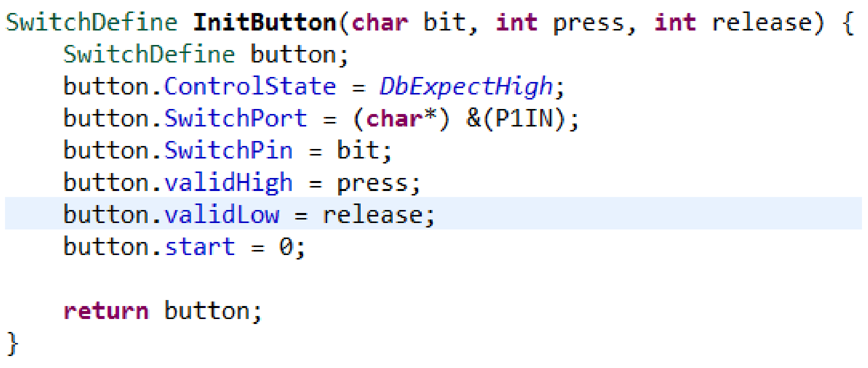
Generalizability

A large component of this lab was focused on creating generalizable code, meaning the ‘Debouncer’ function could be used with multiple switches easily. Several steps were taken to accomplish this. The ‘SwitchDefine’ struct, which was served as a button object, was modified to include many variables required to complete the ‘Debouncer’ function but keep those variables specific to one button. The ‘validHigh’ (time interval for a valid High), ‘validLow’ (time interval



```
typedef struct {
    DbState ControlState;
    SwitchStatus CurrentSwitchReading;
    SwitchStatus CurrentValidState;
    char* SwitchPort;
    char SwitchPin;
    int validHigh;
    int validLow;
    unsigned int start;
} SwitchDefine;
```

required for a valid Low), ‘start’ time, and ‘SwitchPin’ were all added to make it easy to create multiple buttons on different pins/ports and still be able to debounce them simultaneously.



```
SwitchDefine InitButton(char bit, int press, int release) {
    SwitchDefine button;
    button.ControlState = DbExpectHigh;
    button.SwitchPort = (char*) &(P1IN);
    button.SwitchPin = bit;
    button.validHigh = press;
    button.validLow = release;
    button.start = 0;

    return button;
}
```

Another function was created to further generalize the code concerning button/switch objects. Instead of initializing each parameter of each button individually, this function made it simple to create different buttons. The ‘SwitchPin’, ‘validHigh’ and ‘validLow’ were all set to parameters passed to the function. The final ‘Debouncer’ method did not include any program specific variables except the global timer.