Software button debouncing

Student: Nira Tubert
Mentor: Herbert Pötzl
apertus° Association

Google Summer of Code 2019

Project: https://github.com/niratubertc/GSoC

1 Introduction

The main goal of this project is to design a debouncing software based system for the apertus° AXIOM Remote, the remote control unit of the AXIOM Beta Camera system.

Figure 1: AXIOM Remote Concept 2019 - Note: Design is subject to change as improvements are made [1]

The AXIOM Remote contains several buttons around the screen for camera functions. Pressing buttons produces an effect called *bounce*. This effect produces an incorrect reading by the microprocessor, of whether the button is pressed or not. AXIOM Remote switches already contain a hardware based debouncer to avoid this effect. In order to know if this works and to ensure the proper working of the system, the software based debouncing is performed. The purpose of this system is to ensure the correct working of the buttons and the built-in hardware based debouncing system. The software run over two PIC16F microcontrollers which manage the status of the buttons. These two microcontrollers communicate with the PIC32 microprocessor to report the status of the button, pressed or not. To improve the behaviour of the buttons, I have been provided of a program which establishes some communication between PIC32 and PIC16, in order to be able to send some pulses from the PIC32 so we can perform some bouncing and this is sent to the PIC16 pin, and to receive its output. In this way we can check the operation of the debouncing software in the PIC16.

2 Getting started

2.1 What is bouncing/debouncing?

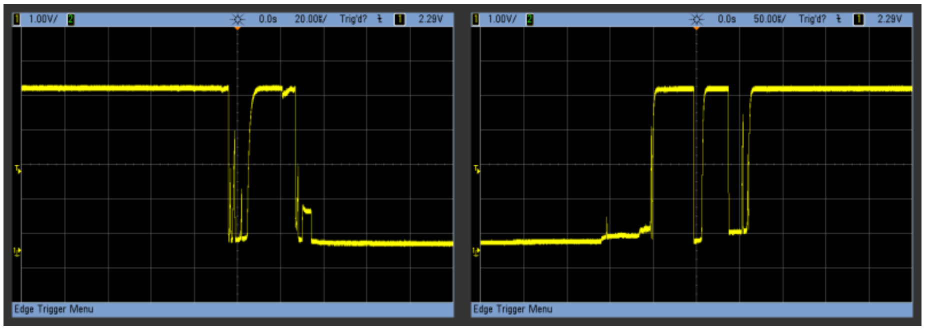


Figure 2: Debounce bouncing [2]

In the AXIOM Remote we have several peripherals: an LCD screen, buttons, switches etc. When we press a button it changes the state to pressed and once we stop pressing it changes again to released. From our point of view we have performed a single pulse, going from low to high and finally low again. From the point of view of the microncontroller, there have been many transitions between low and high that it can interpret as the button has been pressed several times in a very short time. This is because the contact between the drivers is not perfect, as explained in this article:

> "Due to the mass of the moving contact and any elasticity inherent in the mechanism and/or contact materials, contacts will "bounce" upon closure for a period of milliseconds before coming to a full rest and providing unbroken contact". [5].

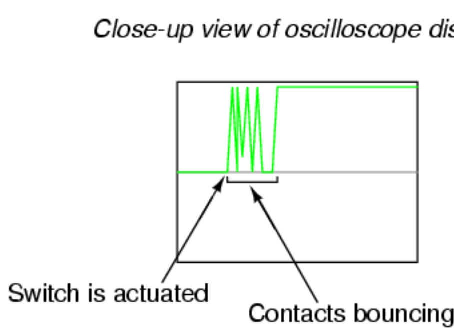


Figure 3: Bounce in a switch when pressed [5]

As we can see in Figure 3, there are more transitions than we expect. We want, when pressing the button once, the microprocessor to interpret that we have pressed only once, therefore, ignoring undesired transitions. To solve this problem there are systems based on hardware and software. This is debouncing, agree with the microprocessor to understand when we are pressing the button and when release. As I mentioned earlier, this problem is very common and the most of the buttons already have a built-in hardware debouncer. The buttons of the AXIOM REMOTE have it incorporated. This project will allow us to see if the debouncer works correctly and also to make a software based debouncing. The hardware based debouncer tries to avoid these transitions using components, such as an RC circuit. This circuit, having a capacitor, prevents rapid voltage variations, resulting in a single transition.

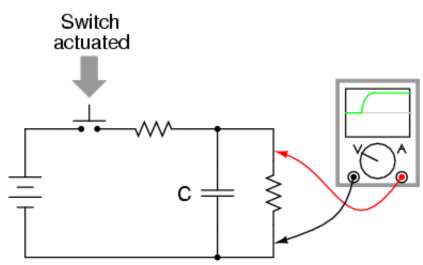


Figure 4: Debounce hardware based [5]

The software based debouncing consists in deciding when the microprocessor interprets the button has been pressed from timers. In the next section I explain in detail the solution proposal of this project.

2.2 Debouncing solution

The microprocessor detects the rising edge and the falling edge in a specific pin. We want to detect the first falling edge (button pressed) and wait for the signal to not bounce. For this behavior, the solution is as follows:

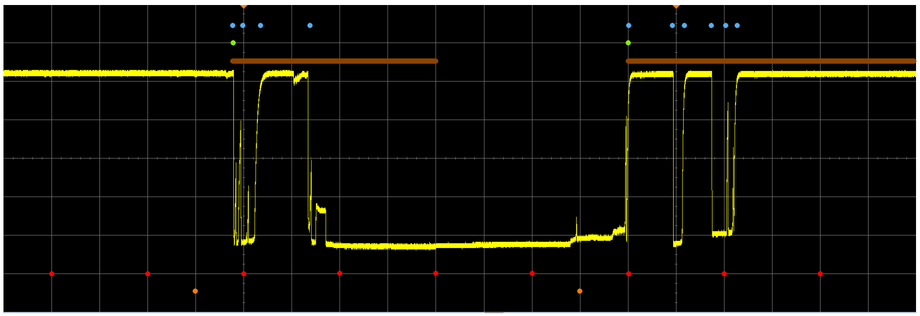


Figure 5: Bounce in a switch when pressed

The proposed solution is understood with the image above. In this image we can see the rising and falling voltage in a pin of the MCU. As we can see there is more than one transition, our purpose is it to be understood as HLH. To that we use interrupts and timers to decide when the value is taken and when not. The first green dot (on the left side of the graph, below the blue dots) represents the first interrupt that is generated by the Interrupt of Change. The red dots represent the timer interrupt, with each "cycle" detects if there is an IOC. In case of detecting an IOC, 2 cycles that are approximately 500us are passed. After this timer, we interpret a pressed button. In this way we give a bounce time in which we only detect the transitions (blue dots), but we do not decide that the button has been pressed. After this time, when you release the button, the first transition is detected again (green dot in the right side of the graph). Assuming the case in which this IOC happens right after the timer interrupt, this IOC will not be detected until the next timer interrupt. After that, as in the previous case, 2 timers are passed and we interpret that the button has released. This case is longer than the previous one, therefore we detect the change after a maximum of 750us.

2.3 Learning about PIC microcontroler

2.3.1 Registers

In all microcontrollers there is an area where data is stored called RAM. In the case of PIC16F1718 this RAM is divided into 32 banks. These banks have a capacity of 128 bytes and are consecutive, that is, on a memory map, after the last address of Bank0 (0x7F), Bank1 begins (0x80). Each Bank contains 12 Core Registers, 20 Special Function Registers, 80 bytes of General Purpouse Ram and 16 bytes of common RAM. These last 16 bytes can be accessed from any Bank. It is very important, before performing any operation on a code for the MCU, to select the Bank. It is necessary to select the Bank and thus indicate to the compiler where we are working. GPRs can be addressed using Linear memory access and thus gain access to 256 bytes.

TABLE 3-3: PIC16(L)F1718 MEMORY MAP (BANKS 0-7)

PIC16(L)F1717/8/9

	BANK 0		BANK 1		BANK 2		BANK 3		BANK 4		BANK 5		BANK 6		BANK 7
000h 00Bh	Core Registers (Table 3-2)	080h 08Bh	Core Registers (Table 3-2)	100h 10Bh	Core Registers (Table 3-2)	180h 18Bh	Core Registers (Table 3-2)	200h 20Bh	Core Registers (Table 3-2)	280h 28Bh	Core Registers (Table 3-2)	300h 30Bh	Core Registers (Table 3-2)	380h 38Bh	Core Registers (Table 3-2)
00Ch	PORTA	08Ch	TRISA	10Ch	LATA	18Ch	ANSELA	20Ch	WPUA	28Ch	ODCONA	30Ch	SLRCONA	38Ch	INLVLA
00Dh	PORTB	08Dh	TRISB	10Dh	LATB	18Dh	ANSELB	20Dh	WPUB	28Dh	ODCONB	30Dh	SLRCONB	38Dh	INLVLB
00Eh	PORTC	08Eh	TRISC	10Eh	LATC	18Eh	ANSELC	20Eh	WPUC	28Eh	ODCONC	30Eh	SLRCONC	38Eh	INLVLC
00Fh	—	08Fh	—	10Fh	—	18Fh	—	20Fh	—	28Fh	—	30Fh	—	38Fh	—
010h	PORTE	090h	TRISE	110h	—	190h	—	210h	WPUE	290h	—	310h	—	390h	INLVLE
011h	PIR1	091h	PIE1	111h	CM1CON0	191h	PMADRL	211h	SSP1BUF	291h	CCPR1L	311h	—	391h	IOCAP
012h	PIR2	092h	PIE2	112h	CM1CON1	192h	PMADRH	212h	SSP1ADD	292h	CCPR1H	312h	—	392h	IOCAN
013h	PIR3	093h	PIE3	113h	CM2CON0	193h	PMDATL	213h	SSP1MSK	293h	CCP1CON	313h	—	393h	IOCAF
014h	—	094h	—	114h	CM2CON1	194h	PMDATH	214h	SSP1STAT	294h	—	314h	—	394h	IOCBP
015h	TMR0	095h	OPTION_REG	115h	CMOUT	195h	PMCON1	215h	SSP1CON1	295h	—	315h	—	395h	IOCBN
016h	TMR1L	096h	PCON	116h	BORCON	196h	PMCON2	216h	SSP1CON2	296h	—	316h	—	396h	IOCBF
017h	TMR1H	097h	WDTCON	117h	FVRCON	197h	VREGCON(1)	217h	SSP1CON3	297h	—	317h	—	397h	IOCCP
018h	T1CON	098h	OSCTUNE	118h	DAC1CON0	198h	—	218h	—	298h	CCPR2L	318h	—	398h	IOCCN
019h	T1GCON	099h	OSCCON	119h	DAC1CON1	199h	RC1REG	219h	—	299h	CCPR2H	319h	—	399h	IOCCF
01Ah	TMR2	09Ah	OSCSTAT	11Ah	DAC2CON0	19Ah	TX1REG	21Ah	—	29Ah	CCP2CON	31Ah	—	39Ah	—
01Bh	PR2	09Bh	ADRESL	11Bh	DAC2CON1	19Bh	SP1BRGL	21Bh	—	29Bh	—	31Bh	—	39Bh	—
01Ch	T2CON	09Ch	ADRESH	11Ch	ZCD1CON	19Ch	SP1BRGH	21Ch	—	29Ch	—	31Ch	—	39Ch	—
01Dh	—	09Dh	ADCON0	11Dh	—	19Dh	RC1STA	21Dh	—	29Dh	—	31Dh	—	39Dh	IOCEP
01Eh	—	09Eh	ADCON1	11Eh	—	19Eh	TX1STA	21Eh	—	29Eh	CCPTMRS	31Eh	—	39Eh	IOCEN
01Fh	—	09Fh	ADCON2	11Fh	—	19Fh	BAUD1CON	21Fh	—	29Fh	—	31Fh	—	39Fh	IOCEF
020h 06Fh	General Purpose Register 80 Bytes	0A0h 0EFh	General Purpose Register 80 Bytes	120h 16Fh	General Purpose Register 80 Bytes	1A0h 1EFh	General Purpose Register 80 Bytes	220h 26Fh	General Purpose Register 80 Bytes	2A0h 2EFh	General Purpose Register 80 Bytes	320h 36Fh	General Purpose Register 80 Bytes	3A0h 3EFh	General Purpose Register 80 Bytes
070h 07Fh	Common RAM 70h – 7Fh	0F0h 0FFh	Accesses 70h – 7Fh	170h 17Fh	Accesses 70h – 7Fh	1F0h 1FFh	Accesses 70h – 7Fh	270h 27Fh	Accesses 70h – 7Fh	2F0h 2FFh	Accesses 70h – 7Fh	370h 37Fh	Accesses 70h – 7Fh	3F0h 3FFh	Accesses 70h – 7Fh

Legend: = Unimplemented data memory locations, read as '0'.

Note 1: Unimplemented on PIC16LF1717/8/9.

Figure 6: Map memory of the first 8 Banks on PIC16F1718 [4]

2.3.2 Interrupts

When we press a button, it changes from state release to pressed. This change is an event. The MCU must manage these events, to that we use the Interrupts. An interrupt manages events that we don't know when they will happen. For example: we think about the case that our MCU is in the middle of a process and we press the button. We expect the MCU to react immediately to this event, even if it is in the middle of another task. To do that we have the interrupts, these are a signal that tells the CPU to execute the code corresponding to this event. Let's look at a practical example: our AXIOM REMOTE is executing a code that shows a menu on the LCD. By pressing a button, we hope that an option is selected from this menu. The Interrupt will handle this event, when we have button pressed, the menu code is interrupted and another code is executed.

In our case, we want to use an interrupt to notice that there has been a transition in a pin, as is the case of Bounce. All pins of the PIC16F1718 can be configured as Interrupt of Change, thus generate a signal each time it falls or raise edge. With this we can know when a falling or raising edge happens for the first time and thus notify the timer interrupt that a state change has happened.

We also have the timer interrupts. These interrupts appear every time that we can set. For example, we set the timer to 500us. So at every 500us we have an interrupt. With this we manage to set a bounce time when the first IOC is produced, which in our case are 2 timers.

3 Coding

In this section I will talk about the different codes that I have made to get all the knowledge and the necessary practice to make the debouncing code. To acquire knowledge of asm, the instructions used in the PIC16F1718 and understand the operation of it. Also to know the hardware we are using and to be skilled with the schematics present in AXIOM Remote.

Therefore, I have made a series of basic codes, such as a basic blink, to start using the code *remote_e* [3] and understand how ports work and the operation of the pins; a code to blink a LED through an interrupt of change, to understand the configuration of the interrupts, their set up, and the necessary initilization; I have also made a code to blink on the same LED with an interrupt timer, learning to set up timer and how interrupt timer works. All these codes have been necessary to understand and be able to code the different parts of the debouncing code, due to it uses everything I mentioned before.

3.1 Blink

As we have said, in this code we have been based on the code already made *remote_e*, editing its main to get a blink from the led.

```
main:

; blink on both LED (S2)
        BANKSEL TRISA
        MOVLW   10011001b        ; RA1,RA2,RA5,RA6 out
        MOVWF   TRISA

        CALL delay

        BANKSEL TRISA
        MOVLW   10011010b        ; RA0,RA2,RA5,RA6 out
        MOVWF   TRISA

        CALL delay
        CALL dloop

        BANKSEL TRISA
        MOVLW   10011000b        ; RA0,RA1,RA2,RA5,RA6 out
        MOVWF   TRISA

        CALL delay

; end blink
```

In this case we move literal 10011001 to W register, and then we move it from W to TRISA. This turns RA1, RA2, RA5, RA6 as outputs. This means that we have the led which we have on RA1 (S2A) turned on. Then, we call a delay which we have defined on the original code to make the blink last enough to make it visible

```
delay:  MOVLW    0x10
        MOVWF    C3
dloop:  DECFSZ   C1,F
        GOTO     dloop
        DECFSZ   C2,F
        GOTO     dloop
        ;DECFSZ  C3,F
        ;GOTO    dloop
        RETURN
```

Now, we switch the LEDs, so having as output RA0 and not RA1. We have turned off S2A and turned on S2B. And we call the delay again. Finally, we turn on both LEDs and call the delay again.

3.2 Interrupt of change

In this case, as we have said, we want to use an interrupt of change (IOC) to switch an LED state for each time the button is pressed.

To start with the code we need to understand what do we want to set and initialize. So we start on the line 62 of the remote_e_int.asm code where we clear LATx

```
init:   BANKSEL  LATA
        CLRF     LATA
        CLRF     LATB
        CLRF     LATC
```

we proceed with setting all the pins as input, less RA2, RA5, RA6 (TP1, TP2, TP5)

```
        BANKSEL  TRISA
        MOVLW    10011011b        ; RA2,RA5,RA6 out
        MOVWF    TRISA
        MOVLW    11111111b        ; all in
        MOVWF    TRISB
        MOVLW    11111111b        ; all in
        MOVWF    TRISC
```

and we set PORTB as digital

```
BANKSEL ANSELA
CLRF    ANSELB
```

we enable individual pull-ups

```
BANKSEL OPTION_REG
BCF     OPTION_REG,NOT_WPUEN
```

on the line 87 of the code we move literal 00111111b to W and move W to IOCBP and IOCBN. IOCBP refers to IOC rising edge and IOCBN refers to IOC falling edge. Setting bits 0-5 of this registers enables their IOC on rising and falling edges.

```
BANKSEL IOCAP
MOVLW   00111111b          ; RB0–RB5
MOVWF   IOCBP
MOVWF   IOCBN
```

and now, we enable IOC, enable peripheral and enable global interrupts

```
BSF     INTCON,IOCIE       ; enable IOC IRQ
BSF     INTCON,PEIE        ; enable peripheral IRQ—
BSF     INTCON,GIE         ; enable global IRQ
```

This would be all for initializing, and now we proceed with when entering at the interrupt of change(IOC) in the line 29, where we check the interrupt flags, if they are not set it finishes interrupt, if they are we clear them , finally we check the button on RB3 (P1), on line 37.

```
BANKSEL IOCBF
MOVLW   0xFF
XORWF   IOCBF,W
ANDWF   IOCBF,F

BTFSS   PORTB,0x03
```

We only want to switch the LED when having pressed and released the button, so by checking if the button is set or not we can make only switch the LED when the button is set, so once it has already been released (the full press has been done).

Basing on the code again, we can see that if the bit is clear it ends the IOC, if the bit is set (the button is released) we continue on line 40, by checking S1 (it is at 0x74). This register would be the LED state, so if the bit was set it moves literal 10011001b to W and then it moves W to TRISA. With this we configure

RA1, RA2, RA5 and RA6 as output pins. RA2, RA5 and RA6 were configured like this at init, the one we have changed is RA1, which is the LED S2A (we have just turned on the LED).

```
BANKSEL S1
BTFSS   S1,0
BRA     _afndj

; change state
BANKSEL TRISA
MOVLW   10011001b       ; RA1,RA2,RA5,RA6 out
MOVWF   TRISA
```

We just need to save the new state of the led. In lines 48, 49 we clear the bit we were using of S1 for this purpose. And then, at the line 50 we go to finish the interrupt of change.

```
BANKSEL S1
BCF     S1,0
GOTO    _iocx
```

Going back where checking the LED state (S1,0), if the bit were clear we continue on line 52, by clearing the bit of RA1 on TRISA (lines 53, 54) and saving the state of the LED as set (S1,0).

```
MOVLW   10011011b       ; RA2,RA5,RA6 out
MOVWF   TRISA
```

Finally, on the line 59 we finish the interrupt (RETFIE)

```
_iocx:  RETFIE
```

The main keeps running, as the main purpose of the code is to detect the interrupts.

3.3 Timer interrupt

This code, as the previous one, is performed for understanding the operation of interrupts, in this case the timer interrupt. We want to make blink the S2A LED through the interrupter timer, that is, to make a pin state switch each time the set time for the interrupt occurs.
As before we start checking how to initialize, and to start, we enable individual pull-ups on the line 44

```
BANKSEL OPTION_REG
BCF     OPTION_REG,NOT_WPUEN
```

and we enable peripheral, global interrupts and Timer1.

```
BSF     INTCON,PEIE     ; enable peripheral IRQ—
BSF     INTCON,GIE      ; enable global IRQ
BANKSEL PIE1
BSF     PIE1,TMR1IE     ; enable
```

We need to set Timer1 to the required timing. For this, we first need to configure its control register (T1CON). By default the bits are cleared, so for enabling Timer1 we need to set TMR1ON (line 60) and for having our wanted timing value we need 1:1 as input clock prescale value and LFINTOSC as clock source for having 31kHz as the Fosc (due to Focs is set to INTOSC , so we finally set TMR1CS bits for using LFINTOSC (lines 58, 59), and let everything else as default.

```
BSF     T1CON,6
BSF     T1CON,7
BSF     T1CON,TMR1ON    ; enable Timer1 On bit
```

For the moment, as it is only to initialize, we set the Timer1 at the shortest value, as it rolls over from FFFFh to 0000h, we want to set it at the maximum value, with that, at the next moment it will happen the interrupt and start the interrupt part. For it to happen we set TMR1H and TMR1L to 0xFF moving these values to the registers.

```
MOVLW   0xFF
IORWF   TMR1H,F
IORWF   TMR1L,F
```

When we check TMR1IF, if it is set we enter on the interrupt processing code. We first want to set the time, so for 0,5s of delay (for the blink to be visible), we should set the register pair (TMR1H:TMR1L) to C3:72
31kHz*0,5s=15500 ->3C8C FFFF - 3C8C - 1 = C372 (1 because of the roll over)

```
BANKSEL TMR1H   ; FFFF - 15500 (in hex) ... 31k*0,5s=15500
MOVLW   0xC3    ; 500ms delay
MOVWF   TMR1H
MOVLW   0x72
MOVWF   TMR1L
```

Once the time is set, we want to switch the LED's state. We do this moving 0x02 to W (for having the bit 1) and doing XOR function with TRISA and saving it on the register itself. This results on having switched bit 1 of TRISA (where S2A LED is).

```
BANKSEL TRISA
MOVLW   0x02
XORWF   TRISA,F
```

Done this, we must clear Timer1 interrupt flag (lines 36,37), and we exit the interrupt.

```
BANKSEL PIR1
BCF     PIR1,TMR1IF
```

3.4 Debouncing code

Finally, in the debouncing code we will use these interrupts to carry out the result that we want.
Our code will be waiting for interrupts. When it detects an interrupt it will check first if it is an interrupt of change, and process it if it is. After this, it will check the timer interrupt, and as before, process it or not and end the interrupt.
Now, I will explain the code we are using. This first code would performe a debouncing for the bounces sent via SPI from PIC32 (on RB6), and for the button P13 (RB5), returning the output to PIC32 via USART.

Once again, we start commenting the initialization, in line 126 where we clear LATx, we only set bit 7 of LATB (RB7) as it is the one used to transmit the output via USART

```
init:   BANKSEL LATA
        CLRF    LATA
        CLRF    LATB
        BSF     LATB,7              ; RB7 default high
        CLRF    LATC
```

and now, on TRISx, we set all the pins I/O as required. We will use RA4, RA3, RB2, RB3, RB5, RC2 and RC3 as buttons, so they must be inputs. RB6 should also be input as we use it for testing the transmitted 'pulses' from PIC32 using SPI. We use RB7 to send back the output, so it must be an output.

```
        BANKSEL TRISA
        MOVLW   10011011b           ; RA2,RA5,RA6 out
        MOVWF   TRISA
        MOVLW   01111111b           ; RB7 out
        MOVWF   TRISB
        MOVLW   11111111b           ;  all  in
        MOVWF   TRISC
```

We set PORTB as digital.

```
        BANKSEL ANSELA
        CLRF    ANSELB
```

and we enable individual pull-ups

```
        BANKSEL OPTION_REG
        BCF     OPTION_REG,NOT_WPUEN
```

now we initialize values (to 0).

```
        CLRF    S5
        CLRF    S6
        CLRF    F1
```

On line 156 we do the IOC setup, we enable falling edge interrupts for PORTB bits, from 1 to 6.

```
BANKSEL IOCAP
MOVLW   01111111b         ; RB0–RB6
;MOVWF  IOCBP             ; we check falling edge first (N)
MOVWF   IOCBN
```

C1 will be used for checking the buttons state. For now no button should be pressed, so we can initialize C1 as 0xFF.

```
MOVLW   0xFF
MOVWF   C1
```

we enable interrupt of change.

```
BANKSEL INTCON
BSF     INTCON,IOCIE      ; enable IOC IRQ
BSF     INTCON,PEIE       ; enable peripheral IRQ—
BSF     INTCON,GIE        ; enable global IRQ
BANKSEL PIE1
BSF     PIE1,TMR1IE       ; enable
```

Here, on line 174 it is used the same way to initialize Timer1 as has happened before on timer interrupt's code.

```
BANKSEL T1CON
BSF     T1CON,6
BSF     T1CON,7
BSF     T1CON,TMR1ON      ; enable Timer1 On bit
MOVLW   0xFF
IORWF   TMR1H,F
IORWF   TMR1L,F
```

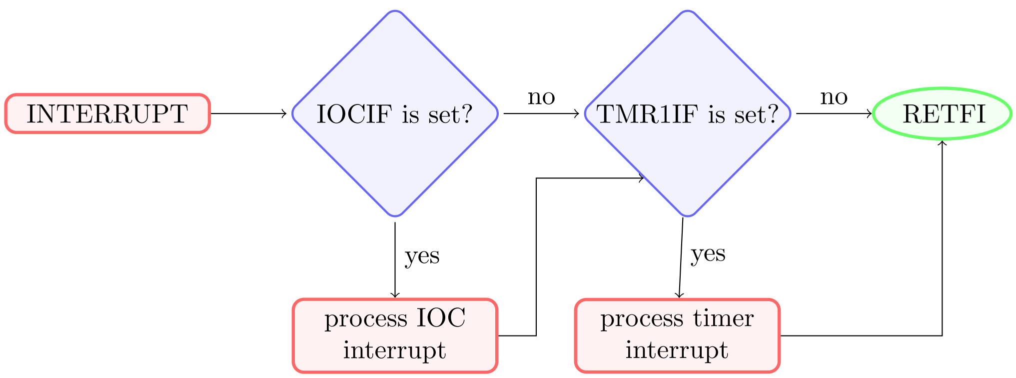
We set Fosc=INTOSC=16MHz (for this we set IRCF to 1111 on lines 192-194)

```
BANKSEL TX1STA
MOVLW   0                 ; Baud rate 1Mbps
MOVWF   SPBRG

BSF     TX1STA,BRGH       ; High speed baud rate
BSF     RC1STA,SPEN
BSF     TX1STA,TXEN       ; Configures TX1STA

BANKSEL OSCCON
MOVLW   01111000b         ; IRCF to 16MHz
MOVWF   OSCCON
```

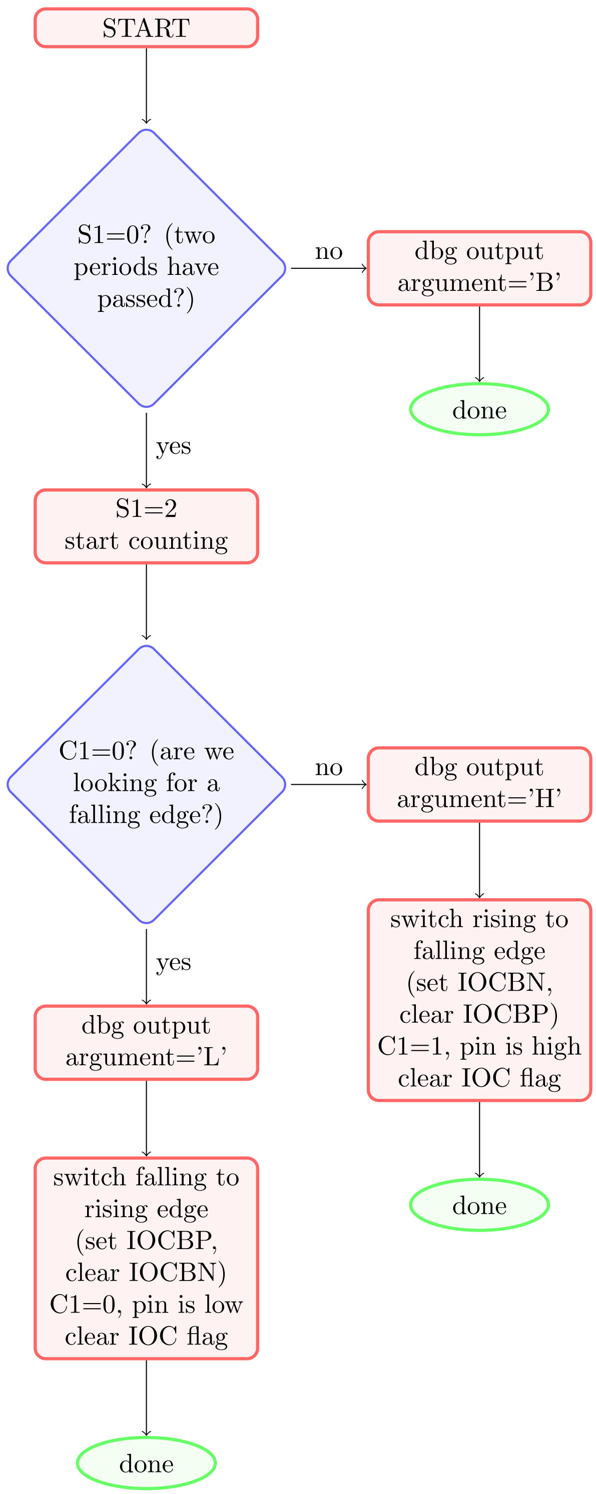
The desired baud rate is 1MBaud, so we must do this evaluation: SPBRG=Fosc/BaudRate/16 - 1=16000000/1000000/16-1=0, and for all of this we must select high baud speed (set BRGH). We enable the transmit, and finally we configure RB7 as TX/CK (we can check table 12-2 on PIC16F1718 datasheet [4] for checking possible pins).

```
BANKSEL  RB7PPS
MOVLW    10100b                    ; Output Source Selection TX/CK
MOVWF    RB7PPS
GOTO     main
```

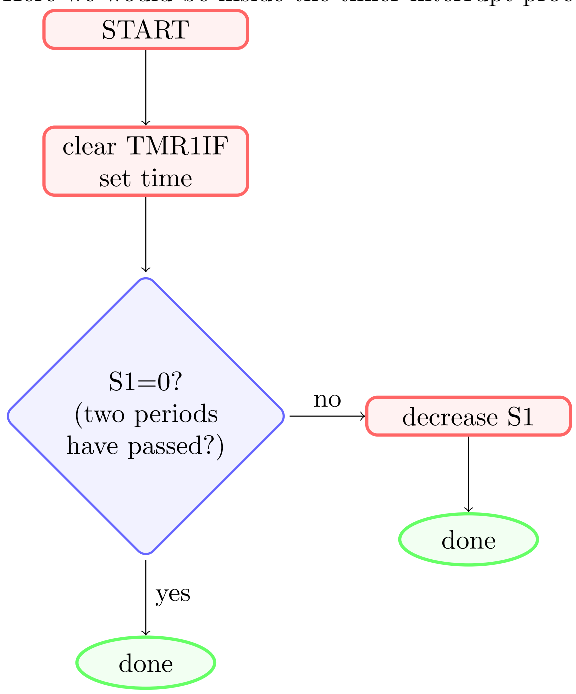
The rest of the code should be understood via this diagrams:



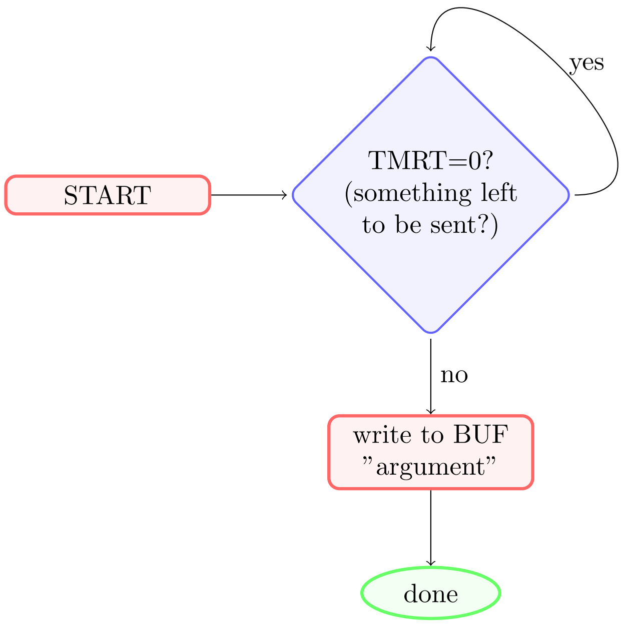
Here is what would happen inside the IOC process box:


START
S1=0? (two periods have passed?)
no
dbg output argument='B'
done
yes
S1=2
start counting
C1=0? (are we looking for a falling edge?)
no
dbg output argument='H'
switch rising to falling edge (set IOCBN, clear IOCBP) C1=1, pin is high clear IOC flag
done
yes
dbg output argument='L'
switch falling to rising edge (set IOCBP, clear IOCBN) C1=0, pin is low clear IOC flag
done

Here we would be inside the timer interrupt process box:



And finally, when we are at the dbg_output box:



The last code, debouncer_kmw.asm includes the rest of the buttons of KMW. Finally, this code tests the bounce on RB6 (SPI), RB5 (P13), RB2 (TS1A), RB3 (TS1B), RA4 (S1A), RA3 (S1B), RC2 (TS2A) and RC3 (TS2B). You can access to this code and the rest of the code via GitHub.

3.5 Script

Finally, for testing the code, I have written some lines which can be used to run it, configure it as wanted, and test some pulses (those can be changed).

This code is on the GitHub as comm.sh.

There are two different ways to have our wanted value, one of them is to write it directly on the script, as it is right now, and the other one is the "commented" way, in order to run it always like the same, but entering different values on terminal once the script is running.

As said, this different values we can enter are the "pulses" which imitate the button presses and bounces. In this case we have:
80000000 00003F00 00000000 FFFFFFFF

If we have this value in binary we can see those "pulses", as every bit lasts 1us on our code, so in this case, we would have high for 1us, then low for 49us, 6us on high again, 840us low and finally, 32us high.

With this, to configure our program, we need to write ?0=1=- ? for KTest, so to enable the receiver, 0=1= to set mclr,pclk and pdat, and finally, - to negate the sequence output, this last one is to fix that the SPI peripherial reverts to low on the output when disabled, so we need to invert it.

Once this is done, we send the sequence we wanted to by sending 0 first, in order to set the sequence index to 0; we send every "word" adding — after it to set the sequence data, and finally, we send 01D*. This means to start from 0 (0) and until 1D, which is the number of the words we want to send (in hexadecimal). The * activates the sequence, so start sending it.

As the ser_dbg.c code is done we can write up to 256 words of 32 bits, so 256 words of 8 hexadecimals. This also means the maximum length of our pulse can be 8192us.

How to run all the different codes is explained on the GitHub, where each code is, with its required commands.

4 Conclusion

I have been learning a lot all of this Summer. I really enjoyed participating on GSoC and think that with the knowledge I had I have achieved a great code.

The project could be upgraded by adding some more actions on the debouncing code, on debouncing_kmw.asm

Some of this things could be:

Check periodically the button state when we know for sure that the button is not bouncing. We were assuming that we start looking for a falling edge, and we keep switching from this assumption, so checking this to be sure would be an important part of the upgrade.
It should be also extended to the other PIC16 (KME), as it doesn't have the same buttons and should be adapted.
It should also be checked its accuracy. It could be great for it to report the estimated time of bouncing and extract from this a more accurate range of time to be skipped for avoiding the bounce.
One more important part would be to implement the part where the PIC32 receives all the buttons states, and it has to process it to make the Remote to react accordingly. For example switching menu or selecting the required option.

References

[1] apertus° open source cinema. https://apertus.org.

[2] Elliot Williams. Embed With Elliot: Debounce Your Noisy Buttons, Part I. https://hackaday.com/2015/12/09/embed-with-elliot-debounce-your-noisy-buttons-part-i/.

[3] H.Poetzl. Axiom remote stuff.

[4] Microchip Technolgy Inc. Pic16(l)f1717/8/9 data sheet.

[5] Tony R. Kuphaldt. Lessons in electronic circuits: Vol. iv - digital. https://www.allaboutcircuits.com/textbook/digital/chpt-4/contact-bounce/.