Introduction
In this experiment, you will familiarize yourself with the Ubicom SX28 microcontroller and the SX-Key Development system. This experiment will also help you see how to program a small microcontroller to control digital outputs based on digital input values. After some simple delay examples, an interrupt service routine is used to establish a time base for events. Writing subroutines is also introduced. The projects in this lab make use a pair of pushbuttons and an LED to demonstrate some basic input/output control techniques. These techniques are extended to servo control and infrared object detection.

Document Conventions
This document makes use of four symbols that dictate your activities:

- Instruction for the student to follow.
- Required lab notes.
- Demonstrate to lab instructor/TA.
- Lab report item.
Pre-Lab:

✓ Use the links in the Resources section at the end of this document to obtain:
(1) SX-Key.exe, (2) The SX-Key/Blitz Development System Manual v1.1.
✓ Enter all code examples into SX-Key.exe and save on a floppy (bring to lab).
✓ Look up and attempt to understand all assembly mnemonics and directives that you are not already familiar with. Using Adobe Acrobat Reader’s search function will assist in quickly locating terms in the manual. The Manual’s Appendix B: SX Instruction Set contains explanations of all assembly instructions for the SX microcontroller used with the SX-Key development system.
✓ Write up narratives for all three example program listings.

Applications

The SX28 microcontroller and SX-Key development system has many features common to microcontroller development environments:

- A programming tool – the SX-Key
- In-circuit debugging tools – the SX-Key in conjunction with the SX-Key software is used to view register values, I/O pin values, the clock, flag bits, and events.

The pushbutton and LED circuits used in this laboratory are common user interface components found in many electronic products such as alarm systems, sprinkler timers, cell phones, and PDAs.

The concept of establishing a time base using the interrupt service routine extends far beyond the user interface. Once you have a time base, the microcontroller can be programmed to control multiple processes such as serial communication, sampling the user interface, motor control with PWM, communication with peripherals such as sensors, memory, and special purpose integrated circuits.

Parts

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Part Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10 kΩ resistor (Brown, Black, Orange)</td>
</tr>
<tr>
<td>1</td>
<td>470 Ω resistor (Yellow, Violet, Brown)</td>
</tr>
<tr>
<td>2</td>
<td>Normally open pushbutton switches</td>
</tr>
<tr>
<td>1</td>
<td>SX-Key</td>
</tr>
<tr>
<td>1</td>
<td>SX Tech Board</td>
</tr>
<tr>
<td>1</td>
<td>7.5 V, 1000 mA AC adaptor</td>
</tr>
</tbody>
</table>
Activity #1: SX-Key Hardware, Debugging Environment, Test Circuit, and First Program

Figure 1.1 shows the hardware setup for the SX-Tech Tool Kit. Plug the SX-key into the 4-pin header on the SX Tech Board exactly as shown in the figure. The other end of the SX-Key should be connected to a serial cable. You will need to plug the other end of the serial cable into an available serial port on your PC and the 7.5 V, 1000 mA supply connected to the SX-Tech board must also be plugged into a wall outlet.

Build the circuit shown in Figure 1.2. Note, this is slightly different from the LED circuit pictured in Figure 1.1.
If you have not already done so, download and run SX-Key.exe. Instructions for this can be found in the Resources section at the end of this document.

Load Program Listing 1.1 from your floppy disk.

; Program Listing 1.1
; ---- Assembler Directives ----------------------------------------------
device SX28L,oscxt2 ; SX28L, external feedback for 4 MHz res.
device stackx_optionX,turbo ; Extend stack & option registers & turbo.
freq 4_000_000 ; Debug frequency.
reset start ; Go to 'Start' label on reset.

; ---- Watch Directives -----------------------------------------------
watch timer_1,8,udec ; Watch timers.
watch timer_2,8,udec
watch rb,8,ubin ; Watch I/O port B.

; ---- I/O Pin Definitions -------------------------------------------
LED_line = RB.6 ; Alias name for port B pin 6.

; ---- RAM Declarations ---------------------------------------------
Input/Output Control Using Interrupt Service Routines to Establish a Time

```
org 8      ; Global Registers.
timer_1   ds 1     ; Two 8-bit registers for
timer_2   ds 1     ; 16-bit counter.

; ---- Interrupt Service Routine -----------------------------------------
org 0      ; Program memory origin.

; ---- Boot Routine------------------------------------------------------
start  mov !option,#%10001000    ; RTCC enabled, reg 0 = w
        mov !rb,#%10111111     ; RC.7 -> output.
        clr timer_1       ; Clear timers
        clr timer_2

; ---- Main Routine ------------------------------------------------------
main   inc timer_1       ; Increment timer low byte.
        snz             ; Increment timer high byte
                         ; on rollover.
        inc timer_2
        jnz main       ; Go to main if timer_2 != 0.
        not LED_line   ; Invert RB.6.
        jmp main       ; Infinite loop.

✓ Select Debug from the SX-Key editor’s Run menu.

Figure 1.3 (next page) shows the SX-Key Editor’s in circuit debugging environment, which is comprised of four windows:

1) Registers – Shows:
   a. all SX28 chip’s registers
   b. Program addresses, machine codes, and assembly commands.
2) Debug – Buttons to control debugging and bring windows to the foreground.
3) Watch – View register values specified in assembly code by the “watch” directive.
4) Code – Shows source code with a blue bar to denote the command about to be executed and a red bar showing a breakpoint.

✓ Click the Run button in the Debug window
✓ Verify that the LED blinks on and off at about 4 Hz. Trouble-shoot the circuit if needed.
✓ Click the Stop button and make a note of the values in the watch window.
✓ Compare these values to the RB register and global registers 08 and 09.
Click the line of code with the `not LED_line` command in the Code window. This should cause the line to be highlighted in red signifying a breakpoint.

Click the Poll button.

Note that the program is running, but the debug information in the registers window is updated every time the program passes the breakpoint. The value of the `RB` register should be updated in the Watch and Registers window.

Can you determine from this the relationship between `RB` and the LED?

---

Figure 1.3: SX-Key in-circuit debugging environment.

---

Click Stop to halt the polling process.
Click Run, and the program will run full speed until it gets to the breakpoint.
Click Step 10 or 15 times to step through the program one assembly command at a time.
Click the number to the right of the timer_1 label in the Watch window.
Enter a value of 253.
Click the number to the right of the timer_2 label in the Watch window.
Enter a value of 255.
Record each assembly command that was executed and how the register values change for the next 20 clicks of the step button.

Explain how the value of the LED was toggled by incrementing the timer in terms of the assembly code that was executed.

Click the Walk button.

Record what happens.

Click the Stop button.

Click the jmp main line to move the breakpoint to that line.

Use the Watch window to set the value of the timer_1 and timer_2 to 250 and 255 respectively.

Click the walk button again.

Explain what just occurred.

Clear the breakpoint by clicking the red, highlighted line jmp main command in the code window.

Click run.

Click poll a few times.

Explain the differences in the debugging environment’s behavior:

- When the Poll button is selected when a breakpoint is set.
- When the Poll button is selected when there is no breakpoint and the program is running.

Click Stop.

Select your own breakpoint in the Code window.

Click poll.

Click reset. What effect did this have on the active command line?

What happens with the first few steps?

Devises a sequence of setting breakpoints and use of poll, walk, and step to effectively demonstrated and explain the theory of operation of a 16-bit counter to your instructor.

Click Quit to exit the debug session.

Project #1: Basic I/O Control

Modify Program Listing 1.1 so that the LED only flashes on and off when the pushbutton attached to RB.2 is pressed. You can add a command such as this to the main routine:

```
loop jb rb.2, loop
```

One problem with this modification is that it does not force the LED into an off state when the button is not pressed. The LED could either be on or off. Modify the main routine so that:
• The LED flashes on/off when the button is pressed
• The LED is always off when the button is not pressed

Activity #2: Interrupt Service Routine for Establishing a Time Base

Program Listing 1.2 causes a similar behavior with the LED, but for a completely different reason. Program listing 1.2 toggles the LED in a code loop called the interrupt service routine and labeled `isr`. Note that all the main routine does is jump to itself, yet the LED is still controlled.

```plaintext
; Program Listing 1.2
; LED on/off Using an Interrupt Service Routine.

; ---- Assembler Directives -------------------------------------
device SX28L,oscxt2 ; Chip, osc-feedback.
device stackx_optionX,turbo ; extend, 1 inst/Hz.
freq 4_000_000 ; Debug frequency.
reset start ; Program entry.

; ---- Watch Directives ----------------------------------------
watch rb,8,ubin ; Watch directives for
watch timer_A,8,udec ; viewing variables in
watch timer_B,8,udec ; the watch window.

; ---- I/O Pin Definitions -------------------------------------
LED_line = RB.6 ; Alias definition.

; ---- RAM Declarations ----------------------------------------
org 8 ; Gen purpose RAM
	timer_A ds 1 ; Declare 2 registers
timer_B ds 1 ; = 16 bit timer.

; ---- Interrupt Service Routine --------------------------------
org 0 ; Program line 0.
isr inc timer_A ; Increment timerA.
	snz ; If timerA rollover,
	inc timer_B ; increment timerB
movb LED_line, timer_B.5 ; Timer bit toggle LED.
mov w,#-63 ; Load -63 into w.
retlw ; Load 2 into RTCC for
timing of next interrupt.

; ----Boot Routine----------------------------------------------
start mov !option,#%10001000 ;
mov !rb,#%10111111
```
Input/Output Control Using Interrupt Service Routines to Establish a Time

```asm
clr timer_A
clr timer_B
clrb LED_line

; ---- Main Routine --------------------------------------------
main jmp main
```

The Interrupt Service Routine, Boot Routine, and Main Routine interact in ways that are not immediately apparent.

- Use the debugging environment to follow the program’s progress with just the Walk button and no breakpoints.
- What is happening periodically to the program?
- What is the RTCC register doing and how does it relate to the periodic visits to the Interrupt service routine.
- Try setting breakpoints at various places within the interrupt service routine and use Walk, Step, and Poll to gather more information about its behavior.

Let’s take a detailed look at this program. The program begins at the `start` label. The first command that is executed is a value moved into the `!option` register.

**NOTE:** The `#` operator tells the assembler that it is working with a constant (a number). If you leave out the `#` operator, the assembler will assume you want to perform an operation on the register at the specified address.

Each bit in this register has specific functions. Note that bits 3 and 7 are set. Bit-7 gives program access to the `w` (working) register. Bit-3, configures the SX chip so that each time the RTCC register (real time clock counter) rolls over (from 255 to 0), an interrupt request is serviced. Bits 0 through 2 can be used to scale the number of clock cycles it takes to increment the RTCC register. By setting/clearing combinations of these bits, you can drastically slow down the frequency of the flashing LED.

```asm
; ----Boot Routine----
start mov !option,#%10001000
```

The remainder of the boot routine is straightforward. The direction bits in I/O register B are configured by moving a binary value into the `!rb` register. Bit 6 is cleared, which causes `RB.6` to function as an output. Bits 0 through 5 and bit 7 are set, which cause `RB.0` through `RB.5` and `RB.7` to function as inputs. The `clr` operator is used to clear the contents of the two `timer` registers and `LED_line`, which was declared to be `RB.6`.

---

Page 1.9
The main routine is one line of code that keeps jumping to itself. This would seem to
be in infinite loop were it not for the fact that interrupts are serviced based on the
RTCC register, which increments with the clock. Keep in mind the clock is oscillating
4 MHz.

Each time the RTCC register rolls over to zero, an interrupt occurs, and the SX chip is
designed to automatically go to line zero of the program (org 0) to service an
interrupt. This being the case, the code that should be executed every time an
interrupt occurs is strategically placed here. The first four lines of the interrupt
service routine feature a 16-bit counter and a command that toggles the state of
the LED based on a bit in the counter.

The last two lines of the interrupt service routine are of interest because they reset the
value of the RTCC register to a particular value (−63) before returning to the main
routine. All stack management is maintained in hardware, so the return to the next
line of code in the main routine is automatic. The retiw command moves
the working register, which contains −63 into the RTCC. This guarantees that the
interrupt will be serviced again in 63 clock cycles. This establishes a time base.

You can calculate the frequency that the timer is incremented by dividing the return
value by the clock rate. You can also derive the fundamental time base established by
the interrupt service routine by taking the inverse of the interrupt frequency.
Input/Output Control Using Interrupt Service Routines to Establish a Time

\[ f_{\text{interrupt}} = \frac{-RTCC}{f_{\text{clock}}} \]

and

\[ T_{\text{interrupt}} = \frac{1}{f_{\text{interrupt}}} \]

✓ Use this formula to calculate the interrupt frequency and the inverse to calculate the interrupt interval in Program Listing 1.2.

**Project #2: Output Control Using an ISR Time Base**

You can control events in the foreground (the main routine) based on events in the ISR (the background process). To move LED control from the interrupt service routine to the main routine, simply move the command:

```assembly
movb LED_line, timer_B.5
```

from the `isr` to the `main` routine. So long as you moved the command so that it falls between the `main` label and the `jmp main` command, the LED should behave the same way when the program is run.

- Explain why the LED’s frequency has not changed.
- Explain why the LED’s state is now updated 16 times as fast as it was before.

(b) Modify the main routine so that the LED flashes twice as fast as in the example.
(c) Modify the main routine so that the LED flashes half as fast as in the example.

Look up commands beginning with `cj` in the *SX-Key/Blitz Developers Manual*. For example, `cjne` performs a “compare and jump if not equal”. You can use compare and jump commands to make decisions in your code. Also, look up the commands using `dj`. `djnz` can be used along with a `mov` command to create a loop. For example:

```assembly
counter ds 1
.
.
mov counter, #50
loop
```
Note that in this example, another register alias (variable) is declared in the RAM Declarations section for use in making comparisons. You can also reset the value of a timer that is incremented in the ISR in your foreground code (the main routine). You can also glean a few hints on decisions and flow control from Program Listing 1.3.

Use these hints to modify your main routine so that:

- (d) the LED to flash on/off at the rate in part 2 (b) for 60 repetitions followed by flashing on/off at the rate in part 2 (c) for 15 repetitions.
- (e) the LED speeds up to the limit of persistence of vision then slows down to 5 to 10 Hz.
- (f) the LED behaves as in part 2 (e) under pushbutton control. Use one pushbutton to increase the frequency and the other to decrease it.

Using the ISR Time Base in Routines and Subroutines

Now, let’s examine how the code in the main routine and subroutines can make use of the time base established in the interrupt service routine. The main routine in Program Listing 1.3 monitors a pushbutton and calls a flash LED subroutine or a delay subroutine depending on whether the pushbutton connected to RB.2 is pressed or not pressed.

Program Listing 1.3 differs from the previous one in three ways:

1) The LED is no longer switched in the ISR.
2) The main routine makes decisions and calls subroutines.
3) One subroutine monitors the counters that are automatically incremented in the ISR and makes decisions (controls the LED) based on their values. The other subroutine delays to debounce the pushbutton.

Remember, once established, the ISR is visited at a frequency of \( f_{\text{interrupt}} \). Since there is a 16-bit counter that is incremented at \( f_{\text{interrupt}} \), the code in the main routine can depend on the time base and counters to make decisions. This program also introduces subroutines and some of the labeling conventions that make subroutine code easier to write as well as more readable.
Input/Output Control Using Interrupt Service Routines to Establish a Time

; ---- Assembler Directives -----------------------------------------------
   device SX28L,oscxt2       ; Chip, osc-feedback.
   device stackx_optionX,turbo ; extend, 1 instrct/Hz.
   freq 4_000_000       ; Debug frequency.
   reset start       ; Program entry.

; ---- Watch Directives -----------------------------------------------
   watch rb,8,ubin     ; Watch directives for viewing variables in
   watch timer_A,8,udec ; the watch window.
   watch timer_B,8,udec

; ---- I/O Pin Definitions -------------------------------------------
   LED_line = RB.6     ; Alias definitions for
   PB_line  = RB.2     ; pushbutton & LED I/O.

; ---- RAM Declarations -------------------------------------------
   org 8        ; Gen purpose RAM origin.
   timer_A  ds 1     ; Declare 2 registers = 16 bit timer.
   timer_B  ds 1     ; 8-bit counter.

; ---- Interrupt Service Routine ---------------------------------
; NOTE: LED toggle removed from ISR.

   org 0        ; Program line 0.
   isr  inc timer_A       ; Increment timerA.
      snz         ; If timerA rollover,
      inc timer_B       ; increment timerB
      mov w,#-63       ; Load –63 into w.
      retiw        ; Load 2 into RTCC for timing of next interrupt.

; ---- Boot Routine ------------------------------------------------
   start  mov !option,#%10001000  ; Enable RTCC interrupts
       mov !rb,#%10111111   ; RB.6 -> output
       clr timer_A     ; Initialize timerA = 0
       clr timer_B     ; timerB = 0
       clrb LED_line    ; LED off.

; ---- Main Routine -----------------------------------------------
   main  jb PB_line,debounce ; Debounce if not pressed
       call flashLED    ; Flash LED sub if pressed
       jmp main       ; Jump to main
   debounce call delay   ; Call delay sub
The main routine makes decisions on what to do based on whether the pushbutton bit is set. If it is set, the program jumps to the debounce label, otherwise, it moves to the next line, which calls the flashLED subroutine. After the flashLED subroutine is called, the next command is a jump to the main label. If the pushbutton bit is set, the program calls flashLED instead, followed by a jump to the main label.

The subroutines make decisions based on the values of the timers. Note that the timers are incremented automatically every 63 clock cycles because of the mov and retiw instructions at the end of the interrupt service routine in conjunction with certain bits that were set/not set in the !option register at startup.

The subroutine names flashLED and delay are global labels. This means that you cannot repeat them elsewhere in the program. Labels that begin with a colon, such as :loop can be re-used in different routines and subroutines. In terms of scope, they are attached to the label that precedes them. Note how both the flashLED and delay subroutines both have local :loop labels. Note that each subroutine has a conditional jump to a local :loop label. The conditional jump in the flashLED subroutine jumps to the :loop found in that subroutine while the conditional jump in the delay subroutine jumps to the :loop found its subroutine.
A. Pushbutton Control of Pulse Width

The servos you will be provided with are modified so that they rotate continuously. Instead of holding a position somewhere within a 180° range of rotation, these servos function as continuous rotation motors. The signal that made the servo turn to 180°, now makes it turn full speed counterclockwise. The signal that made the servo turn to 90° (the center position) now makes it stay still. The signal that made it turn to 0° now makes it rotate full speed clockwise.

A servo expects a signal similar to the one we have been sending to the LED. The only difference is the duration of the on and off times. The off time is not critical for servo control, but it should remain in the neighborhood of 20 ms between pulses. The pulse width is what controls the position of a normal hobby servo, which translates to speed and direction for our modified servos.

✓ Leave the pushbutton circuits connected, but replace the LED circuit with a servo using the parts from Table 1.2 and the circuit shown in Figure 1.4.

<table>
<thead>
<tr>
<th>Table 1.2, Servo Circuit Parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Misc</td>
</tr>
</tbody>
</table>

![Figure 1.4 Servo Circuit.](image)

The SX chip needs to be programmed to send a pulse signal to the servo every 20 ms. The pulse width is what controls the servo rotation as shown in Figure 1.5 and table 1.3.
Vdd = 5 V

Vss = 0 V

1.0 ms <= highTime <= 2.0 ms

LowTime = 20 ms

Figure 1.5 Pulse train sent to servo, not to scale.

Table 1.3, Pulse Widths for Servo Control

<table>
<thead>
<tr>
<th>Time, ms</th>
<th>Servo Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Full speed clockwise</td>
</tr>
<tr>
<td>1.25</td>
<td>Half speed clockwise</td>
</tr>
<tr>
<td>1.50</td>
<td>No rotation</td>
</tr>
<tr>
<td>1.75</td>
<td>Half speed counterclockwise</td>
</tr>
<tr>
<td>2.00</td>
<td>Full speed counterclockwise</td>
</tr>
</tbody>
</table>

Modify the program you developed in Project 2(f): pushbutton control of LED Frequency so that you can use the pushbuttons to step through the servo speeds listed in Table 1.3. You will need to make use of timer_A to control the 1 to 2 ms pulse widths required by the servo.

B. Proportional Control Loop

✓ Build and test the distance detector introduced in the document entitled:

*Distance Detection for Personal Robotics*

*Infrared Emitting Diode & 40 kHz Infrared Detector*

The application is kind of fun because it uses the infrared transmitter in a TV remote and the infrared receiver in the TV to make low resolution distance measurements. Unlike the example programs in this document, this program
performs a frequency sweep by adjusting the value loaded into `w` before the `retiw` command in the interrupt service routine. This changes the `f_{interrupt}`, and it’s used to perform a frequency sweep on a bandpass filter. The infrared hardware is available in your kit.

1. Modify Program Listing 1.1 in so that the measured values are multiplied by a constant and then stored as a two ASCII digits, example: 01, 05, 11, 14, etc.

2. Look up the Watch Directive in the SX-Key/Blitz Development System Manual and use it to display the two ASCII digits.

3. Incorporate the distance detection subroutine from Program Listing 1.1 in *Distance Detection for Personal Robotics, Infrared Emitting Diode & 40 kHz Infrared Detector* so that it works in the servo control program you developed in Project 2: Pushbutton Control of Servo.

4. Modify the code so that instead of pushbuttons controlling the servo, the detected distance determines the servo’s motion. When the object is close, the servo should turn full speed counterclockwise. When the object is far, the servo should turn full speed clockwise. When the object is in the middle of the distance range, the servo should stay still. The pushbuttons should be used to enable and disable the system.

The recommended design for controlling the servo using the distance sensor should be a proportional control loop shown in Figure 1.6.

![Proportional control loop](Diagram)

**Figure 1.6** Proportional control loop.

In words, this figure is telling you to take the measured distance and multiply it by a constant. Then add/subtract that value to/from the
value that makes the servo stay still. Use your result value to set the timer to pulse the servo.

Resources
Documentation and software for the SX-Key and SX chips is available for free download from www.parallaxinc.com. The web address that takes you directly to the SX Tech Downloads page is:


Below are the links you should download to complete these projects:

- Self-extracting archive of the SX-Key v1.30
  - 340 KB
  - Software (for Rev. E/F SX-Keys) and addendum information. Supports the 18/28/48/52-pin chips in one executable.

- SX-Key and SX-Blitz Manual v. 1.1
  - 1.2 MB

There are also free, downloadable tutorials for the SX-Tech toolkit:

- Introduction to Assembly Programming with the SX
  - Version 1.2
  - 1.3 MB

- I/O Control with the SX Microcontroller
  - 1.2 MB