CS 431
Lab Manual
Spring 2014 (*Revision*: 1.1.0)

Table of Contents

1 Reference Documents

1. **dsPIC-datasheet.pdf**: General description of the dsPIC33f family (including description of all modules and their registers).

2. **dsPIC33F.xx - xxxx.pdf**: Full dsPIC33f manual with detailed description of dsPIC33f's components (e.g. **dsPIC33F.11 - Timers.pdf** is the full manual for the Timers which is an extention of Section 11 in **dsPIC-datasheet.pdf**)
3. **dsPIC-C-compiler.pdf**: dsPIC microcontroller C compiler manual.
4. **LCD-datasheet.pdf**: Datasheet of the LCD module from SparkFun. Includes basic concept of operation and supported commands.
5. **MCP4822-datasheet.pdf**: Datasheet of the MCP4822 dual output Digital to Analog Converter. Includes concept of operation, command registers and timing sequences.
6. **MPLab-IDE.pdf**: manual for MPLab IDE.

2 Introduction

A significant portion of CS 431 is laboratory work. This lab manual is designed to supplement the course lectures with the detailed information and references required to carry out the lab assignments. Sections 3.3 and 4 discuss the dsPIC33 microprocessor and the Amazing Ball System. Section 6 talks about Linux programming and the PCI-DAS1602/12 card.

3 Microchip dsPIC Microcontroller (dsPIC33FJ256MC710)

dsPIC33F (Figure 1) is 16-bit microcontroller using a modified Harvard architecture. The following table summarizes some key parameters of dsPIC33F.

Parameter Name	**Value**
Architecture	16-bit
CPU Speed (MIPS)	40
Program Memory (KB)	256
RAM Bytes	30,720
I/O Pins	85
Pin Count	100
Digital Communication Peripherals	2-UART, 2-SPI, 2-I2C
Analog Peripherals	2-A/D 32x12-bit at 500(ksps)
Timers	9 x 16-bit 4 x 32-bit
CAN	2 ECAN
16-bit PWM resolutions	16
DMA	8

3.1 Microcontroller Programming Techniques

In this course all programming for the microcontroller will be done in C on the Microchip MPLab IDE and compiled with an dsPIC cross-compiler.

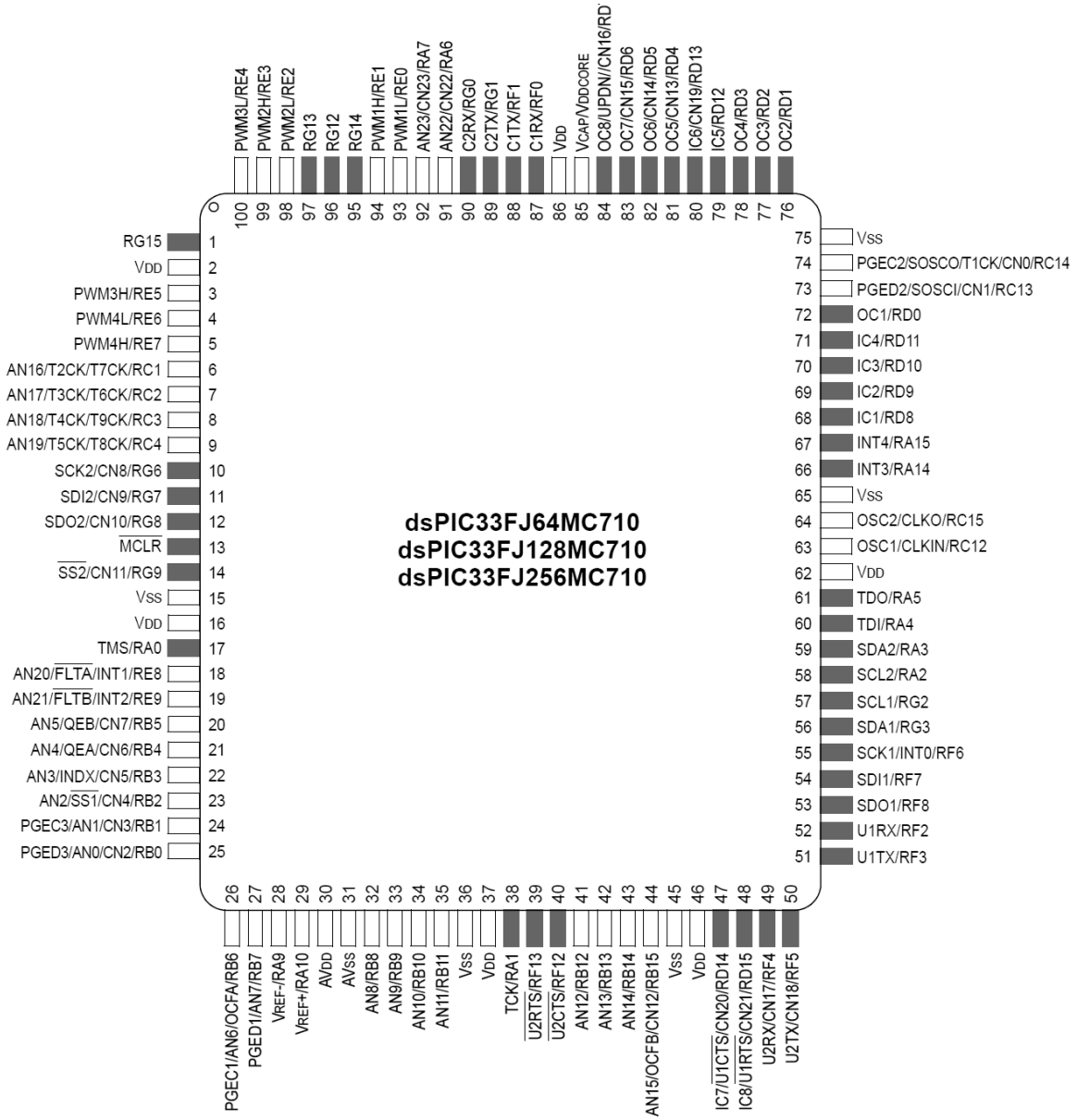


Figure 1: dsPIC33f Microprocessor

3.1.1 Data Types

When programming for a microcontroller with limited memory and CPU speed, it is often beneficial to carefully choose the size and signedness of integer data types. Use of `int`, `long`, `char`, etc. can be troublesome since the size and/or signedness of those types is machine dependent.

Using the smallest integer required for a particular task has two payoffs. First, it obviously takes up less memory. Second, the dsPIC33F microcontroller can only operate on 16-bit values in hardware. Operations (arithmetic, etc.) on larger values require additional assembly instructions, which take longer to execute.

Unless signed values are required for arithmetic, the use of unsigned integers is encouraged. This is because performing some bitwise operations on signed types can cause unexpected side effects. For example, when right shifting a negative value, sign extension will fill in 1s instead of 0s to maintain the result's negativeness.

The following list shows some useful data types defined in *labs/include/types.h*.

`uint8_t`	8-bit unsigned integer
`uint16_t`	16-bit unsigned integer
`uint32_t`	32-bit unsigned integer
`int8_t`	8-bit signed integer
`int16_t`	16-bit signed integer
`int32_t`	32-bit signed integer
`float`	32-bit floating point value
`double`[1]	32-bit floating point value
`long double`	64-bit floating point value

When declaring a global variable that will be accessed within an interrupt service routine, the `volatile` keyword should be used. For example, `volatile uint8_t var` rather than `uint8_t var`. This tells the compiler not to make any assumptions about the variable's value.

3.1.2 Specifying Hexadecimal Constants

In addition to specifying numeric constants in decimal notation, the C compiler allows hexadecimal values to be provided. This is done by prefixing the value with `0x`. For example, `0x8f` is equivalent to `143` (decimal) and 10001111 (binary). This feature is very useful because it is trivial to convert hex values to binary and vice-versa.

3.1.3 Bitwise Operations

To effectively discuss bitwise manipulations, there must be a clear way to identify each bit within a value. In CS 431, the following convention will be used. An n-bit value will have bits numbered consecutively from 0 through $n-1$. Bit 0 is the least significant bit in the value. Bit $n-1$ is the most significant bit.

[1] On the dsPIC architecture, `double` is equivalent to `long double` if `-fno-short-double` is used.

Operator		Example
`~`	Bitwise inverse (one's compliment)	`~`00101111 ⇒ 11010000
`&`	Bitwise AND	01010101 `&` 11110000 ⇒ 01010000
`|`	Bitwise OR	01010101 `|` 11110000 ⇒ 11110101
`^`	Bitwise XOR (exclusive OR)	01010101 `^` 11110000 ⇒ 10100101
`<<`	Left shift	00101111 `<<` 2 ⇒ 10111100
`>>`	Right shift	00101111 `>>` 2 ⇒ 00001011

Table 1: Bitwise operators available in C

Bitwise operations are used frequently when programming a microcontroller. Table 1 shows C bitwise operators that can be used on all integer types. Do not confuse bitwise AND and OR with the logical operators `&&` and `||`.

All of these operators can be used just like the arithmetic operators `+`, `-`, `*`, and `/`. For example, all of the following are legal.

```
uint8_t x, y;    // x and y are 8-bit unsigned integers

x = y << 2;      // Shift the value of y by 2 bits to the left
                 //and store the result in x
x = x | 0x01;    // Set the least significant bit in x to 1
x |= 0x01;       // Set the least significant bit in x to 1
x &= 0xfe;       // Set the least significant bit in x to 0
x &= ~0x01;      // Set the least significant bit in x to 0
x ^= 0x01;       // Toggle the least significant bit in x
```

Bits and Masks: When writing code for dspic33f, the `BV(i)` macro can be used to generate a mask that consists of all 0s, except for bit `i` which is 1. Mathematically, `BV(i)` is equivalent to 2^i. Table 2 shows the result of `BV(i)` for the first eight values of `i`. Note that since dsPIC33F is a 16-bit processor, you can have up to `BV(16)`.

Using `BV(i)` can make manipulating bits easier and less error prone. The following examples show how to perform some common bit operations.

```
uint8_t x, y;                    // x and y are 8-bit unsigned integers

x |= BV(4);                      // Set bit 4 in x to 1
x &= ~BV(4);                     // Set bit 4 in x to 0
x ^= BV(4);                      // Toggle bit 4 in x

x |= BV(2) | BV(4) | BV(5);      // Set bits 2, 4, and 5 in x to 1
x &= ~(BV(2) | BV(4) | BV(5));   // Set bits 2, 4, and 5 in x to 0
x ^= BV(2) | BV(4) | BV(5);      // Toggle bits 2, 4, and 5 in x
```

Macro	Binary	Hexadecimal	Decimal
`BV(0)`	00000001	0x01	$2^0 = 1$
`BV(1)`	00000010	0x02	$2^1 = 2$
`BV(2)`	00000100	0x04	$2^2 = 4$
`BV(3)`	00001000	0x08	$2^3 = 8$
`BV(4)`	00010000	0x10	$2^4 = 16$
`BV(5)`	00100000	0x20	$2^5 = 32$
`BV(6)`	01000000	0x40	$2^6 = 64$
`BV(7)`	10000000	0x80	$2^7 = 128$

Table 2: Evaluation of `BV(i)` for eight values of `i`

```
// The following two lines work together to "copy" bit 3 from y to bit 6 of x
x &= ~BV(6);                    // This line sets bit 6 in x to 0
x |= (y & BV(3)) >> 3 << 6;     // This line picks out bit 3 from y,
                                // "moves" it to bit 6 with two shifts,
                                // and then ORs the result into x
```

Direct bit access: Beside bitwise operations, C language also provides a method for defining and accessing fields directly. Using this method, bitwise operators are not needed as bit members can be accessed the same as struct members. An example using a struct follows:

```
struct reg {
    unsigned int bit0   : 1;
    unsigned int bit1_4 : 4;
    unsigned int bit5_7 : 3;
};
```

You can have direct access to different bits of a varibale as follows:

```
struct reg regA;

regA.bit0=1;
regA.bit1_4=16;
```

Two macros `CLEARBIT(BIT)` and `SETBIT(BIT)` provided in *labs/include/types.h* can be used to set the value of `BIT` to 0 and 1, respectively.

3.1.4 Microcontroller Registers

The dsPIC33f microcontroller's many features are controlled through the use of 16-bit I/O registers. These registers are memory mapped into the data address space. A full list of available I/O registers can be found in the dsPIC data sheet (**dsPIC-datasheet.pdf**).

Each register can be read and written in C as though it were an 16-bit variable. In most cases, `p33Fxxxx.h` has macros defining each I/O register and each set of bits in a I/O register by name. Each register macro has the format of MODnREGm and each set of bits in a register has the format of MODnREGmbits.FUNC where:

- MODn refers to the module MOD number n - eg. AD1 refers to ADC 1, T3 refers to Timer 3.
- REGm refers to the register REG number m of module MOD - eg. AD1CON1 refers to Control register 1 of ADC 1.
- FUNC refers to a set of function bits in the register - eg. AD1CON1bits.ADON refers to the Operating Mode bit in Control register 1 of ADC 1.

.

The following code will turn ADC 1 on:

```
#include <p33Fxxxx.h>

// Set the ADON bit in the AD1CON1 register to 1
SETBIT(AD1CON1bits.ADON);
```

3.2 Inputs/Outputs

All of the device pins (except `VDD`, `VSS`, `MCLR` and `OSC1/CLKIN`) are shared between the peripherals and the parallel I/O ports. All I/O input ports feature Schmitt Trigger inputs for improved noise immunity. There are 7 I/O ports denoted by letters from A to G

The general purpose I/O ports allow the dsPIC33F to monitor and control other devices. Most I/O pins are multiplexed with alternate function(s). The multiplexing will depend on the peripheral features on the device variant. In general, when a peripheral is functioning, that pin may not be used as a general purpose I/O pin.

All port pins have three main registers directly associated with their operation as digital I/O: `TRIS`, `PORT`, `LAT`. The data direction register `TRIS` determines whether the pin is an input or an output. If the data direction bit is a 1, then the pin is an input. All port pins are defined as inputs after a Reset.

Data on an I/O pin is accessed via a `PORT` register. A read of the `PORT` register reads the value of the I/O pin, while a write to the `PORT` register writes the value to the port.

The `LAT` registers are not important in CS431 labs. If you are interested in knowing more about them, please read the dspic manual.

Note that **p33Fxxxx.h** has macros representing bits, which belong to each I/O pin, on the above-mentioned registers. For example, to access the TRIS and PORT bits of I/O pin 6 on port D, we can use `TRISDbits.TRISD6` and `PORTDbits.PORTD6`, respectively.

The following example code sets up the I/O pin `D6` to be an output pin and writes 1 to its.

```
CLEARBIT(TRISDbits.TRISD6);
SETBIT(PORTDbits.PORTD6);
```

3.3 Interrupts

The following steps must be taken in order to enable an interrupt (see Chapter 7 of dsPIC-datasheet.pdf for more details).

1. Select the user-assigned priority level for the interrupt source by writing the control bits in the appropriate IPCx Control register. There are 8 priority levels with 7 to be the highest one. The CPU will handle the interrupt with the highest priority among all pending interrupts first. The interrupts with the same priority level will be handed in the order of their interrupt vector numbers. In the CS431 labs, you can set every interrupts to priority level 1 (interrupts with level 0 will be disabled).
2. Clear the interrupt flag status bit associated with the peripheral in the associated IFSx Status register.
3. Enable the interrupt source by setting the interrupt enable control bit associated with the source in the appropriate IECx Control register.

Note that in the CS431 labs you do not have to work with the nested interrupts and the associated control bits can be left at the default values.

The following example shows code to enable interrupt of Timer 1.

```
IPC0bits.T1IP = 0x01; // Set Timer1 Interrupt Priority Level
IFS0bits.T1IF = 0;     // Clear Timer1 Interrupt Flag
IEC0bits.T1IE = 1;     // Enable Timer1 interrupt
```

The following section will describe how to connect an interrupt service routine with an interrupt.

3.3.1 Interrupt Service Routines

dsPIC33F C library provides the following macro to define ISR functions:

```
void __attribute__((__interrupt__)) PrimaryName(void)
```

where `PrimaryName` is the predefined name of the interrupt you want the ISR to connect with. The following table shows all of the interrupts, which will be used (or may be used) in the CS431 labs, and their associated `PrimaryName`[2].

[2]See Section 8 of dsPIC-C-compiler.pdf for more details

Interrupt Number	Primary Name	Function
2	_OC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	TMR1 Timer 1 expired
6	_OC2Interrupt	OC2 Output compare 2
7	_T2Interrupt	TMR2 Timer 2 expired
8	_T3Interrupt	TMR3 Timer 3 expired
11	_U1RXInterrupt	UART1RX Uart 1 Receiver
12	_U1TXInterrupt	UART1TX Uart 1 Transmitter
13	_ADC1Interrupt	ADC 1 convert completed
20	_INT1Interrupt	INT1 External interrupt 1
21	_ADC2Interrupt	ADC 2 convert completed
30	_U2RXInterrupt	UART2RX Uart 2 Receiver
31	_U2TXInterrupt	UART2TX Uart 2 Transmitter

Note that in dsPIC33F, you have clear the interrupt flag inside its ISR. Following is an example of a simple interrupt service routine.

```
// Interrupt Service Routine triggered when Timer1's counter matches the timer period.
// Interrupts will be globally disabled during the ISR's execution.
void __attribute__((__interrupt__)) _T1Interrupt(void)
{
        global_counter++;         // Increment a global counter
        IFS0bits.T1IF = 0;        // clear the interrupt flag
}
```

3.4 Timers/Counters

The dsPIC33F device family offers several 16-bit Timer modules. With certain exceptions, all 16-bit timers have the same functional circuitry, and are classified into the following three types according to their functional differences:

- Type A timer (Timer1)
- Type B timer (Timer2, Timer4, Timer6 and Timer8)
- Type C timer (Timer3, Timer5, Timer7 and Timer9)

The Type A timer has access to an external 32kHz clock. The Type B and Type C timers can be combined to form a 32-bit timer. Each Timer module is a 16-bit timer/counter consisting of the following readable/writable registers:

- TMRx: 16-bit Timer Count register
- PRx: 16-bit Timer Period register associated with the timer
- TxCON: 16-bit Timer Control register associated with the timer

Each Timer module also has these associated bits for interrupt control:

- Interrupt Enable Control bit (TxIE)
- Interrupt Flag Status bit (TxIF)
- Interrupt Priority Control bits (TxIP<2:0>)

The Timer module can operate in 4 different modes, in which we only use Timer mode. In Timer mode, the input clock to the timer is derived from the internal clock ($F_{CY} = 12.8$Mhz), divided by a programmable prescaler (`TxCONbits.TCKPS`). When the timer is enabled, `TMRx` increments by one on every rising edge of the input clock and the timer generates an interrupt when `TMRx` matches the period set on `PRx`. After a period match, `TMRx` is automatically reset to 0.

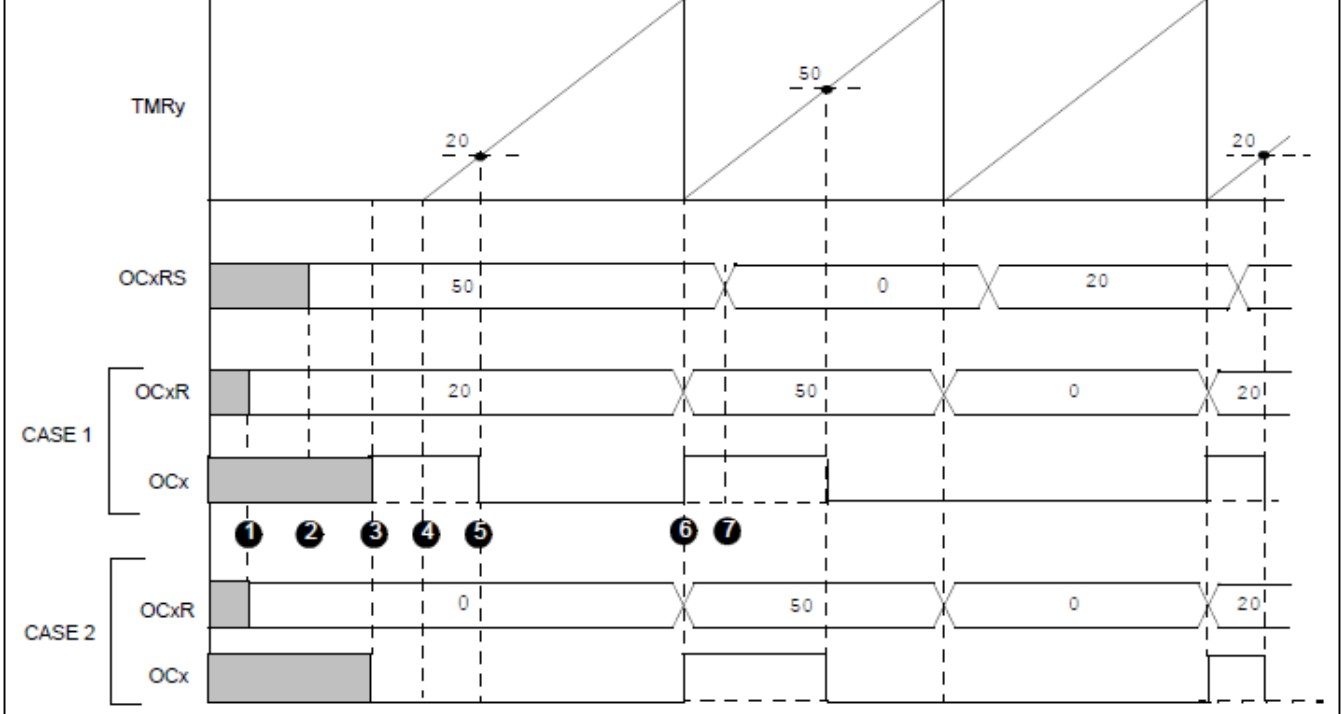
The following code setup Timer 1 to raise an interrupt every 5ms (note that the system clock operates at 12.8Mhz).

```
CLEARBIT(T1CONbits.TON);      // Disable Timer
CLEARBIT(T1CONbits.TCS);      // Select internal instruction cycle clock
CLEARBIT(T1CONbits.TGATE);    // Disable Gated Timer mode
TMR1 = 0x00;            // Clear timer register
T1CONbits.TCKPS = 0b10; // Select 1:64 Prescaler
PR1 = 1000;             // Load the period value
IPC0bits.T1IP = 0x01;   // Set Timer1 Interrupt Priority Level
CLEARBIT(IFS0bits.T1IF);      // Clear Timer1 Interrupt Flag
SETBIT(IEC0bits.T1IE);      // Enable Timer1 interrupt
SETBIT(T1CONbits.TON);      // Start Timer
```

For detailed description of Timers see Section 11 of the dsPIC33F manual.

Timer1 has access to a 32kHz crystal that is helpful in timing longer intervals (>ms). To use the 32kHz crystal you must use the macro below to enable it, then set Timer1 to use an external clock. This sample code will trigger an interrupt every 1 second.

```
//enable LPOSCEN
__builtin_write_OSCCONL(OSCCONL | 2);
T1CONbits.TON = 0; //Disable Timer
T1CONbits.TCS = 1; //Select external clock
T1CONbits.TSYNC = 0; //Disable Synchronization
T1CONbits.TCKPS = 0b00; //Select 1:1 Prescaler
TMR1 = 0x00; //Clear timer register
PR1 = 32767; //Load the period value
IPC0bits.T1IP = 0x01; // Set Timer1 Interrupt Priority Level
IFS0bits.T1IF = 0; // Clear Timer1 Interrupt Flag
IEC0bits.T1IE = 1;// Enable Timer1 interrupt
T1CONbits.TON = 1;// Start Timer
```



1. The duty cycle for first PWM cycle is written to OCxR register before enabling the PWM mode.
2. The duty cycle for second PWM cycle is written to OCxRS register.
3. PWM mode is enabled, OCx pin is driven low if OCxR is zero. OCx pin is driven high if OCxR is non-zero.
4. Timer is enabled and starts incrementing.
5. On a compare match, the OCx pin is driven low.
6. On timer rollover, OCxRS value is loaded into OCxR register. OCx pin is driven high if OCxR is zero. OCx pin is driven low if OCxR is non-zero.
7. The duty cycle for third PWM cycle is written to the OCxRS register.

Figure 2: Output Compare Operation: PWM mode

3.5 Output Compare

The module controls output pins (OCx) by comparing the value of a timer with the value of one or two compare registers depending on the operating mode selected. The state of the output pins change when the timer value matches the compare register value. The Output Compare module can select either Timer2 or Timer3 for its time base. The Output Compare module generates either a single output pulse or a sequence of output pulses, by changing the state of the output pin on the compare match events. Each Output Compare module consists of the following readable/writable registers:

- OCxR: 16-bit Output Compare register
- OCxRS: 16-bit Secondary Output Compare register
- OCxCON: 16-bit control register associated with the Output Compare

Output Compare Operations: The Output Compare has 7 operating modes of which "PWM mode without fault protection" will be used in CS431 labs. Figure 2 illustrates the output compare operation in this mode. It works as follows:

- When the PWM mode is enabled, the OCx pin is:
  - Driven high if the OCxR register value is non-zero (refer to Case 1 in Figure 2)
  - Driven low if the OCxR register value is zero (refer to Case 2 in Figure 2)
- When a selected timer is enabled, it starts incrementing until it reaches the value in the period register. The Compare Register (OCxR) value is constantly compared with the timer value. When a match occurs, the OCx pin is driven low.
- On a timer rollover, the OCxRS value is loaded into the OCxR register and the OCx pin is:
  - Driven high if the OCxR register value is non-zero
  - Driven low if the OCxR register value is zero

Ouput Compare setup: The following steps must be taken to configure the Output Compare module to control output pins OCx.

1. set the related timer with desired values
2. set OCx to be an output (see Section 3.2),
3. set the Output Compare module to a desired operating mode,

The following code sets up OC8 to work in PWM mode and be controlled by Timer 2. When operating, the OC8 pin will be set to high for 5ms every 40ms.

```
//setup Timer 2
CLEARBIT(T2CONbits.TON);     // Disable Timer
CLEARBIT(T2CONbits.TCS);    // Select internal instruction cycle clock
CLEARBIT(T2CONbits.TGATE);  // Disable Gated Timer mode
TMR2 = 0x00;                // Clear timer register
T2CONbits.TCKPS = 0b10;     // Select 1:64 Prescaler
CLEARBIT(IFS0bits.T2IF);    // Clear Timer2 interrupt status flag
CLEARBIT(IEC0bits.T2IE);    // Disable Timer2 interrupt enable control bit
PR2 = 8000;                 // Set timer period 40ms:
                            // 8000= 40*10^-3 * 12.8*10^6 * 1/64

//setup OC8
CLEARBIT(TRISDbits.TRISD7); /* Set OC8 as output */
OC8R = 1000;              /* Set the initial duty cycle to 5ms*/
OC8RS = 1000;             /* Load OCRS: next pwm duty cycle */
OC8CON = 0x0006;      /* Set OC8: PWM, no fault check, Timer2 */

SETBIT(T2CONbits.TON); /* Turn Timer 2 on */
```

PIN	PORT	ADC Single Ended Analog Input	Connection
RB4	B	ADC2CH4	Joystick's x-axis
RB5	B	ADC2CH5	Joystick's y-axis
RB15	B	ADC1CH15	Balance Board's x-axis
RB9	B	ADC1CH9	Balance Board's y-axis

Table 3: Microcontroller ADC analog input connections

For detailed description of the Output Compare module see Section 13 of the dsPIC33F manual. Other related sections are Section 11 and 10.

3.6 Analog to Digital Converter (ADC)

The dsPic microcontroller contains two analog to digital converters (ADC). They have an extremely flexible confirguration, we suggest that you review Section 16 Analog-to-Digital Converter (ADC) in the dsPic33f manual (DS70183A). Each converter can operate in 10-bit or 12-bit mode, the result can be an integer or floating point, and can measure voltages between 0 and 3.3 volts. 32 analog input pins (AN0-31) are connected to the ADC1 and 16 analog input pins (AN0-15) are conneced to ADC2, the joystick and balance board connections are as shown in Table 3. Notice that RBx (bit x of PORTB) and ANx (x analog input pin) share the same physical microcontroller PIN (as mentioned in the table). See picture of microcontroller (Figure 1) for a complete listing of dsPic pin overloading.

3.6.1 Initialization

The dsPic has four configuration registers, ADxCON1 - ADxCON4. Both ADC1 and ADC2 have their own set of configuration registers so below we use ADx to refer generically in commands. In your code it should read, for example AD1CON1 or AD2CON1 depending on the appropriate ADC you are configuring or reading. ADxCH123, and ADxCH0 configure what analog input pins and channels[3] are sampled; however, ADxCH123 is used for scanning (simultaneous) sampling, which is unnecessary for the labs. ADxSSH and ADxSSL are used in scanning sampling and are also therefore unnecessary. Each analog input pin can also function in digital mode so ADxPCFGL configures pins as analog or digital for AN0-15. AD1PCFGH configures AN16-31 as analog or digital; note since ADC2 has no access to AN16-31, there is NO AD2PCFGH register. Each analog input pin must also be set as an input in its appropriate TRISx register where x is the appropriate port (A, B, C, etc.).

Therefore the following steps must be taken to configure the ADC for use in the lab.

1. make sure the ADC is disabled by clearing the ADxCON1 ADON bit.

[3]The ADC units support four channels for simultaneous sampling of up to 4 different analog input pins. This feature is not used in CS431 Lab, so only channel 0 will be used. Channel 0 can be used both by ADC1 and ADC2.

2. Set the appropriate analog input pins to input using TRISx, where x is the appropriate port (A, B, C, etc.), and then set the appropriate analog input pins to analog using the ADxPCFGL for AN0-15 or ADxPCFGH for AN16-31.

3. set ADxCON1 to configure 10 or 12 bit Operation Mode, Data Output Format, and Sample Clock Source. We suggest using the appropriate bit mode, integer, and automatic conversion.

4. set ADxCON2 to configure scanning sampling. We suggest you set it to zero (i.e. no scanning sampling).

5. set ADxCON3 to configure the Conversion Clock Source, Auto Sample Time Bits, and Auto Conversion Clock Select. See following sample code and the ADC reference manual for proper lab values.

6. set ADxCON4 to configure Number of DMA Buffer Locations per Analog Input. This by default is set to one, so should not need to be modified for these labs.

7. Enable the ADC unit, using ADxCON1 by setting the ADDON bit.

The code below configures and enables ADC1, AN20 (PIN = RE8) to 10 bit, integer, automatic conversion mode, with no scanning.

```
//disable ADC
CLEARBIT(AD1CON1bits.ADON);
//initialize PIN
SETBIT(TRISEbits.TRISE8);        //set TRISE RE8 to input
CLEARBIT(AD1PCFGHbits.PCFG20);     //set AD1 AN20 input pin as analog
//Configure AD1CON1
CLEARBIT(AD1CON1bits.AD12B)  //set 10b Operation Mode
AD1CON1bits.FORM = 0;   //set integer output
AD1CON1bits.SSRC = 0x7; //set automatic conversion
//Configure AD1CON2
AD1CON2 = 0;  //not using scanning sampling
//Configure AD1CON3
CLEARBIT(AD1CON3bits.ADRC);    //internal clock source
AD1CON3bits.SAMC = 0x1F; //sample-to-conversion clock = 31Tad
AD1CON3bits.ADCS = 0x2;  //Tad = 3Tcy (Time cycles)
//Leave AD1CON4 at its default value
//enable ADC
SETBIT(AD1CON1bits.ADON);
```

3.6.2 Operation

The following steps must be taken to read the ADC.

1. set ADxCHS0 to the proper analog input pin.

PIN	PORT	Interrupt Registers	Connection
RF2	F	IEC0/IFS0	UART1 RX
RF3	F	IEC0/IFS0	UART1 TX
RF4	F	IEC1/IFS1	UART2 RX
RF5	F	IEC1/IFS1	UART2 TX

Table 4: Microcontroller UART connections

2. set the SAMP bit in the ADxCON1 to start a sample.
3. wait on DONE bit in the ADxCON1 to signal the sample and conversion are done.
4. clear DONE bit and retrieve the value from ADCxBUF0.

The code below will sample ADC1 AN20 with the above configuration:

```
AD1CHS0bits.CH0SA = 0x014;        //set ADC to Sample AN20 pin
SETBIT(AD1CON1bits.SAMP);        //start to sample
while(!AD1CON1bits.DONE);        //wait for conversion to finish
CLEARBIT(AD1CON1bits.DONE);        //MUST HAVE! clear conversion done bit
return ADC1BUF0;                        //return sample
```

3.7 UART

The dsPic has a two built-in USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) channels that can be used for serial communication. Both U1 and U2 have their own set of configuration registers so below we use Ux to refer generically in commands. In your code it should read, for example U1MODE or U2MODE depending on the appropriate UART you are configuring or reading. Each UART can be configured for interrupts as well using the IECy and IFSy registers, where y is 0 for U1 and 1 for U2. So for U1 you would use IEC0 and IFS0. UART1 is connected to the FlexUI LCD, and UART2 is connected to the FlexUI DB-9 RS-232 connector for communication with the lab PC. U1 and U2 connections are as shown in Table 4. See Section 17 UART in the dsPic33f manual (DS70188C)for detailed information.

3.7.1 Initialization

The following steps must be taken to configure the UART for use.

1. Stop the UART Port with register UxMODE UARTEN bit.
2. Disable and Clear UART Interrupts on correspondent ICE and IFS registers. The registers are ICE0 and IFS0 for UART1, and ICE1 and IFS1 for UART2.
3. Set I/O Pins in the TRISF register.

4. Configure UART speed by setting the baud rate in UxBRG register. Since CPU clock frequency is 12.8Mhz, the following formula is used to calculate the value of UxBRG: `UxBRG=800000 / baud -1`, where `baud` is the desired baudrate e.g. baud=115200. Note that, dsPIC33F allows UART to run at higher baudrates by setting UxMODEbits.BRGH to 1. However in this lab, this mode is not needed and UxMODEbits.BRGH should be set to 0.
5. Configure the UART mode using the UxMODE register.
6. Start the UART Port with register UxMODE UARTEN bit.
7. Configure Interrupts with the UxSTA register.

The following code intitializes UART1 to transmit and receive at the baudrate of 9600 with disabled interrupts.

```
/* Stop UART port */
CLEARBIT(U1MODEbits.UARTEN);  //Disable UART for configuration

/* Disable Interrupts */
IEC0bits.U1RXIE = 0;
IEC0bits.U1TXIE = 0;

/* Clear Interrupt flag bits */
IFS0bits.U1RXIF = 0;
IFS0bits.U1TXIF = 0;

/* Set IO pins */
TRISFbits.TRISF2 = 1;        //set as input UART1 RX pin
TRISFbits.TRISF3 = 0;        //set as output UART1 TX pin

/* baud rate */
// use the following equation to compute the proper
// setting for a specific baud rate
U1MODEbits.BRGH = 0;                    //Set low speed baud rate
U1BRG  = (uint32_t)800000 / 9600 -1;    //Set the baudrate to be at 9600

/* Operation settings and start port */
U1MODE = 0; // 8-bit, no parity and, 1 stop bit
U1MODEbits.RTSMD = 0;   //select simplex mode
U1MODEbits.UEN = 0;     //select simplex mode
U1MODE |= 0x00;

U1MODEbits.UARTEN = 1;  //enable UART
U1STA = 0;
U1STAbits.UTXEN = 1;    //enable UART TX
```

3.7.2 Transmitting

The following steps must be taken to send data through the UART. Note that you need to first enable transmitting mode in the UART module (see the previous section).

1. Wait on UTxBF bit in the UxSTA register.
2. Load the UxTXREG register with an 8 bit value.
3. Wait on TRMT bit in the UxSTA register.

The following code is used to transmit data on UART1.

```
while (U1STAbits.UTXBF);
U1TXREG = data;
while(!U1STAbits.TRMT);
```

3.7.3 Receiving

The heart of the receiver is the Receive (Serial) Shift (UxRSR) register. However, The UxRSR register is not mapped in data memory, so it is not available to the user application. After sampling the UxRX pin for the Stop bit, the received data in UxRSR is transferred to the receive FIFO (if it is empty). The UART receiver has a 4-deep, 9-bit wide FIFO receive data buffer. UxRXREG is a memory mapped register that provides access to the output of the FIFO. It is possible for four words of data to be received and transferred to the FIFO and a fifth word to begin shifting to the UxRSR register before a buffer overrun occurs.

UART RX can be enabled to operate in interrupt mode. Alternatively, the user can poll the UxSTAbits.URXDA bit to check the contents of the receive FIFO buffer. The following steps must be taken to read the receiving data from UART. Note that you first need to enable receiving mode in the UART module (see the previous section).

1. Check if there is data overflow error (by checking UxSTAbits.OERR). You must clear this bit if it has been set in order to receive new data.
2. Check if there is data in the buffer (by checking UxSTAbits.URXDA. If yes, then read the data from UxRXREG.

The following code read the data from the buffer of UART1.

```
if (U1STAbits.OERR) {
        U1STAbits.OERR = 0;
}

if (U1STAbits.URXDA) {
        *data = U1RXREG & 0x00FF;
}
```

4 Amazing Ball System (ABS)

The Amazing Ball System is a platform designed to give CS431 students experience programming an actual embedded system. The core of the system is a dsPIC33f microcontroller running at 12.8Mhz. The system consits of three main circuit boards in a stacked configuration: FLEX Light Base Board, FLEX Demo2 Daughter Board, and our custom designed FLEX-UI board. Each board extends the capability of the FLEX Light Base Board.

At the core of the board is a dsPIC33 microcontroller running at 12.8 MHz. The selected dsPIC33 is a modified Harvard archiecture with 256 Kbytes on-chip Flash and 30 Kbytes of Data SRAM. The dsPIC33 also has two Analog-to-Digital hardware modules (24 channels with either 10-bit or 12-bit resolution), nine 16-bit timers, and two UART modules. With the two additional circuit boards, the system also contains the following features.

4.1 Board Layout

As mentioned before the Amazing Ball System consits of three main circuit boards in a stacked configuration: FLEX Light Base Board, FLEX Demo2 Daughter Board, and the custom designed FLEX-UI board. Each board extends the capability of the FLEX Light Base Board.

The system run on 12V which is supplied to the FLEX-UI (with the main power switch) which then supplies 1A of current to itself and 1A to the FLEX Light Base board via the jumper wires in the front. Both of these two boards have over-current fuse protection and their own voltage regulators.

The FLEX Light Base board is mounted to the system's base plate with the servo motors and touch screen. It extends most of the I/O capability through two main headers CON5 and CON6 which is mounting point for the Demo2 Daughter board. Some of the I/O capability is then extended to the FLEX-UI board via the Demo2 CON8 and CON16 with ribbon cables.

4.2 FLEX Light Base Board

The dsPIC33 microcontroller and the power system for the FLEX Demo2 Daughter board are located on this board along with the RJ45 connection for the programmer.

4.3 FLEX Demo2 Daughter Board

The FLEX Demo2 Daughter Board primarily provides communication capability with a CAN tranceiver module, RS232 (UART) module and also has a power and control module for the touch screen sensors and servo motors.

4.4 FLEX UI Custom Board

The primary purpose of the FLEX-UI board is provide an nice user interface to the system with standard connections for power, joystick gameport and RS232-DE9 connection for serial communications. It also has the 5 LEDs, a dual-channel 12-bit DAC and 4 GPIO pins that can be used for future labs.

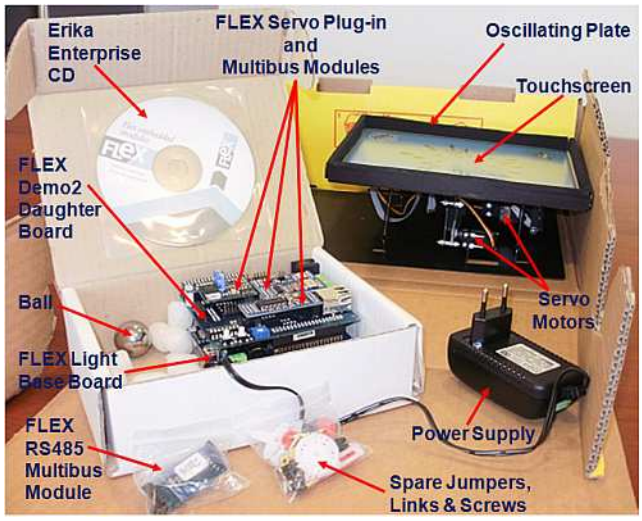


Figure 3: Amazing Ball System

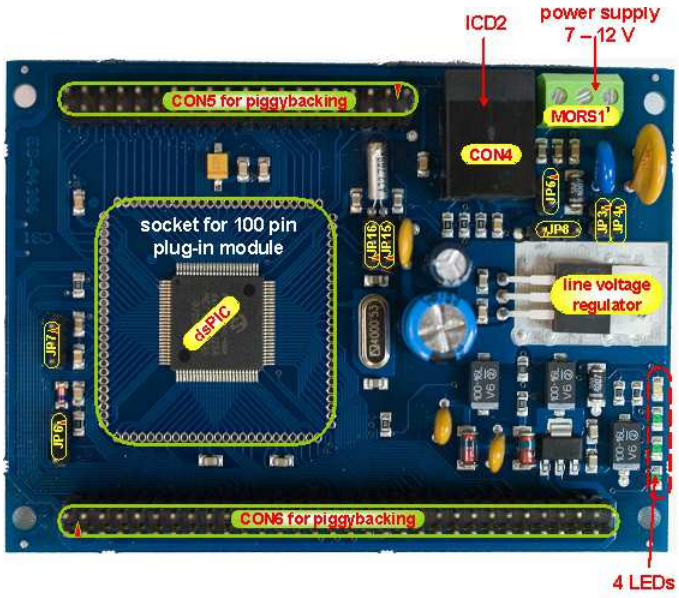


Figure 4: FLEX Light Base Board

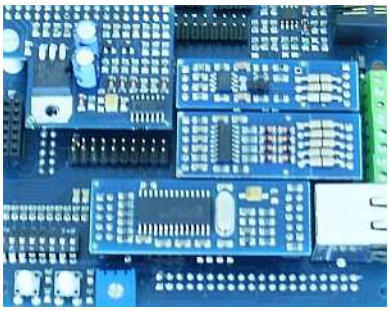


Figure 5: Demo2 Daughter Board

The LCD Screen (which uses UART1) is also mounted to the FLEX-UI board and is supplied with an on-board 6V voltage regulator. The LEDs and associated MOSFET transistors are powered from subsequent dual channel voltage regulator at 5V. The supporting analog Joystick hardware is supplied at 3.3V which is required for the dsPIC33 I/O pins (3.6V max).

The final item provided on the FLEX-UI board is the reset button located at the lower left corner. When pressed, this grounds the MCLR signal of the dsPIC33 and resets the microcontroller only. The associated LED is an indicator of the reset condition.

4.5 Light Emitting Diodes (LEDs)

There are five LEDs on the FLEX UI board that can be controlled by the microcontroller. These are connected to 5 MOSFET Transistors that are controlled by the digital input/output (I/O) port A: pins 0, 4, 5, 9, and 10. Primarily the function of these pins are controlled by the TRISA register (stands for tri-state) which determines whether each pin associated with the I/O port is an input or an output and the PORTA register which sets the value of the pin, 1 for HIGH and 0 for LOW. The following macros are defined in *led.h* to make controlling the LEDs easier.

```
// Tri-state registers
#define LEDTRIS                TRISA
#define LED1_TRIS         TRISAbits.TRISA4
#define LED2_TRIS         TRISAbits.TRISA5
#define LED3_TRIS         TRISAbits.TRISA9
#define LED4_TRIS         TRISAbits.TRISA10
#define LED5_TRIS         TRISAbits.TRISA0

// Port registers using predefined structs
#define LEDPORT                PORTA
#define LED1_PORT         PORTAbits.RA4
#define LED2_PORT         PORTAbits.RA5
#define LED3_PORT         PORTAbits.RA9
#define LED4_PORT         PORTAbits.RA10
```

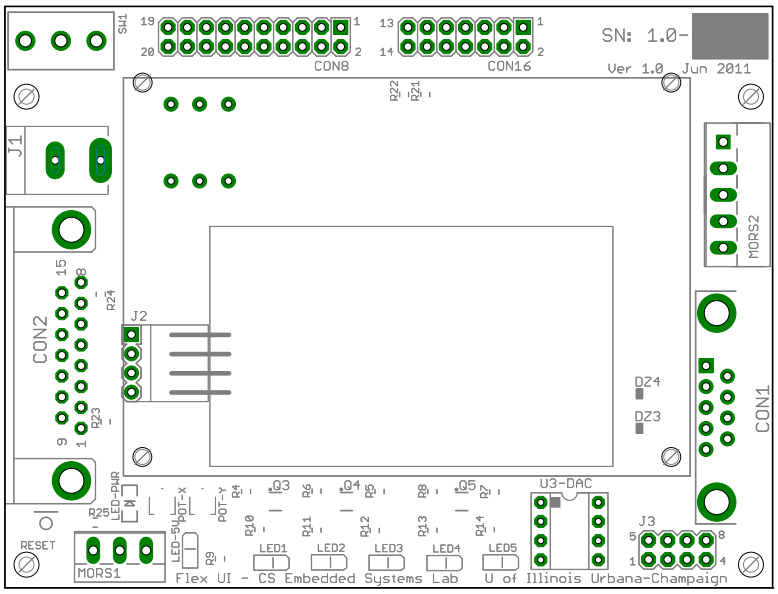


Figure 6: FLEX UI Custom Board

```
#define LED5_PORT        PORTAbits.RA0

// LEDPORT Bitwise definitions
#define LED1        4
#define LED2        5
#define LED3        9
#define LED4        10
#define LED5        0
```

If the TRIS bit for an I/O pin is a '0', then the pin is configured for an output. Hence, if pin 4 of TRISA register is set to 0 and pin 4 of PORTA register is set to 1, then LED1 will light.

Special care must be taken with dsPIC33F I/O ports; in fact, one instruction cycle is required between a port direction change or port write operation and a read operation of the same I/O port, so you may need to insert a Nop() between consecutive manipulations of PORTA. Therefore the code below is incorrect.

```
CLEARBIT(LED1_TRIS);  //set Pin to Output
CLEARBIT(LED2_TRIS);  //set Pin to Output
SETBIT(LED1_TRIS);  //Turn on LED1
SETBIT(LED2_TRIS);  //Turn on LED2
```

To correct the code typically an instruction would be inserted, such as the NOP or Nop() macro function call (alias to an assembly level instruction). See the dsPIC33F datasheet section 11.4 for

more details. The *led.h* file has included some macros that account for the NOP. As a result, the following code examples can be used.

```
CLEARLED(LED1_TRIS); // set Pin to Output
CLEARLED(LED2_TRIS); // set Pin to Output
CLEARLED(LED3_TRIS); // set Pin to Output
CLEARLED(LED4_TRIS); // set Pin to Output
CLEARLED(LED5_TRIS); // set Pin to Output

/* Using macros defined in \path{labs/include/types.h} */
SETLED(LED1_PORT);     // Turn on LED1
CLEARLED(LED1_PORT);   // Turn off LED1
TOGGLELED(LED1_PORT);  // Toggle LED1(BIT ^= 1)
```

If you want to use these macro functions as part of a conditional statement, you must ensure that you also surround them with brackets since the macro is a multiline definition (see the below example):

```
if (n == 5)
   {SETLED(LED5_PORT);} // correct use of brackets
else if (n == 3)
   SETLED(LED1_PORT); // incorrect\, needs brackets and will not compile
else
   CLEARLED(LED1_PORT); // compiles, but sematically incorrect
```

Alternately you can also control the LEDs using bitwise operations (make sure to use NOP instructions where appropriate):

```
/* Setting individual pins */
LED2_PORT = 1;         // Turn on LED2
Nop();
LED2_PORT = 0;         // Turn off LED2
Nop();
LED2_PORT ^= 1;        // Toggle LED2
//Nop();
//uncomment Nop() if performing additonal PORTA (LED) manipulation
```

You can also control the LEDs using following bitwise operations:

```
/* Setting pins using bitwise operations*/
LEDPORT ^= 1<<LED1; Nop();  // Toggle LED1 by setting 1 and shifting to bit postion
LEDPORT |= (1<<LED2) | (1<<LED3); Nop();    // Turn on LED2 and LED3
LEDPORT &= ~((1<<LED3) | (1<<LED4));        // Turn off LED3 and LED4
//Nop();
//uncomment Nop() if performing additonal LEDPORT manipulation
```

4.6 Joystick

4.6.1 Joystick Analog Axes

See section 3.6 for configuration of analog sampling of the X and Y axis. Due to the construction of the joysticks and the FlexUI, the voltage range seen on each analog axis will differ. If an accurate position is required, obtain calibration data (such as the x-axis and y-axis voltages at the upper left and lower right corners) and scale future samples accordingly.

4.6.2 Joystick Buttons

One standard game port joystick may be connected to the Flex UI board at once. The joystick has two active buttons. The trigger is button #1 and one of the top thumb buttons is #2. These pins are normally held at a high voltage by an external pull-up resistor. When a button is pressed, its pin gets shorted to ground. Therefore, if bit PORTDbits.RD10 is a 1, the joystick's button #2 is not pressed, and if bit PORTEbits.RE8 is a 0, then the joystick's button #1 is pressed.

The following Registers are necessary to configure the buttons.

```
AD1PCFGHbits.PCFG20  //ADC1 Port Configuration Register High Channel 20
TRISEbits.TRISE8  //PortE Pin 8 I/O Configuration
PORTEbits.RE8  //PortE Pin 8 value
IEC1bits.INT1IE  //Interrupt Enable Control Register 1
                 //External Interrupt 1 Enable bit
IPC5bits.INT1IP  //Interrupt Priority Control Register 5
                 //External Interrupt 1 Priority bits
IFS1bits.INT1IF  //Interrupt Flag Status Register 1
INTCON2bits.INT1EP  /*Interrupt Control Register 2 External
                      Interrupt 1 Edge Detect Polarity Select bit*/
TRISDbits.TRISD10  //PortD Pin 10 I/O Configuration
PORTDbits.RD10  //PortD Pin 10 value
```

Button #2 is located on PORTD PIN10 configure it as you would a standard I/O pin as described in section 3.2. The state can be read from PORTDbits.RD10.

Button #1 is located on PORTE PIN8 and is shared with ADC1CH20 so AD1PCFGHbits.PCFG20 must be set to '1' to place it in digital mode. Set TRISEbits.TRISE8 to 1 to enable the reading of button#1 state from PORTEbits.RE8. In addition to reading the buttons as described above, button #1 the joystick trigger is connected to a pin which can be set up to generate external interrupt requests. See section 3.3 for more information on how to enable and configure the interrupt. A special note that since the button is active high set INTCON2bits.INT1EP to '1' for falling edge triggering of the interrupt.

As a result, code such as the following examples can be used.

```
if (PORTEbits.RE8 == 0)
{
        // The statement is true when joystick A's button #1 (trigger) is pressed.
```

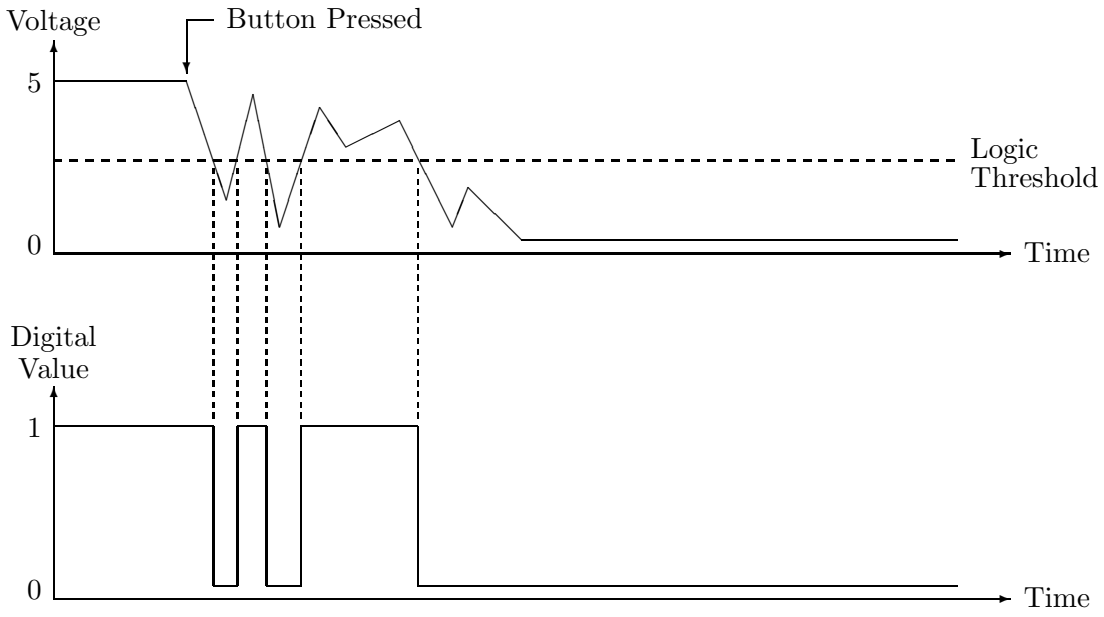


Figure 7: Mechanical switch contact bounce

```
}

// Spin until joystick B's button #1 (trigger) is pressed.
while (PORTEbits.RE8 == 1);

// Set x equal to 0x01 if joystick A's button #2 (thumb) is pressed
//and 0x00 otherwise.
uint8_t x = ~(PORTDbits.RD10);

// The first line here turns off LED2.  The second line then turns on
// LED2 only if joystick A's button #1 (trigger) is currently pressed.
LEDPORT &= ~(1<<LED2);
LEDPORT |= ~(PORTEbits.RE8) << LED2;
```

4.6.3 Button Debouncing

Mechanical switches, such as the joystick buttons, can suffer from an issue known as contact bounce. The problem results because, immediately after the button is pressed or released, the voltage across the switch can fluctuate for a brief period of time. As illustrated in Figure 7, these changes in voltage often cross the logic threshold, thereby causing the digital input bit to oscillate between 1 and 0. The bouncing only lasts for several milliseconds, but this is long enough to be noticeable on the microcontroller.

The act of overcoming the contact bounce problem is called debouncing. One approach is to implement a low pass filter in hardware (capacitors and resistors or other). Although all I/O input ports of the dsPIC33F feature Schmitt Trigger inputs for improved noise immunity and eliminate high frequency bouncing (essentially forcing the noisy signal to logic HIGH or logic LOW) they do not account for the oscillating digital results. So debouncing (or filtering) must be done in software. A simple algorithm is to check the digital value periodically. Only when the previous T consecutive samples are 0 is the button considered pressed. Similarly, only when the previous T consecutive samples are 1 is the button considered released. An appropriate value for the threshold T can be determined experimentally.

4.7 LCD module

The LCD module used on the FlexUI board contains 128x64 pixels. When used for text, the display can hold 8 rows of 21 ASCII characters each. Low level communication with controller chips on the LCD module is not complicated, but some basic code to perform primary functions is provided in lcd.c. Students interested in advanced features, or low level LCD details should review lcd.h, the lcd.c code, as well as the LCD-09351 data sheets.

The LCD is connected to UART1 on the dsPic for the purpose of connecting to the dsPic for serial communication. Using the LCD for text output is relatively straightforward. Include lcd.h call the function lcd intialize function.

The following functions may then be used to control the display. Additional functions are listed in lcd.h.

```
void lcd_initialize(); //Initialize the LCD UART connection.
void lcd_clear(); //Clear the entire LCD. (Turn off all pixels.)
void lcd_clear_row(uint8 row); //Clear the specified row on the LCD.
                               //Valid values for row are 0 through 7.
void lcd_locate(uint8 column, uint8 row); //Move the cursor to location
                                          //(column, row). Valid values
                                          //for row are 0 through 7.
                                          //Valid values for column are
                                          //0 through 20.
```

A note on the use of stdio.h. The firmware of the microcontroller that controls the LCD interprets the newline (n) character different than most terminal programs. It will be interpreted as a strict line feed, moving the cursor to the next row, but the same column. You should instead use a carriage return (r) which is will result in both a new line and a carriage return to the first column. Also the dsPic does not flush its UART buffer unless a newline (n) or carriage return (r) is present in a printf or with use of the fflush() command. There is an additional macro that will automatically perform the fflush for you. This method requires the use of the lcd locate command to move the cursor, which gives you greater control over where the cursor is.

```
void lcd_printf( const char * format, ... ); /*functions as printf
with an fflush() afterwards, use lcd_locate to go to a newline. */
```

the following statements are equivalent to each other

```
//using printf with fflush()
printf("Hello World!");                //print to buffer
fflush();                          //flush to LCD
lcd_locate(0,1);                  //carriage return

//using lcd_printf
lcd_printf("Hello World!");        //print to buffer and flush to LCD
lcd_locate(0,1);                  //carriage return

//using printf with newline
printf("Hello World! \n");         //print to buffer flush to LCD and newline
lcd_locate(0,1);                  //carriage return

//using lcd_printf with carriage return
lcd_printf("Hellow World! \r");     //print to buffer flush to LCD and carriage return
```

Read section 4.13 STDIO.H INPUT AND OUTPUT in the 16-BIT LANGUAGE TOOLS LIBRARIES (DS51456C) reference manual for the permitted printf() format strings. Note that the language tool libraries are 16 bit, and therefore printing values larger than 16 bits is not supported by the libraries.

Finally, note that characters written past the right edge of the display will wrap around to the following line. Characters written past the bottom line of the display will wrap around to the top line.

4.8 Touchscreen

The Amazing Ball System (ABS) has a four-wire touch screen. The touchscreen has two homogeneous resistive layers. The point of contact divides each layer in a series of resistor network with two resistors (see Figure 8). By measuring the voltage at this point the user gets information about the position of the contact point orthogonal to the voltage gradient. To get a complete set of coordinates, the voltage gradient must be applied once in vertical and then in horizontal direction: first a supply voltage must be applied to one layer and a measurement of the voltage across the other layer is performed, next the supply is instead connected to the other layer and the opposite layer voltage is measured. For example, to measure the X-coordinate, the user must apply Vdd to the bottom, GRN to the top and connect the left to high impedance (Hi-Z), and measure the signal from the right. Please refer to Table 5 for connections while measuring the coordinates.

In the ABS, the measuring pin of the touchscreen's X-coordinate is connected to ADC pin AN15 and that of touchscreen's Y-coordinate is connected to ADC pin AN9. There is also a circuit designed to control the touchscreen measurement. This circuit is controlled through three I/O pins E1, E2, E3. For example, setting E1=1, E2=1, E3=0 puts touchscreen into the standby mode.

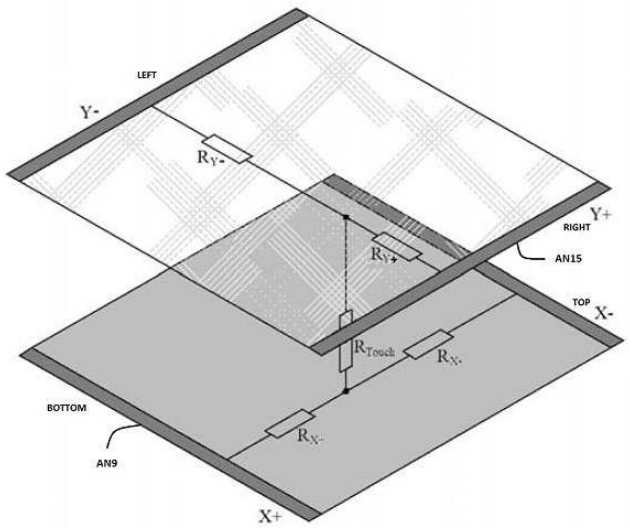


Figure 8: 4-wire Touchscreen

	Bottom (X+)	**Top (X-)**	**Right (Y+)**	**Left (Y-)**
Standby	Hi-Z	Hi-Z	Hi-Z	GND
Read X-coordinate	Vdd	GRN	Hi-Z (AN15)	Hi-Z
Read Y-coordinate	Hi-Z (AN9)	Hi-Z	Vdd	GND

Table 5: Connections while measuring the touchscreen's coordinates

Please refer to Table 6 for the details on how these pins work.

The following example code will enable the touchscreen X-coordinate pin to connect to the ADC

```
//set up the I/O pins E1, E2, E3 to be output pins
CLEARBIT(TRISEbits.TRISE1); //I/O pin set to output
CLEARBIT(TRISEbits.TRISE2); //I/O pin set to output
CLEARBIT(TRISEbits.TRISE3); //I/O pin set to output

//set up the I/O pins E1, E2, E3 so that the touchscreen X-coordinate pin
//connects to the ADC
CLEARBIT(PORTEbits.RE1);
SETBIT(PORTEbits.RE2);
SETBIT(PORTEbits.RE3);
```

Note that it takes about 10ms for the touchscreen output signal to be stable when the touchscreen is switched from one operation mode to another.

	Bottom (X+)	**Top (X-)**	**Right (Y+)**	**Left (Y-)**
E1=1	Hi-Z	Hi-Z	No effect	No effect
E1=0	Vdd	GRN	No effect	No effect
E2=1	No effect	No effect	Hi-Z	No effect
E2=0	No effect	No effect	Vdd	No effect
E3=1	No effect	No effect	No effect	Hi-Z
E3=0	No effect	No effect	No effect	GRN

Table 6: The effect of I/O pin E1, E2, and E3

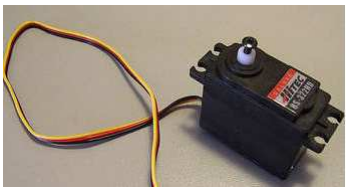


Figure 9: Servo

4.9 Servos

The Amazing Ball System (ABS) has two servos: one for X dimension and one for Y dimension. Each servo is controlled by three wires: ground (black), power (red), and command (yellow). Servos are commanded through "Pulse Width Modulation," or PWM, signals sent through the command wire. Essentially, the width of a pulse defines the position. For the servos in this lab, sending a 0.9ms pulse to the servo, tells the servo that the desired position is 0 degrees, while sending 1.5ms and 2.1ms pulse will render 90 degree and 180 degree positions, respectively (see Figure 10). In order for the servo to hold this position, the command must be sent at about 50Hz, or every 20ms. If command pulse is longer than 2.1ms or shorter than 0.9ms, the servo would attempt to overdrive (and possibly damage) itself.

In the ABS, the command wires of X and Y servo are connected to Output Compare pin OC8 and OC7, respectively, through servo controllers. Notice that the servo controllers **invert** the signal from OC8 and OC7, so when the Output Compare pin is 1 the servo command signal is 0 and vice-versa. As a consequence, to correctly drive the servo motors, the Output Compare module needs to generate an 'inverted' signal as shown in Figure 11 in order to output a servo command signal as shown in Figure 10. Finally, to control the servos, users will have to setup OC8 and OC7 to work in PWM mode. See Section 3.5 for references on how the Output Compare works.

4.10 Serial Ports

The FlexUI has a single DB-9 RS-232 port connected to UART2 on the dsPic for the purpose of connecting to the computer for serial communication. See the Section 3.7 for information on how

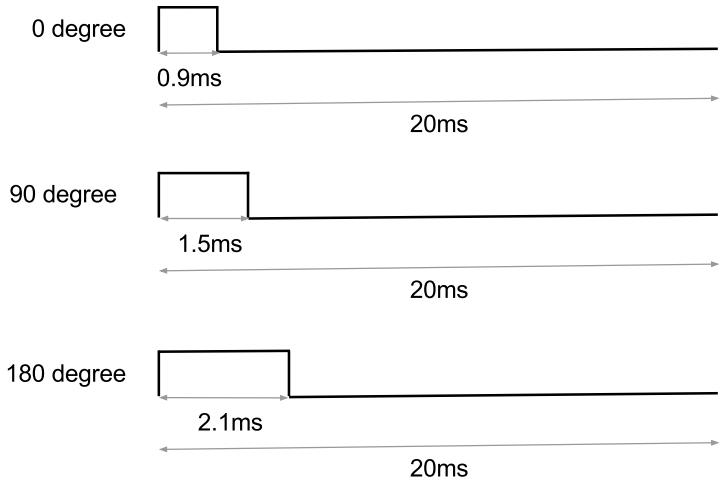


Figure 10: Servo Control Timing

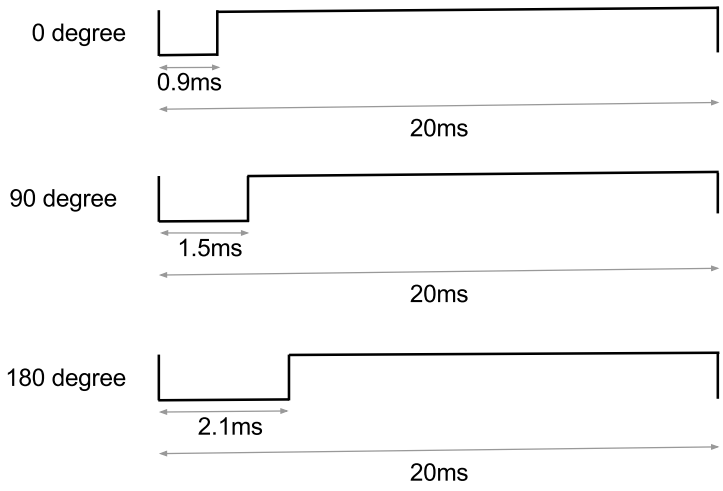


Figure 11: Output Compare Output Timing

Microcontroller	DAC	Description
PORTD bit 8	$\overline{\text{CS}}$	Chip Select
PORTB bit 11	SCK	Serial Clock Input
PORTB bit 10	SDI	Serial Data Input
PORTB bit 13	$\overline{\text{LDAC}}$	Latch DAC Input

Table 7: Digital to analog converter chip connections to the microcontroller

FLEX UI Jumper J3	MCP4822 DAC Functions
Pin 5	VOUTA
Pin 6	GND
Pin 7	VOUTB
Pin 8	GND

Table 8: MCP4822 DAC Output Pin Functions

to configure the UART on the dsPic.

4.11 Digital to Analog Converter (DAC)

The FLEX UI board contains a Microchip MCP4822 dual-voltage output digital to analog converter chip (labeled U3). These chips have 12 bits of resolution with the ideal output range of (a) 0.0V to 2.048V when gain setting = 1x; or (b) 0.0V to 4.096V when gain setting = 2x. The MCP4822 is controlled through a Serial Peripheral Interface (SPI). Table 7 shows how the DAC is wired to the microcontroller.

Several of I/O pins connected to the DAC are also internally connected to the microcontroller's ADC modules (ADC1 and ADC2) which is denoted by ANx. This includes SDI (RB10/AN10), SCK (RB11/AN11) and $\overline{\text{LDAC}}$ (RB13/AN13). In order to use these pins as digital I/O, these must be configured as digital for both ADC registers by setting the ADxPCFGL registers for both ADC modules. The following example shows code to configure the SCK pin:

```
#define DSCK_AD1        AD1PCFGLbits.PCFG11
#define DSCK_AD2        AD2PCFGLbits.PCFG11
#define DSCK_TRIS        TRISBbits.TRISB11

SETBIT(DSCK_AD1); // set Pin to Digital
SETBIT(DSCK_AD2); // set Pin to Digital
CLEARBIT(DSCK_TRIS); // set Pin to Output
```

The output of each DAC channel is connected to the J3 header on the right side of the FLEX UI board, pins 5 and 7 (see table). An oscilloscope or volt meter may be connected to these posts to verify the output voltage. For more wiring details see the FLEX UI schematic.

FLEX UI Jumper J3	Microcontroller Functions
Pin 1	RB3/AN3/INDX/CN5
Pin 2	RG9/CN11
Pin 3	RA1/TCK
Pin 4	RB12/AN12

Table 9: GPIO connections to the microcontroller

Figure 12: Jumper J3 of the Flex UI board (lower right corner of board)

4.11.1 Concept of Operation

Commands and data are sent to the device via the SDI pin, with data being clocked-in on the rising edge of SCK. The communications are unidirectional and, thus, data cannot be read out of the MCP4822 devices. The CS pin must be held low for the duration of a write command. The write command consists of 16 bits and is used to configure the DACs control and data latches. Bit 15 of the command specifes the DAC channel and which input latch register to load, DACA and DACB.

Once the desire input latch register has been programmed, the $\overline{\text{LDAC}}$ (latch DAC synchronization input) pin is used to transfer its value to the corresponding DAC registers (output latches, VOUT). When this pin is low, both VOUTA and VOUTB are updated at the same time with their input register contents.

Section 5 of the MCP4822 data sheet provides more detailed instructions and timing information for operating the chips.

4.12 General Purpose I/O (GPIO)

The FLEX UI board extends four General Purpose In/Out (GPIO) pins to jumper J3. Table 9 shows which pins are connected to the microcontroller. For more details on using the GPIO pins, see the dsPIC33F datasheet

4.12.1 I/O Port Write/Read Timing

One instruction cycle is required between a port direction change or port write operation and a read operation of the same port. Typically, this instruction would be a NOP or `Nop()` macro function call (alias to an assembly level instruction).

5 Microchip MPLab IDE

In the CS431 labs, we use MPLab Integrated Developement Environment (IDE) to write code and program the dsPIC33F processor. **MPLab-IDE.pdf** has a full manual of the IDE. You only need to read Chapter 2 of this manual. Most of the steps need to be taken to create a project has been done for you in the provided project template. The project template can be found in *cs431/labs/template/*.

When using MPLab, you should only use the Debug mode when it is absolutely necessary, e.g when you want to see the content of registers etc.. Most of the time you should use the Release mode and use LCD and LEDs for debugging. The Release mode runs much faster than the Debug mode.

6 Linux

Linux programming will be used to explore the basics of signals, timers, inter-process and intra-process communication. Interfacing with the PCI-DAS1602/12 ADC/DAC card will also be performed.

6.1 Reference Documents

When coding for Linux, most documentation will come in the form of man pages. These pages are available for virtually all library functions and system calls. They describe each function's arguments, operation, return value, and any required header files.

To get the man page on a C function called *name*, execute: `man` *name*

Occasionally an executable program or shell command will have the same name as a C function (`printf`, for example). In this case, tell `man` to only search sections 2, 3, and 3p by executing:
`man 2 3 3p printf`

Once in a man page, the arrow keys scroll, `/`*blah* searches for the text *blah*, `n` repeats the last search, and `q` quits.

Details on the PCI-DAS1602/12 Analog & Digital I/O Board can be found in the following manual: *pci-das1602-12_register_map.pdf*
http://www.mccdaq.com/registermaps/RegMapPCI-DAS1602-12.pdf

6.2 Return Value Checking

When invoking a library function or system call, it is good practice to check the return value for errors. Many functions return 0 upon success and -1 to indicate an error. However, this is not always the case. When in doubt, check the man page for a RETURN VALUE section near the end.

If an error has occurred, `perror()` can be used to print out a description of the `errno` value. Examples of checking the return value and using `perror()` will be provided throughout the remainder of section 6.

6.3 Pointers and Memory Allocation

Several library functions and system calls used in this class take a pointer to a structure as an argument. In these situations, it is vital to ensure that the passed pointer is valid and points to correctly allocated memory.

For example, consider the following `increment()` function which takes an int pointer and increments the integer that it points to.

```
void increment(int *pointer)
{
        (*pointer)++;
}
```

The following code shows how *not* to use `increment()`.

```
// Declare an int pointer.
int *my_int_pointer;

// Pass the int pointer to increment().  This is INCORRECT because the
// pointer is current uninitialized and could be pointing anywhere in memory!
// A segmentation fault or otherwise unexpected behavior will result.
increment(my_int_pointer);
```

There are two ways to correct this. The easiest is to allocate an int on the stack and pass its address to `increment()` as seen here.

```
// Declare an int on the stack.
int my_int;

// Pass the address of the int to increment().
increment(&my_int);
```

Alternatively, dynamic memory may be allocated for the int with `malloc()` as seen here.

```
// Declare an int pointer.
int *my_int_pointer;

// Allocate enough dynamic memory for an int and set the pointer equal to the
// start of that memory.
my_int_pointer = malloc(sizeof(int));

// Pass the int pointer to increment().  This is correct because the pointer now
// points to correctly allocated memory.
increment(my_int_pointer);
```

Signal	Description	Default Action
`SIGINT`	Interrupt (CTRL+C)	Kill the process
`SIGALRM`	Timer alarm (see section 6.5)	Print "Alarm clock" and kill the process
`SIGUSR1`	User defined signal #1	Print "User defined signal 1" and kill the process
`SIGUSR2`	User defined signal #2	Print "User defined signal 2" and kill the process

Table 10: Some Linux signals and associated default actions

```
// ... use my_int_pointer ...

// When done, free the dynamic memory and set the pointer to NULL to be safe.
free(my_int_pointer);
my_int_pointer = NULL;
```

6.4 Signals

In Linux, a signal is a message sent by the operating system to a process to notify the program of an event. For example, the `SIGINT` signal is sent to a process when CTRL+C has been pressed by the user. Signals can be trapped by a process in several ways. In this class, both signal handler functions and blocking will be used.

If a process is not configured to trap a particular signal and that signal arrives, then a default action will be performed. Table 10 shows some signals and their default actions.

6.4.1 Asynchronous Signal Handlers

One way to trap a signal is to associate it with a handler function. Whenever the signal is received, normal execution of the process will be suspended and the handler function will execute. When the handler returns, normal execution of the process resumes where it left off. This is analogous to the way hardware interrupts and interrupt service routines operate.

The association between a signal and a handler function is controlled by `sigaction()`. An example of how to use `sigaction()` is seen below. Check the man page for additional information.

```
// Define a signal handler function.  The function name can be anything, but
// it must accept a single integer.
void handler_function(int signum)
{
        printf("CTRL+C was pressed!  Exiting.\n");
        exit(0);
}

// This function associates the SIGINT signal with the handler function.
void setup_signal_handler()
```

```
{
        struct sigaction action;

        // Ensure that the entire structure is zeroed out.
        memset(&action, 0, sizeof(action));

        // Set the sa_handler member to point to the desired handler function.
        action.sa_handler = handler_function;

        // Call sigaction to change the action taken upon receipt of the SIGINT signal.
        if (sigaction(SIGINT, &action, NULL) != 0)
        {
                // If there is an error, print out a message and exit.
                perror("sigaction");
                exit(1);
        }
}
```

6.4.2 Blocking

Another way to detect when a signal has been received by a process is by blocking with `sigwait()`. `sigwait()` takes a set of signals and suspends until one of the signals in the set is received. To work reliably, the signal(s) being waited for must be blocked with `sigprocmask()`. The following example shows how to set this up.

```
int signal_received;
sigset_t signal_set;

// Empty the set, and then add one (or more) signals to it that sigwait() will wait for.
sigemptyset(&signal_set);
sigaddset(&signal_set, SIGALRM);

// Set the process signal mask to block the signals that sigwait() will wait for.
sigprocmask(SIG_BLOCK, &signal_set, NULL);

while (1)
{
        // sigwait() will suspend until one of the signals in signal_set is received.
        sigwait(&signal_set, &signal_received);

        // Do something useful here.  signal_received will be set to the signal to
        // that caused sigwait() to return.
}
```

6.5 Timers

In Linux, timers[4] can be used to send a signal to a process after a specified period of time has elapsed. Timers may be used in one of two modes: one-shot or periodic.

When a one-shot timer is set up, a *value* time is specified. When that time has elapsed, the operating system sends the process a signal and deletes the timer.

When a periodic timer is set up, both a *value* and an *interval* time is specified. When the *value* time has elapsed, the operating system sends the process a signal and reschedules the timer for *interval* time in the future. When the *interval* time has elapsed, the OS sends another signal and again reschedules the timer for *interval* time in the future. This will continue until the process manually deletes the timer.

By default a timer will send the `SIGALRM` signal. If multiple timers are used in one process, however, there is no way to determine which timer sent a particular `SIGALRM`. Therefore, an alternate signal, such as `SIGUSR1`, may be specified when the timer is created.

6.5.1 Resolution

Timers are maintained by the operating system, and they are only checked periodically. A timer that expires between checks will be signaled (and rescheduled if periodic) at the next check. As a result, a process may not receive signals at the exact time(s) that it requested. For example, suppose the OS checks the timers every 10 milliseconds and a process schedules a periodic timer with *value* = 5 milliseconds and *interval* = 21 milliseconds. Then the process will receive a signal after 10 milliseconds, and every 30 milliseconds thereafter.

The period at which the timers are checked, called the clock resolution, is operating system and hardware dependent. The actual value can be determined at runtime by calling `clock_getres()` on the system-wide real-time clock (`CLOCK_REALTIME`).

6.5.2 Operation

`create_timer()` is used to create a new timer. As with `clock_getres()`, the system-wide real-time clock (`CLOCK_REALTIME`) should be used. The following code shows how to create a timer that sends the default `SIGALRM` signal.

```
timer_t timer1;

// Create a new timer that will send the default SIGALRM signal.
if (timer_create(CLOCK_REALTIME, NULL, &timer1) != 0)
{
        // If there is an error, print out a message and exit.
        perror("timer_create");
        exit(1);
}
```

[4] Not to be confused with Timer/Counter units on the microcontroller.

The following code shows how to create a timer that sends the `SIGUSR1` signal.

```
timer_t timer2;
struct sigevent timer2_event;

// Zero out the structure and configure for use of the SIGUSR1 signal.
memset(&timer2_event, 0, sizeof(timer2_event));
timer2_event.sigev_notify = SIGEV_SIGNAL;
timer2_event.sigev_signo  = SIGUSR1;

// Create a new timer that will send the SIGUSR1 signal.
if (timer_create(CLOCK_REALTIME, &timer2_event, &timer2) != 0)
{
        // If there is an error, print out a message and exit.
        perror("timer_create");
        exit(1);
}
```

The `timer_settime()` function is used to schedule a timer. The `struct itimerspec` definition taken from */usr/include/linux/time.h* is seen here.

```
struct itimerspec {
        struct timespec it_interval;    /* timer period */
        struct timespec it_value;       /* timer expiration */
};

struct timespec {
        time_t tv_sec;         /* seconds */
        long   tv_nsec;        /* nanoseconds */
};
```

The `it_value` member sets the time until the timer first expires. If it is set to 0, the timer will never go off. The `it_interval` member sets the period of the timer after it first expires. If it is set to 0, the timer will be one-shot.

Following is an example of scheduling `timer1` (created in a preceding example) to go off in 2.5 seconds, and then every 100 milliseconds thereafter.

```
struct itimerspec timer1_time;

// The it_value member sets the time until the timer first goes off (2.5 seconds).
// The it_interval member sets the period of the timer after it first goes off (100 ms).
timer1_time.it_value.tv_sec     = 2;          // 2 seconds
timer1_time.it_value.tv_nsec    = 500000000;  // 0.5 seconds (5e8 nanoseconds)
timer1_time.it_interval.tv_sec  = 0;          // 0 seconds
```

```
timer1_time.it_interval.tv_nsec = 100000000;  // 100 milliseconds (1e8 nanoseconds)

// Schedule the timer.
if (timer_settime(timer1, 0, &timer1_time, NULL) != 0)
{
        // If there is an error, print out a message and exit.
        perror("timer_settime");
        exit(1);
}
```

6.6 PCI-DAS1602/12 ADC/DAC Card

The PCI-DAS1602/12 is a PCI card that contains an analog to digital converter (ADC) and two digital to analog converters (DAC). In CS431 labs, we use only a DAC.

Communication with the PCI-DAS1602/12 is performed via low-level port input (`inw()`) and output (`outw()`) To use these functions, `sys/io.h` must be included. `inttypes.h` should also be included to use the fixed width integer types (`uint16_t`, etc.).

- `uint16_t inw(uint16_t port)`
  Returns the byte read from the specified I/O port.
- `void outw(uint16_t val, uint16_t port)`
  Writes byte `val` to the specified I/O port.

A description of the available ports on the card can be found in the PCI-DAS1602/12 Register Map data sheet. To find the card's base addresses, use the linux terminal command `lspci -v`.

Example:

```
> lspci -v
Class ff00: Measurement Computing PCI-DAS1602/12
Flags: bus master, fast devsel, latency 64, IRQ 9
I/O ports at ccc0 [size=64]
I/O ports at cc40 [size=32]
I/O ports at cc60 [size=32]
I/O ports at cc80 [size=32]
I/O ports at cca0 [size=32]
```

The output of this command tells you the the physical addresses for the base addresses of the card, in this case: BADR0 = 0xCCC0, BADR1 = 0xCC40, BADR2 = 0xCC60, BADR3 = 0xCC80, BADR4 = 0xCCA0

When using the functions `inw` and `outw`, these will be the ports used for input/output. It is best practice to declare these ports in your code, for example:

```
#define BADR0 0xCCC0
```

```
#define BADR1 0xCC40
#define BADR2 0xCC60
#define BADR3 0xCC80
#define BADR4 0xCCA0
```

Then, you can use the inw/outw functions with these port names, for example:

```
outw(0, BADR0+2);
```

6.6.1 I/O Permissions Wrapper

Read and write access to I/O ports is restricted in Linux. A regular user program attempting to communicate with the PCI-DAS1602/12 will cause a segmentation fault. To get around this problem, a wrapper program called `wrap-ioperm` is provided. `wrap-ioperm` starts running set-uid root. It opens up read/write access to card's base address ports, switches to the invoking user, and executes the desired program.

To use the wrapper, simply prefix a normal command with `wrap-ioperm`. For example, instead of running `./myprogram my args`, execute `wrap-ioperm ./myprogram my args`.

6.6.2 Initialization

The following steps initialize the card for digital to analog conversions:

- Set BADR1+8 configuration parameters
  - Set LDAEMCL to 1, resets EMPTY status flag of DAC FIFO queue (FIFO queue has four entries)
  - Set DACEN to 1, enables the DACs
  - Set DAPS bits low, tells DAC that data is from software source
  - Set HS bits to select desired DAC chip
  - Set DACnR bits to select voltage range (we will use bipolar 5V)
  - Set START bit to 1, tells DAC to start buffering data
- Clear the DAC FIFO buffer by writing any value to BADR4 + 2

6.6.3 Digital to Analog Converter (DAC)

The PCI-DAS1602/12 digital to analog converter has 12-bits of resolution over a software programmable voltage range. We will use bipolar 5V (-5 volts to $+5$ volts) as the voltage range in our labs.

To send the digital value and start the D/A conversion, write the value to the low bits of port BADR4+0. After the D/A conversion starts, you do not need to wait for the conversion to finish. However, you can check the status of DAC FIFO queue by reading ADNE and LADFUL bits of BADR1+0.

Function	Use
`tcsetattr()`	Change device attributes (baud, parity, etc.)
`tcgetattr()`	Read existing device attributes
`tcflush()`	Flush pending sends or receives
`fileno()`	Returns the integer file descriptor for a `FILE *` stream

Table 11: Linux serial port control functions

6.7 Serial Port

Under Linux, the serial port can be accessed through `/dev/ttyS[0-3]`, which correspond to COM1-4 respectively.

The serial port is configured by the operating system at boot. To see the current settings for `ttyS0`, execute: `setserial -a /dev/ttyS0`

The source for a simple Linux program that writes to the serial port is included in section 7.2.1 for reference.

6.7.1 Communication Settings

Serial port settings such as baud, data bits, stop bits, and parity can be adjusted by any process with permission to access the serial device. Since the device resource is shared among many processes, it is good practice to flush pending transfers and save/restore existing device settings whenever a change is made.

The Linux serial port control functions are listed in table 11.

6.7.2 Sending and Receiving

Opening, closing, reading from, and writing to a serial device under Linux is identical to standard file I/O. The code below sends "Hello world!" via the serial port using the existing baud rate and data format.

```
int fd = open("/dev/ttyS0", O_WRONLY);
dprintf(fd, "Hello world!");
close(fd);
```

6.8 Processes

A process is simply an instance of a running program. In Linux, processes can communicate with each other using a number of mechanisms such as pipes, sockets, shared memory, and message queues. CS 431 will only focus on the use of message queues.

6.8.1 Message Queues

Message queues have a name (similar to files) and can be opened by multiple processes simultaneously for read and/or write access. `mq_open()` is used to open a message queue as seen in the following example. See the man page for additional details.

```
struct mq_attr queue_attribute;
mqd_t queue;

// message can be of any type (basic type, struct, etc.).
uint8_t message;

// Setup the message queue attributes:
//   - the queue can hold up to 10 messages
//   - each message can be up to sizeof(message) bytes
memset(&queue_attribute, 0, sizeof(queue_attribute));
queue_attribute.mq_maxmsg = 10;
queue_attribute.mq_msgsize = sizeof(message);

// Open the queue write-only and non-blocking.  If the queue does not exist,
// create it (O_CREAT) with permission 600.  Note that on Linux, the name
// string must begin with a '/'.
queue = mq_open("/queue_name", O_WRONLY | O_NONBLOCK | O_CREAT, 0600, &queue_attribute);
if (queue == ((mqd_t) - 1))
{
        // If there is an error, print out a message and exit.
        perror("mq_open");
        exit(1);
}
```

Once a message queue is opened, `mq_send()` inserts a message into the queue, and `mq_receive()` removes a message from the queue. `mq_close()` and `mq_unlink()` close and delete a queue, respectively. Check the man pages for how to use those functions.

6.9 Threads

A process may have several threads of execution concurrently running within it. The threads all share the same memory space and permissions of the process.

6.9.1 Creating and Terminating

`pthread_create()` is used to create a new thread as seen here.

```
// Thread start routine.
void *new_thread(void *arg)
```

```
{
        // When the new thread is created, it will start executing in this function.
}

// Create a new thread.
pthread_t new_thread_id;
if (pthread_create(&new_thread_id, NULL, new_thread, NULL) != 0)
{
        // If there is an error, print out a message and exit.
        perror("pthread_create");
        exit(1);
}
```

To terminate a thread, use `pthread_exit()`. If the initial "main" thread of a program is terminated by reaching the end of executing or calling `pthread_exit()`, the entire process will exit.

6.9.2 Mutual Exclusion

When multiple threads share access to a resource such as data (global variables), it is necessary to ensure that the integrity of the data is maintained. For example, one thread should not modify the data while another is in the process of reading it. One thread synchronization mechanism which can solve this problem is a mutual exclusion lock, or mutex.

A mutex is in one of two states at all times: unlocked or locked. When unlocked, any thread may acquire a lock on the mutex. All other threads requesting a lock are then queued up until the thread holding the mutex unlocks it. To protect shared data with a mutex, each thread must lock the mutex before accessing the data and unlock it after the data usage is complete.

The three functions for mutex control are `pthread_mutex_init()`, `pthread_mutex_lock()`, and `pthread_mutex_unlock()`. See the man pages for details on their usage.

6.9.3 Signals

Some thread-related functions are not safe to be used within a signal handler. For example, calling `pthread_mutex_lock()` or `pthread_mutex_unlock()` inside a signal handler can deadlock the calling thread. A good thread-friendly alternative to signal handlers is the `sigwait()` blocking technique discussed in section 6.4.2.

Instead of blocking each signal to wait on individually, the `sigwait()` man page makes the following recommendation for multi-threaded applications:

> For sigwait to work reliably, the signals being waited for must be blocked in all threads, not only in the calling thread, since otherwise the POSIX semantics for signal delivery do not guarantee that it's the thread doing the sigwait that will receive the signal. The best way to achieve this is to block those signals before any threads are created, and never unblock them in the program other than by calling sigwait.

This can be accomplished by executing the following code before any new threads are created.

```
// Block all signals before creating any threads.
sigset_t signal_set;
sigfillset(&signal_set);
if (pthread_sigmask(SIG_BLOCK, &signal_set, NULL) != 0)
{
        // If there is an error, print out a message and exit.
        perror("pthread_sigmask");
        exit(1);
}
```

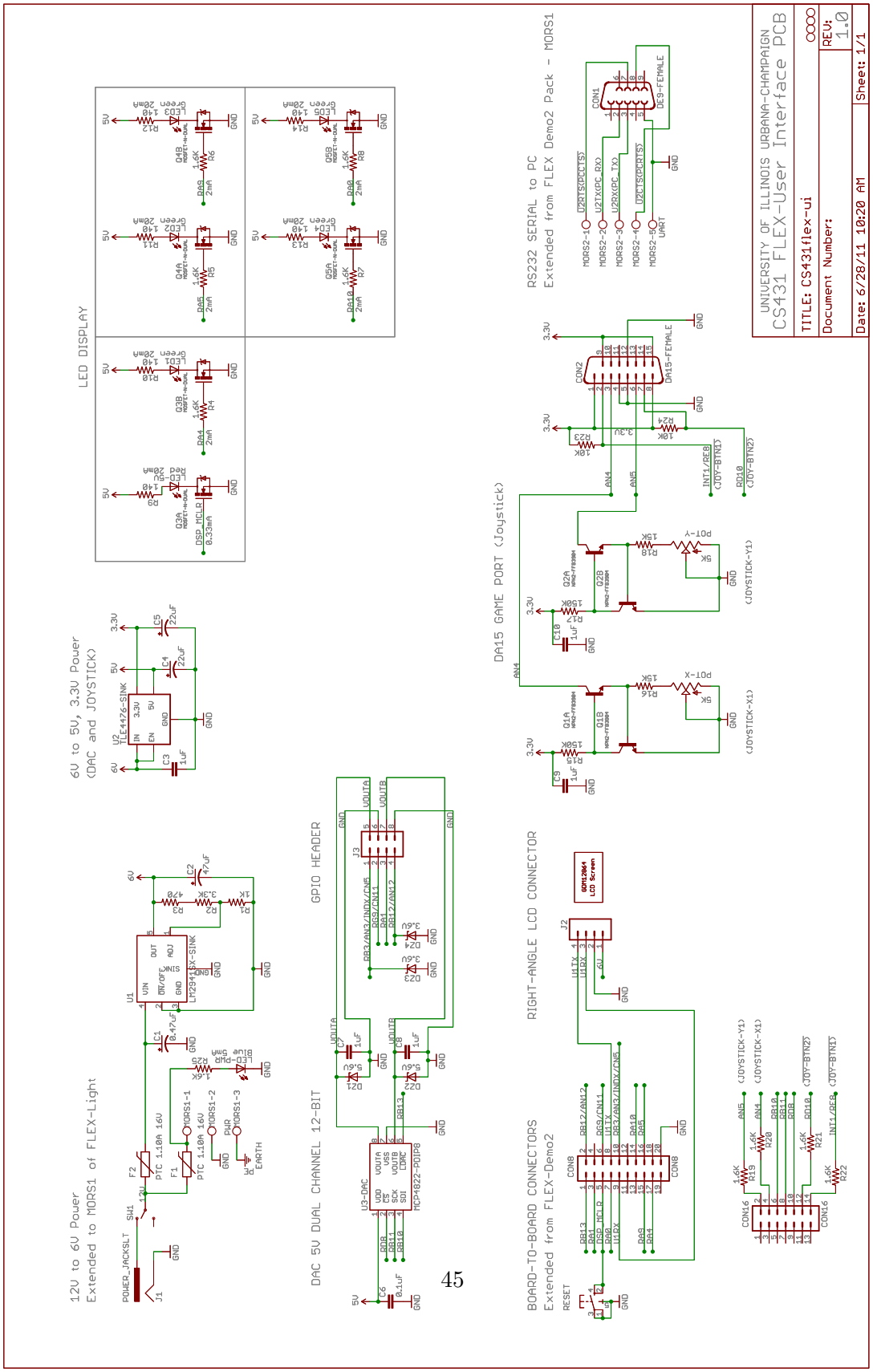
`sigwait()` can then be used as seen here.

```
sigset_t signal_set;
int signal_received;

// Setup a signal set that contains all of the signals that sigwait() will wait for.
sigemptyset(&signal_set);
sigaddset(&signal_set, SIGALRM);
sigaddset(&signal_set, SIGUSR1);

while (1)
{
        // Block until either SIGALRM or SIGUSR1 is received.
        sigwait(&signal_set, &signal_received);

        // Act according to the signal that was received.
        if (signal_received == SIGALRM)
        {
                // Do SIGALRM stuff...
        }
        else if (signal_received == SIGUSR1)
        {
                // Do SIGUSR1 stuff...
        }
}
```

7 Appendix

7.1 FLEX-UI Schematic

Schematic diagram of the custom FLEX-UI board.


UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
CS431 FLEX-User Interface PCB
TITLE: CS431flex-ui
Document Number:
REV: 1.0
Date: 6/28/11 10:20 AM
Sheet: 1/1
LED DISPLAY
6V to 5V, 3.3V Power
(DAC and JOYSTICK)
12V to 6V Power
Extended to MOR51 of FLEX-Light
DAC 5V DUAL CHANNEL 12-BIT
GPIO HEADER
DA15 GAME PORT (Joystick)
RS232 SERIAL to PC
Extended from FLEX Demo2 Pack - MOR51
BOARD-TO-BOARD CONNECTORS
Extended from FLEX-Demo2
RIGHT-ANGLE LCD CONNECTOR
(JOYSTICK-Y1)
(JOYSTICK-X1)
(JOY-BTN27)
(JOY-BTN17)

7.2 FLEX-UI Pinouts

Pin mapping from dsPIC microcontroller to FLEX-UI board.

7.2.1 Linux Serial Demo

This code can be found in *cs431/labs/lab03/code/server/serialdebug.c* on the lab machines. It opens the serial port for write access and sends characters entered by the user.

Pins Extended from FLEX Demo Daughter Board

Connector	DEMO Pin	dsPIC33F Pin	Description of Ports	Custom PCB Use	Other Notes
CON7 – UART to output DB9	1	-	CTS(232),CTS(TTL),NC(485),TX+(422)	**UART_CTS – U2CTS#**	PIN40 with UART support hardware
CON7 – UART to output DB9	2	-	RX(232),RX(TTL),485-,TX-(422)	**UART_RX – U2RX**	PIN49 with UART support hardware
CON7 – UART to output DB9	3	-	TX(232),TX(TTL),485+,RX+(422)	**UART_TX – U2TX**	PIN50 with UART support hardware
CON7 – UART to output DB9	4	-	RTS(232),RTS(TTL),NC(485),RX-(422)	**UART_RTS – U2RTS#**	PIN39 with UART support hardware
CON7 – UART to output DB9	5	-	GND	**UART_GND**	
CON8 – AUX Connector	1	PIN42	RB13/AN13	**DAC_LDAC - RB13**	
CON8 – AUX Connector	2	PIN41	RB12/AN12	**GPIO – RB12/AN12**	Can be used as ADC port for loopback
CON8 – AUX Connector	3	PIN38	RA1/TCK	**GPIO - RA1**	
CON8 – AUX Connector	4	PIN47	RD14/IC7/U1CTS#/CN20	**--**	**Same as CON3-3**
CON8 – AUX Connector	5	PIN13	MCLR#	**Soft Reset PushBtn**	
CON8 – AUX Connector	6	PIN14	RG9/SS2#/CN11	**GPIO - RG9**	
CON8 – AUX Connector	7	PIN17	RA0/TMS	**LED5 – RA0**	
CON8 – AUX Connector	8	PIN51	RF3/U1TX	**LCD_TX - U1TX**	Sparkfun Serial Graphic LCD (09351)
CON8 – AUX Connector	9	PIN52	RF2/U1RX	**LCD_RX – U1RX**	Sparkfun Serial Graphic LCD (09351)
CON8 – AUX Connector	10	PIN22	RB3/AN3/INDX/CN5	**GPIO – RB3/AN3/INDX/CN5**	Can be used as ADC port for loopback
CON8 – AUX Connector	11	PIN74	RC14/PGEC2/EMUC2/SOSCO/T1CK/CN0	**--**	**Jumper JP16 to XT1**
CON8 – AUX Connector	13	-	+AVDD_EXT	**--**	**Jumper JP6 to PIN30 +AVDD**
CON8 – AUX Connector	14	PIN29	RA10/VREF+	**LED4 – RA10**	
CON8 – AUX Connector	15	PIN28	RA9/VREF-	**LED3 – RA9**	
CON8 – AUX Connector	16	PIN61	RA5/TDO	**LED2 – RA5**	
CON8 – AUX Connector	17	PIN60	RA4/TDI	**LED1 – RA4**	
CON8 – AUX Connector	18	-	+5V	**--**	
CON8 – AUX Connector	19	-	+3.3V	**--**	
CON8 – AUX Connector	20	-	GND	**GND – Digital**	
CON16 – DC Motor 1 Connector	2	PIN20	RB5/AN5/QEB/CN7	**Joystick AY -AN5(ADC2)**	Game Port Pin6, +3.3V (LDO-VREG)
CON16 – DC Motor 1 Connector	4	PIN21	RB4/AN4/QEA/CN6	**Joystick AX -AN4(ADC2)**	Game Port Pin3, +3.3V (LDO-VREG)
CON16 – DC Motor 1 Connector	6	PIN34	RB10/AN10	**DAC_SDI – RB10**	
CON16 – DC Motor 1 Connector	8	PIN35	RB11/AN11	**DAC_SCK – RB11**	
CON16 – DC Motor 1 Connector	10	PIN68	RD8/IC1	**DAC_CS – RD8**	
CON16 – DC Motor 1 Connector	12	PIN70	RD10/IC3	**Joystick A2 -RD10**	Game Port Pin7, +3.3V (LDO-VREG)
CON16 – DC Motor 1 Connector	14	PIN18	RE8/AN20/FLTA#/INT1	**Joystick A1 -INT1/RE8**	Game Port Pin2, +3.3V (LDO-VREG)
CON16 – DC Motor 1 Connector	16	PIN19	RE9/AN21/FLTB#/INT2	**--**	