Digital Input and Output with the Cerebot MX7cK™

Revision: 29 Jan 2019 (JFF)
Author: Professor Richard Wall, University of Idaho, rwall@uidaho.edu

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 1: Digital Input and Output

Table of Contents

Project 1: Digital Input and Output 1

Table of Contents 1

Purpose 2

Minimum Knowledge and Programming Skills 2

Equipment List 2

Software Resources 2

References 2

Background Digital Systems Concepts 2

I/O Concepts 3

I/O Specific to the PIC32 Processor 4

Digital I/O Signal Conditioning 6

Project Tasks 7

Project Program Verification 9

Appendix A. X32 C I/O Syntax 10

TRIS: 10
LAT: Read-Modify-Write 10
ODC: 11
PORT: 11
Appendix B. Peripheral Library I/O Functions 12

Appendix C. Detailed I/O PIN Block Diagram 14

Appendix D. Software Models and Planning 14

References for Code Organization: 19

Purpose

The purpose of this project is to familiarize you with the methods of reading from and writing to the input and out (I/O) pins of the PIC32 microcontroller. Software modeling concepts are presented using data flow diagrams and control flow diagrams.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB ® IDE
3. Introductory Digital Systems
4. Fundamental understanding of microprocessors

Equipment List

1. Cerebot 32MX7cK processor board with USB cable
2. Microchip MPLAB ® X IDE
3. MPLAB XC32 Compiler

Software Resources

1. XC32 ® C/C++ Compiler Users Guide
2. PIC32 Peripheral Libraries for MBLAB C32 Compiler
3. PIC32 Family Reference Manual Section 12: I/O Ports
4. MPLAB ® X Integrated Development Environment (IDE)
5. C Programming Reference

References

1. Cerebot MX7cK Board Reference Manual
2. Cerebot MX7cK Board Reference Schematic
3. PIC I/O Ports - http://www.mikroe.com/chapters/view/4/
4. Digital Systems - http://www.electronicsteacher.com/computer-architectures/digital-circuits/
5. The Basics – Output - http://www.basicmicro.com/downloads/docs/Smith_halfChp3.pdf .

Background Digital Systems Concepts

Projects 1 through 10 are presented with the expectation that the reader has a fundamental knowledge of digital systems. The key concepts with which the reader should be familiar are Boolean algebra, binary math, and number base conversion (base 10 to base two and base sixteen or hexadecimal). Along with an understanding of logic gates (AND, OR, INVERT, XOR),

the reader will also find it useful to have an understanding of basic digital elements like flip flops, latches, multiplexers, and tri-state buffers.

I/O Concepts

Microprocessor pins are commonly either configured to be a digital input or digital output hence are usually referred to as I/O pins. Digital input pins allow microprocessors to receive binary data from their environment. Binary data can only have two possible values: either one or zero. These to binary values can be used to represent various conditions such as on or off, high or low, and true or false. A single item of binary data is often referred to as a bit (short for binary digit). By sensing the presence or absence of a voltage, input pins can be used to detect binary events such as the pressing of a button. Individual digital output pins can be used to control single function devices such as turning an LED on or off. Alternatively, a collection of pins can be thought of as representing a "binary word". The use of a *word* to represent a collection of data bits is generic and the number of bits it represents depends on both the processor and the specific C compiler being used. Having made the distinction between the use of *word* to represent a collection of bits and a specific integer data type, we will follow the rules established in section 6.4 of the MPLAB ® XC32 C Compiler User's Guide.

The most basic means of communicating to and from a microprocessor is through I/O pins. On the PIC32 microprocessor, I/O pins are grouped in terms of *ports*, with 16 pins per port, and the collection of bits representing the state of the I/O pins of a designated port itself can be thought of as a word. The pins of each port are identified by a bit position in the form Rp0 to Rp15, where "p" is a letter that represents the particular I/O port on the processor and the numerical value (in the range 0 through 15) is the bit position within the word. The way in which each bit within a word should be valued or interpreted is at the discretion of the code developer. The conventional C (unsigned) integer data type uses binary encoding where the implicit "place" value associated with the position of a bit is given by 2 raised to the value of the bit's position. For example, the implicit value associated with the bit Rp3 is 8, corresponding to 2 raised to the third power. The value of a given bit is further dictated by whether the bit has an actual value of one or zero. For example, if the bit Rp3 is zero (low), then its actual value is actually zero (i.e., zero times the implicit place value). On the other hand, if the bit Rp3 is one (high), then its explicit value is 8 (i.e., one times the place value). Words may be interpreted, or "encoded," in other ways. For example, signed data use two's complement encoding. Other encoding schemes include floating point (used for real numbers), ASCII (used to represent characters), and Gray code (often used to represent position), to name just a few.

With 16 I/O pins available on a PIC32 port, values can be represented ranging from zero to 65535 (base 10), or, in hexadecimal, zero to 0xFFFF (the leading "0x" indicates a hexadecimal representation of a number where each digit can have one of 16 values [0 through F] while a leading "0b" will be used to indicate a binary representation where each digit can only have one of two values [0 or 1]). On the PIC32MX7 microcontroller, I/O ports are labeled A through G. Because digital ports are limited to 16 bits, values ranging from zero to 65535 can be directly read from or written to external connections. For example, if RB3 is set high and all other Port B pins are set low, then the processor is outputting a value of eight (0x0008 or in binary, 0b0000 0000 0000 1000). (See Section 6.4 of Microchip XC32 Users Guide for alternate constant expressions.)

Input pins can also be used to detect events by sensing switches, button pushes, or some other device that outputs a binary signal, provided that an appropriate voltage level is generated by

the sensor device. Multiple event type inputs can be connected to a single 16-bit processor port.

I/O Specific to the PIC32 Processor

Figure **1** shows a simplified block diagram for a PIC32 I/O pin. The more detailed block diagram appearing in Appendix C is a reproduction from section 12 of the PIC32MX Users Guide and may provide additional useful information for advanced users. The characteristics of each I/O pin are individually controlled by setting bits in four, or in some cases, five registers. The PIC32MX795 processor used on Cerebot MX7cK supports 16 pins that can be used as either an analog input or a digital I/O. Ten of these analog input pins are routed to Pmod connectors, as discussed in the Analog Inputs section of Cerebot MX7cK Reference Manual. By default, when the PIC32 is reset, these pins are set as analog input. The bits in the AD1PCFG register corresponding to the register and pin number must be set to a 1 if the pin is to be used as a digital input or output.

One may also use *PORTSetPinsDigitalIn(port, bits);* or *PORTSetPinsDigitalOut(port, bits);* as an alternative to setting the AD1PCFG register. These are provided by the XC32 peripheral library to set the pin as a digital input or output. For this project, only digital I/O will be used. Refer to Appendix A and B for alternate methods for configuring I/O pins.

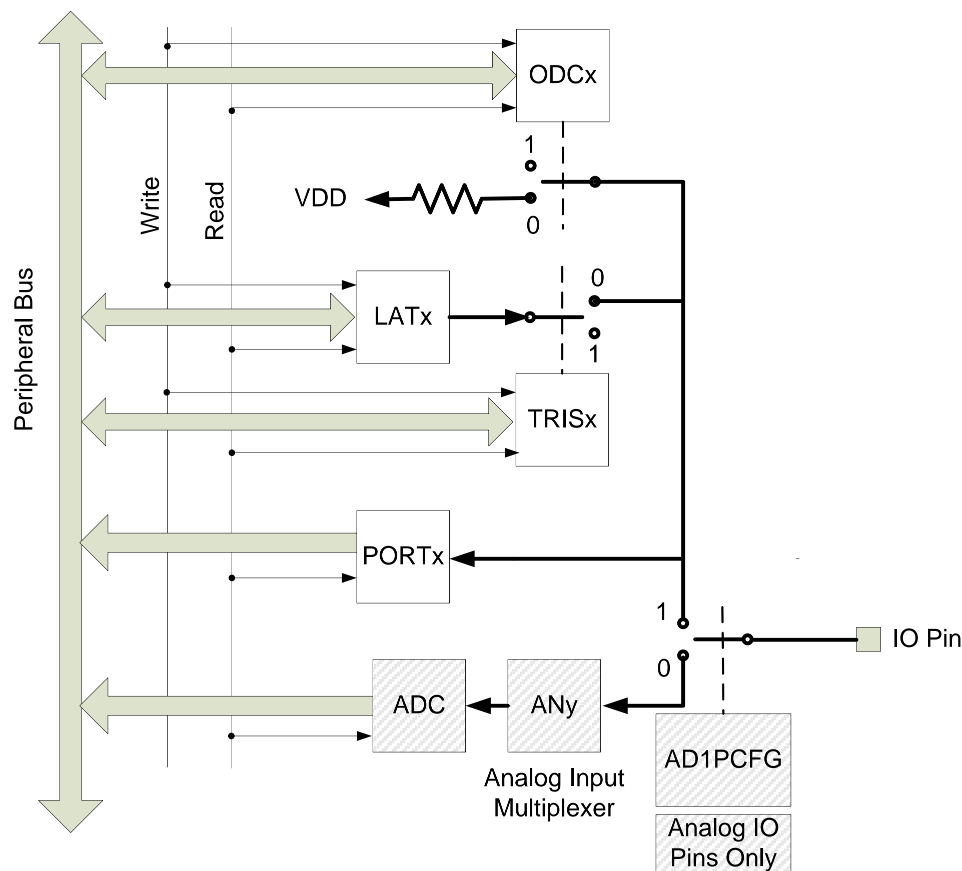

Figure 1. Simplified block diagram for a PIC32 I/O pin.

The TRISi, PORTi, LATi, and ODCi are all 16 bit registers with the designation of *i* ranges from A through G. The TRISi, LATi, and ODCi registers control the individual I/O pins corresponding to the bit position. For example, RG6 is the designation for bit 6 of the PORTG register. The TRIS (tri-state) control determines whether the pin is to be used as an input or an output. This is commonly referred to as setting the I/O pin direction. Setting the bit position in the TRIS register to a '0' makes the pin an output. This means that the state of the pin is controlled by the values written to the LAT and ODC registers. Setting the bit position in the TRIS register to a '1' makes the pin an input.

The PORT register allows the PIC32 to sense the "state" of the physical I/O pin. The following example shows how to read the PORT G register:

```
int x;
x = PORTG;
```

Regardless of the TRIS register's settings, the state of an I/O pin can always be determined by reading the associated PORT register. Setting bit 6 of TRISG to a one makes pin RG6 an input but this is only because the output is disconnected by the TRISA tri-state control.

The PIC32MX family of processors is designed to operate with 3.3V inputs. More specifically, an input below the threshold of 0.66V is read by the processor as a low (a value of zero or a logical false). A voltage on an input pin that is above the threshold of 2.64V is read as a high (a value of one, or a logical true). All digital input-only pins are 5V tolerant, meaning that a device that outputs 5V for logic high can be connected to digital input pins and will not damage the PIC32MX processor. The maximum voltage that can be applied to I/O pins that can be used as either analog or digital I/O is V_{DD}+0.3V, or 3.6V for the Cerebot MX7cK. Exceeding this voltage limit will damage the processor.

The latch (LAT) register is used to control the output level. If the TRIS register is set to configure the I/O pin as an output, a logic zero written to the LAT register always results in a low voltage level pin. A logic one written to the LAT register results in the output pin being either a high voltage level (3.3V) or it results in high impedance, depending on the state of the open drain control (ODC) register. The ODC allows the output to either source up to 25mA at 3.3V or to be in the high impedance state . The default setting (reset condition) for the ODC register is all zeros, which means the outputs can both sink and source current.

The LAT register is a memory element that stores the output state of the I/O pins. The LAT register can be written to at any time regardless of the direction of the bits, as controlled by the TRIS register. To assign a value to the LAT register, one might use the following example

```
LATG = 0xA5;
```

However, the TRIS register still controls which LAT bits are enabled to the I/O pins. The bits in the LAT registers can be read as well written. For example:

```
int x;
x = LATG;
```

As previously discussed, even when a pin is declared as an output, the logic value of the pin voltage can still be read by reading the PORT register. However, there is no guarantee that a

device attached to the pin is not loading the output pin to the degree that the output appears to be low when it is actually programmed to be high. Thus when one wants to know the output state that an I/O pin was previously programmed for, it is preferable to read the LAT register associated with the I/O port. The preferred method for reading the logic levels previously programmed to an I/O pin is using the following code:

```
int x;
x = LATG;  // Read IO LAT register for port G,
```

In addition to the register assignments that result in all 16 bits of the register being set at one time, there are modifier instructions that provide bit control. The modifiers that provide atomic bit manipulation are SET, CLR, and INV. For the SET modifier, the bit positions that are set to a one in the instruction will set the register bits high, while not altering the bit in the unspecified positions. For example, the instruction *LATGSET = 0x8020;* will result in bits 5 and 15 being set to a one. Bits 0 through 4 and 6 through 14 will not be changed.

Similarly, the modifier CLR sets the specified bits to a zero. The instruction *LATGCLR = 0x8020;* will result in bits 5 and 15 being set to zero. Bits 0 through 4 and 6 through 14 will not be changed. The modifier INV toggles the selected bits. If prior to the instruction, the specified bit location was set high, it will be set low after execution of the instructions, and vice versa. The instruction *LATGINV = 0x8020;* will toggle bits 5 and 15.

The pins associated with the buttons and LEDs on the Cerebot MX7cK processor board can be determined by referring to either the Cerebot MX7cK reference manual or the Cerebot MX7cK schematic. LED1 through LED4 are connected to PORTG pins 12 through 15, respectively. BTN1 and BTN2 on the PIC32MX7 are connected to PORTG pins 6 and 7. Although BTN3 is not used in this project, it will be used in future projects, and it is worth noting that BTN3 connects to I/O Port A bit 0 and requires disabling the JTAG programmer using the instruction: *DDPCONbits.JTAGEN = 0;*.

Refer to Appendix B and http://ww1.microchip.com/downloads/en/DeviceDoc/61120D.pdf for alternative methods of using and configuring PIC32 I/O pins.

Digital I/O Signal Conditioning

The detailed block diagram of an I/O pin for a PIC32 port is shown in Appendix C. The logic value at any I/O pin can be read through the PORT register, regardless of whether it is assigned as an input or an output. As addressed above, the maximum voltage that can be applied to pins that can be assigned as an analog input is 3.6VDC. Other pins are 5V tolerant, which means that when the pin is assigned as an input, an external voltage of up to 5V can be applied to the PIC32 processor without causing damage. The schematic diagram for the Cerebot MX7cK has a transient super diode with 5V clamping on each processor I/O pin that is connected to a Pmod interface jack. This diode's purpose is to mitigate transient voltages, and it is not designed to limit the voltage applied to the processor pin.

The input current of an I/O pin configured as an input is ±1 μA, which makes the input resistance greater than 3MΩ. Active high inputs from switches should have a pull-down resistor to drain trapped charges on processor inputs. Unused inputs should be terminated with either a pull-up or pull-down resistor, or processor current consumption may increase due to transistor

switching action causing noise inside the processor. An alternative to terminating unused I/O pin is to program the pins to output a low level.

The maximum current capability of each conventional I/O pin is 25mA (sink or source), while the combined current of all I/O pins is 200mA subject to total power constraints. I/O pins have both open drain and active source output capability. The open drain capability provided by the ODC register is useful when interfacing with switch array keypads. (See PmodKYPD for an example key pad device.) The outputs that drive the four LEDs on the Cerebot MX7cK board are limited to approximately 5mA.

Additional insight into the effects of I/O pin loading is provided by the Microchip application note AN234 and the link found at "The Basics – Output".

Project Tasks

The only hardware required for this project is the Cerebot MX7cK processor board. We will be using two of the momentary pushbuttons as inputs and the four LEDs as outputs. The goal is to use combinations of buttons that are pressed to control which LED is on, based on the action described in a truth table.

Initially, generate a new project as described in Project 0. Copy the three common files config_bits.h, CerebotMX7cK.h and CerebotMX7cK.c to the project directory and add them to the project. Modify the C file called Project1.c and modify it as shown in Listing 1 below:

Listing 1.

```
#include <plib.h>
#include "CerebotMX7cK.h"

int main(void)
{
	sys_init();	/* See the function Cerebot_mx7cK_setup() in CerebotMX7cK.c
			for details of configuration of the processor board  */

	/* Project initialization code goes here   */

	while(1)
	{
		/* Project operational code goes here */
	}
	return 1;	// This line of code will never be executed.
}
```

The first step for any microprocessor based designed is to configure the microprocessor to accommodate the hardware platform and the application requirements. For this project, the board initialization requires only a few program statements. Port G bits 6 and 7 must be set as digital inputs to allow reading of the pushbuttons BTN1 and BTN2. This can be accomplished using the following statement:

```
PORTSetPinsDigitalIn(IOPORT_G, BIT_6 | BIT_7);
```

This project does not use BTN3 but If an application did require the use of BTN3, the next two lines of code are needed.

```
DDPCONbits.JTAGEN = 0;      // Disable JTAB programming
PORTSetPinsDigitalIn(IOPORT_A, BIT_0);
```

Port G bits 12 through 15 must be set as digital outputs to control LED1 through LED4. Usually, the initial value of the outputs must also be set. This can be accomplished using the following statements:

```
PORTSetPinsDigitalOut(IOPORT_G, BIT_12 | BIT_13 | BIT_14 | BIT_15);
PORTClearBits(IOPORT_G, BIT_12 | BIT_13 | BIT_14 | BIT_15);
```

Constants BIT_0 through BIT_15 are defined as constants 0x0001, 0x0002, 0x0004 … 0x8000 in the included file, *plib.h*. The *CerebotMX7ck.h* file also defines LED1 through LED4 as BIT_12 through BIT_15 as well as defining BTN1 as BIT_6, BTN2 as BIT_7 and BTN3 as BIT_0. It is then possible to use the following lines of code to set the button and LED I/O if the file *CerebotMX7ck.h* is included in a project:

```
PORTSetPinsDigitalIn(IOPORT_G, BTN1 | BTN2);
DDPCONbits.JTAGEN = 0;      // Disable JTAB programming
PORTSetPinsDigitalIn(IOPORT_A, BTN3);

PORTSetPinsDigitalOut(IOPORT_G, LED1 | LED2 | LED3 | LED4);
PORTClearBits(IOPORT_G, LED1 | LED2 | LED3 | LED4);
```

The latter method is preferable because there is more information conveyed to someone reading the program when, for example, BTN1 is used instead of BIT_6, even though they represent the same value. From a sustainable software perspective, both BTN1 and BIT_6 are preferred over using the value 0x40 or 64 to indicate a bit position. Such explicit numerical representations constitute a magic number because they appear with no explanation. Magic numbers in software programming are to be avoided and usually can usually be eliminated using "#define" declarations.

Appendix A discusses alternatives methods for programming I/O that the reader may encounter in other PIC32 literature and reference designs.

After the processor has been initialized, the program must continually read the input to determine whether a particular button is pressed (logical value of true) or released (logical value of false), execute logic to determine which of the LEDs to turn on and off, and set the output pins accordingly. The program maps the two button inputs to one of four LED outputs using the logic expressed by the truth table shown in Table 1. The LEDs are driven by the output pins so writing a logical 1 to the LED port will cause the LED to be lit.

The infinite processing is implemented inside the *while(1)* loop, as illustrated in the flow diagram shown in Figs. 3 and 4 of Appendix D. The action of using explicit software instruction to determine the state of an input is called "polling."

Project Program Verification

After compiling your project and downloading the code to the microprocessor, the code can be verified by simply pressing BTN1 and/or BTN2 and visually observing the on/off status of the four Cerebot MX7cK LEDs. For example, when BTN1 and BTN2 are pressed at the same time, LED4 should be on. Never underestimate the value of information that can be gained by turning an LED on or off to indicate an action. The technique is simple to code and easily observed.

Alternatives to polling are addressed in Project 5 –that deals with interrupts or preemption.

Table 1. Truth table for mapping inputs to outputs

Inputs		Outputs			
BTN2	**BTN1**	**LD4**	**LD3**	**LD2**	**LD1**
Off	Off	Off	Off	Off	On
Off	On	Off	Off	On	Off
On	Off	Off	On	Off	Off
On	On	On	Off	Off	Off

Appendix A. X32 C I/O Syntax

TRIS:

The “tri-state” register, or “TRIS” register, is used to specify whether a pin is used as an output or an input. I/O pins are individually assigned as inputs or outputs by assigning a zero or one to the corresponding bit position within the TRIS register. Writing a zero to the bit position causes the pin to operate as an output. Writing a one to the bit position causes the pin to operate as an input. An example of a C statement to simultaneously configure all 16 bits of the TRIS register is:

```
TRISA = 0x0030;
```

This instruction sets PORT A pins 4 and 5 as input pins and all others as output pins.

In general, it is not a good practice to set the “direction” of all 16 bits of an I/O port when the goal is really only to set the direction of a subset of the I/O port pins. For example, if only pins 12 to 15 are to be set as output and bits 0 to 3 are to be set as inputs while direction of all other pins is left unspecified, then it is best to refrain from setting the direction of all 16 bits. To accomplish this, use two instructions:

```
TRISASET = 0x000F; // Set RA0, RA1, RA2, and RA3 to be input pins without changing
                   // the direction of other PORT A pins
TRISACLR = 0xF000; // Set RA12, RA13, RA14, and RA15 to be output pins without
                   // changing the direction of other PORT A pins
```

Modifying the direction of specific bit positions eliminates the possibility of “undoing” something that was configured earlier in the program execution.

LAT: Read-Modify-Write

Being able to both read and write to the LAT register is an important feature of many modern microprocessors. Sometimes it is necessary to implement a read-modify-write sequence of instructions to change a subset of port pins without modifying the state of other pins sharing the same port register. As discussed in the text above, due to electrical loading on the I/O pin by an external device or components, the processor may read the state of the pin at a different logic level using the PORT register than is actually set into the LAT register. The solution to this conflict is to determine the output settings by reading the LAT register, rather than the PORT register. Hence, a proper read-modify-write sequence is shown in the following code example where the constant, *MASK*, defines the bits locations that will be modified and the variable new_data contains the values the bits at the locations that are to be modified.

```
int temp, new_data;

temp = LATG;                  // Read all 16 port G bits
temp = temp & ~MASK;          // Clear all bit locations set to one in the constant MASK
new_data = new_data & MASK;   // Clear data bits not set to one in MASK
temp = temp | new_data;       // Combine the existing bits with new bits
LATG = temp;                  // Write the modified bits to port G
```

A more concise sequence of code to accomplish the same result can be written as:

temp = LATG & ~MASK;   // Read all 16 port G bits and clear MASK bits
LATG = temp | (new_data & MASK;) // Write the modified bits to port G

For as more detailed explanation of the read-modify-write issue, refer to this web link:
http://www.mikroe.com/download/eng/documents/compilers/mikroc/pro/pic/help/rmw.htm

ODC:

Open Drain Control causes the I/O pin high state to be high impedance. The most common uses of open drain output is to accommodate an interface with a device that operates on different voltage levels and to tie multiple outputs together in a wired AND configuration. All PIC32 I/O pins have individual open drain control. An example use of the ODC is:

ODCGSET = BIT_13;;

PORT:

To read from a specific port you use the PORT command. The PORT for each register stores the state of the register at a given time. You can assign this value to a variable to check conditions such as:

```
unsigned int x;
x = PORTG;
```

This instruction assigns the current value of PORT G to the variable 'x'.

To write to a port, use the PORT value to assign a value to the port using the instruction:

```
PORTG = 0x1234;
```

This instruction will result in the same output as does the instruction:

```
LATG = 0x1234;
```

In reality, as shown in Figure 2, the WR PORT and WR LAT control both set the LAT requester, while the RD PORT control reads the state of the processor pin. Thus, assignments to LATG and PORTG are equivalent.

NOTE: The documentation for the Cerebot MX7cK processor boards refers to combinations of Pmod connectors as JA, JB, JC, etc. These letters do not necessarily correspond with the port names and pins of the PIC32MX7 processor nor are consecutive I/O pins assigned to the Pmod connectors. To learn more about the difference refer to the Cerebot MX7cK reference manual. For example, the pin assignment for Pmod connector JA is shown in TABLE I.

Table II. PIC32 port assignments for Cerebot MX7cK Pmod connector JA

JA Pin number	MCU Port Bit	PIC32 Signal Name

JA-01	RB02	AN2/C2IN-/CN4/RB2
JA-02	RB03	AN3/C2IN+/CN5/RB3
JA-03	RB04	AN4/C1IN-/CN6/RB4
JA-04	RB06	PGEC2/AN6/OCFA/RB6
JA-05	GND	
JA-06	VCC	
JA-07	RB07	PGED2/AN7/RB7
JA-08	RB08	AN8/C1OUT/RB8
JA-09	RB09	AN9/C2OUT/RB9
JA-10	PB10	CVrefout/PMA13/AN10/RB10
JA-11	GND	
JA-12	VCC	

Appendix B. Peripheral Library I/O Functions

Some I/O pins can support multiple peripheral resources. Although the preferred approached that will be used throughout this and following projects will be to use the peripheral library functions whenever possible, the following equivalent statements are shown in the event that the reader encounters these alternate forms. Pins can that can be configured to be either analog inputs or digital I/O must be set accordingly. (See page 9 of the PIC32 Family Data Sheet to determine the pins that can be configured as either analog or digital inputs) Section 10 of the Microchip C32 Peripheral Library contains functions to assist in setting the direction of the PIC32MX7 I/O pins. (PIC32 Family Reference Guide, Section 12.3.1 specifically states that the default to analog inputs on power up reset. Hence the use of the AD1PCFGSET instruction in conjunction with the TRISx, TRISxSET, TRISxINV, and TRISxCLR instructions is required.) The following two functions may be used in place of setting the port TRIS registers to set the pin directions.

1. To set the I/O pins as digital inputs, use the function:
   void PORTSetPinsDigitalIn(IO_PORT_ID port, unsigned int inputs);

   The port to be set is identified by the first variable and is specified by the identifiers IOPORT_x, where x is A through G.

   The second variable passed to this function is the pin values for those pins to be set as outputs. It is possible to set the pins individually by combining the pins with the bitwise or function using the bit constants BIT_0 through BIT_15 that are predefined in the plib.h system header file.

2. To set the I/O pins as outputs, use the function:
   void PORTSetPinsDigitalOut(IO_PORT_ID port, unsigned int outputs);

   The port to be set is identified by the first variable and is specified by the identifiers IOPORT_x where x is A through G.

   The second variable passed to this function is the pin values for those pins to be set as outputs. It is possible to set the pins individually by combining the pins with the bitwise or function using the predefined bit constants BIT_0 through BIT_15.

3. The following examples illustrate equivalent alternate methods of implementing I/O pin operations.

```
/* Configuring Port B pin 10 as an output */
        PORTSetPinsDigitalOut(IOPORT_B, BIT_10);

        /* The following two instruction are both required to accomplish the same
        * configuration */

        AD1PCFGSET = BIT_10;      // Set for digital I/O
        TRISBCLR = BIT_10;        // Set direction

/* The following statements are equivalent and result in setting Port B, pin 10 output high
*/
        PORTSetBits(IOPORT_B, BIT_10);         // Provided by Peripheral Library
        mPORTBSetBits(BIT_10);                 // Provided by Peripheral Library
        LATBSET = BIT_10;
        PORTBSET = BIT_10;
        LATB = LATB | BIT_10;

/* The following statements are equivalent and result in setting Port B, pin 10 output low
*/
        PORTClearBits(IOPORT_B,BIT_10);        // Provided by Peripheral Library
        mPORTBClearBits(BIT_10);               // Provided by Peripheral Library
        LATBCLR = BIT_10;
        PORTBCLR = BIT_10;
        LATB = LATB & ~BIT_10;

/* The following statements are equivalent and result in writing Port B, pin 10 high and all
other pins low */
        LATB = 0x0400;
        LATB = BIT_10;
        PORTB = 0x0400;
        PORTB = BIT_10;

/* The following statements are equivalent and result in configuring Port B pin 10 as a
digital input */
        PORTSetPinsDigitalIn(IOPORT_B, BIT_10);

        //The following two instruction are both required
        AD1PCFGSET = BIT_10;      // Set for digital I/O
        TRISBSET = BIT_10;        // Set direction

/* The following statements are equivalent and result in reading all Port B I/O pins */

        (int) pins = PORTB;
        (int) pins = PORTRead(IOPORT_B);       // Provided by Peripheral Library
        (int) pins = mPORTBRead();             // Provided by Peripheral Library
```

```
/* The following statements are equivalent and result in reading only Port B pin 10 */
        (int) pin10 = PORTB & BIT_10;
        (int) pins = PORTReadBits(IOPORT_B, BIT_10);    // Provided by Peripheral Library
        (int) pins = mPORTBReadBits(BIT_10);            // Provided by Peripheral Library
```

Appendix C. Detailed I/O PIN Block Diagram

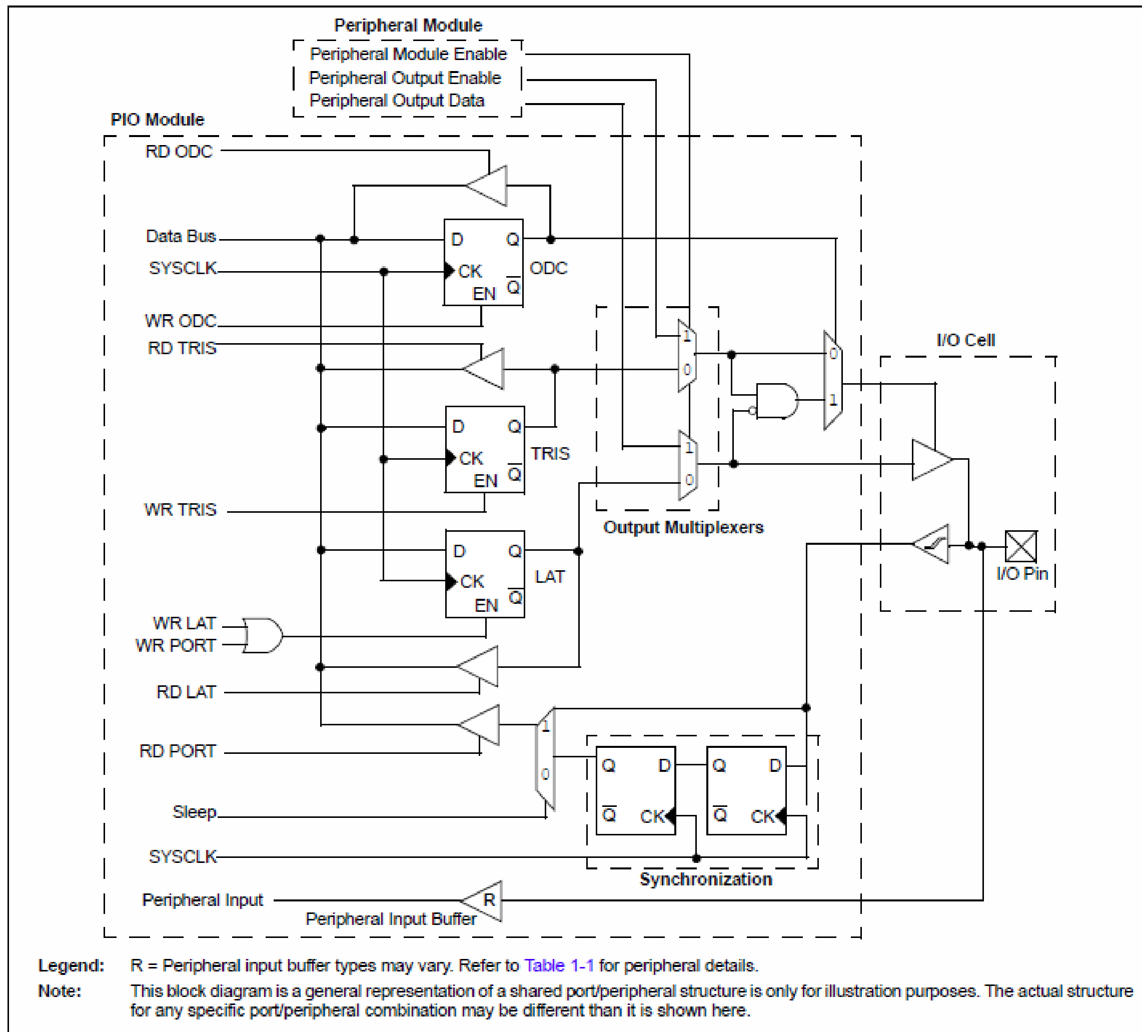

Figure 2. Block diagram of a typical multiplexed port structure (PIC32 Users Guide).

Appendix D. Software Models and Planning

Software models are an abstraction of the functionality of a computer program. The common software abstractions are the data flow diagram (DFD), the control flow diagram (CFD) -- more commonly called a flow chart, and the finite state machine diagram (FSM). An indepth discussion of finite state machines is provided in Project 3.

Data flow diagrams convey how the problem is partitioned into single-purpose blocks of code or functions that complete a single action and the information that is exchanged between the partitions. A data flow diagram for the proposed design in this project is shown in Figure 3. Each block represents a task, has a name describing the functionality and the software function name in parenthesis. The heavy lines between blocks represent associations. The smaller vectored line represents data and the direction of flow. For example, the function “buttons” collects button status data and communicates that to the function “main”. A well designed and comprehensive DFD allows the software developer to construct a program shell without ever writing a line of operational code.

For example, from Figure 3 we know the program will consist of five functions, *main*, *syst_init*, *buttons*, *map*, and *leds*. We also know that the function sys_init is called by function main, has no information passed to it and no information returns from it. Hence the prototype for the functions sys_init is:

void sys_init(void); // Initialize PIC32 for project 1

Similarly, we see that the function *map* is also called from *main*. This function receives button status information and returns LED control information. The prototype for the *map* function is:

int map(int buttons); // Determine which LED to turn on based on button status

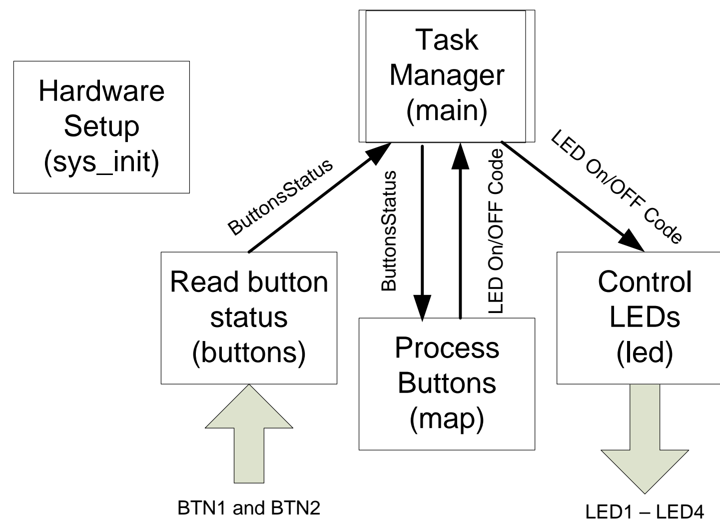

Figure 3. Example data flow diagram

Control flow diagrams convey the execution order of the control process, as illustrated in Figure 4. Very effective control flow diagrams can be generated using four fundament shapes. Boxes represent operations or processes, circles show where code execution paths join, diamonds show where execution paths diverge based upon a decision, and directed lines depict the control paths. A properly drawn control flow diagram constrains the program to be exclusively at one point on the graph at any given time. The control flow diagram describes the logic the controls to order in which the functions are executed. Figure 4 shows groupings of logic that correlate to the tasks depicted in the DFD.

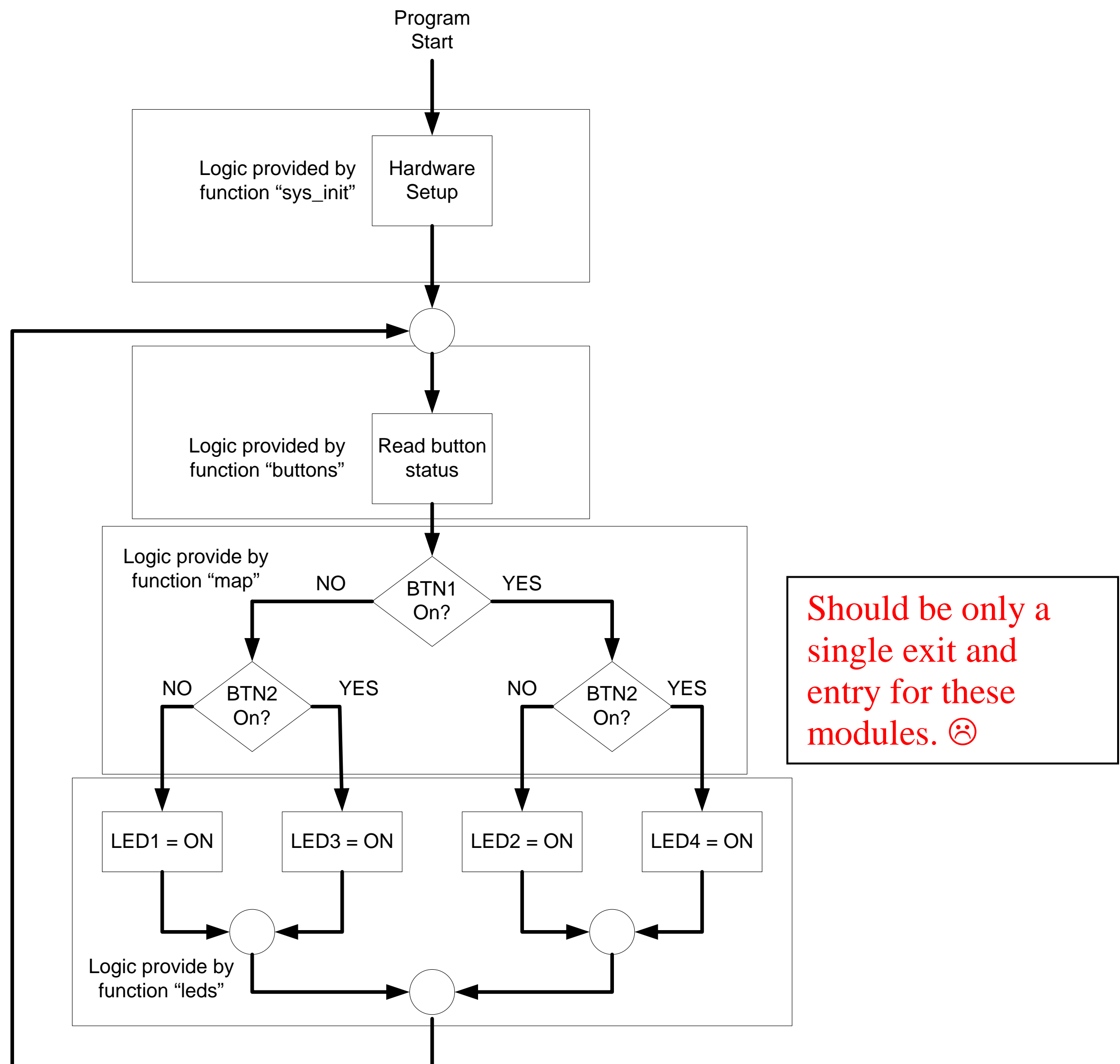

Figure 4. Example control flow diagram. (flattened)

After completing the DFD and CFD for a project or subset of a project, a basic outline of the code can now be generated. Starting with the code provided in Listing 1, we can begin to construct the outline structure of the program needed to complete Project 1. Normally, programs follow a standard order. First, provide comments that describe information concerning the purpose of the program, the author's name and data that the program was created. This is followed by a list of file that that need to be included so the compiler can resolve function declarations, references and definitions that are not provided in the current file. Next, the function prototypes that instruct the compiler how to setup the interfaces between functions are contained in this file. This is followed by a listing of global definitions and global variable

declarations if any are required. Listing 2 is an example of the result of following this program development process.

Listing 2.

```
/************************************** Project1.c **************************************************
 *
 *        This program controls the off/on status of the four LEDs on the Cerebot MX7cK based
 *        on which combinations of buttons 1 and 2 are pressed.
 *
 *        Author:        Richard Wall
 *        Date:          July 5, 2013
 *
*****************************************************************************************************/

#include <plib.h>
#include "CerebotMX7cK.h"

void sys_init(void);       // Initialize PIC32 for project 1
int buttons(void);         // Determine button status
int map(int buttons);      // Determine which LED to turn on based on button status
void leds(int led_ctrl);   // Turn LEDs on or off

int main(void)
{
int button_status, led_ctrl;

        sys_init()

        /*  Project initialization code goes here */

        while(1)
        {
                /* Project operational code goes here */
                button_status = buttons();
                led_ctrl = map(button_status);
                leds(led_ctrl);
        }
        return 1;        /* This line of code will never be executed. */
}

void sys_init(void)      /* Initialize PIC32 for project 1 */
{

        Cerebot_mx7cK_setup();      /* See the function Cerebot_mx7cK_setup() in
                                  CerebotMX7cK.c for details of configuration of the processor board  */

        /* Add additional code to implement task initialization as needed */
}
```

```
int buttons(void)        // Determine button status
{
        /* Add code to implement task */
}

int map(int buttons)    /* Determine which LED to turn on based on button status */
{
        /* Add code to implement task */
}

void leds(int led_ctrl)  /* Turn LEDs on or off */
{
        /* Add code to implement task */
}
/* End of Project1.c */
```

Listing 3. Project 1 header file

```
/************* Project 1 ***************
 *
 * File name:          Project1.h
 * Author:             Richard W. Wall (C)opyright 2011
 * Rev Date:           Aug. 1, 2012
 * Rev Date:           Aug, 28, 2013
 *
****************************************/

#ifndef __PROJECT_1_H__
   #define __PROJECT_1_H__
#endif

/* Function prototypes */
void initialize_system();
int read_buttons(void);
int decode_buttons(int buttons);
void control_leds(int leds);

/* End of Projec1.h */
```

After completing the code as required for Listing 2, the project can be compiled and checked for syntax errors. At this point in time, the program will not generate any results because the execution details have not been added. Once the structure of the program has been defined, functionality details for each function is added and tested. Add code to one function at a time. Make certain that all compiler errors are resolved before you start working on the next function. Start with functions that generate data (inputs) and progress through to functions that use the data (outputs). If possible, make the development order follow the order described in your flow diagram.

References for Code Organization:

Although it is beyond the scope of this project to teach programming styles and techniques, using a consistent coding style is critical to good software development practices and result in fewer errors and less development time. Below are some suggested references to assist you in generating quality software.

1. http://software-dev.blogspot.com/2011/03/code-organization.html
2. http://ece.uidaho.edu/ee/classes/ECE340/Lecture_Notes/L1/Code%20Format.pdf.
3. http://en.wikipedia.org/wiki/Best_Coding_Practices