Analog to Digital Conversion and Using Your Joystick – A Hardware and Software Primer – revised Oct. 22, 2015

The Joystick

The joystick has five pins of interest:

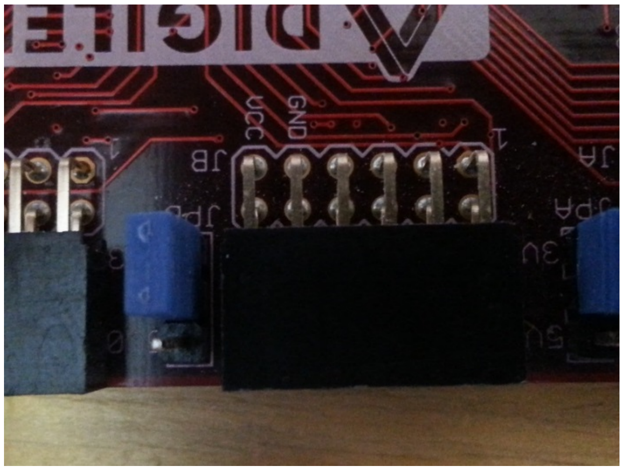
- L/R+: This is the high reference voltage for the potentiometer whose output voltage provides information on the left and right movement of the joystick.
- U/D+: This is the high reference voltage for the potentiometer whose output voltage provides information on the up and down movement of the joystick.
- L/R: This is the **actual output voltage** that provides information on the left and right movement of the joystick. This is one of the two values that we will supply as inputs to the ADC. There are actually two such pins, but we only need to connect one of them.
- U/D: This is the **actual output voltage** that provides information on the up and down movement of the joystick. This is one of the two values that we will supply as inputs to the ADC. There are actually two such pins, but we only need to connect one of them.
- GND: This is the low reference voltage for the potentiometers. The joystick uses a common ground for both potentiometers, so even though there are two GND pins, we only have to connect one.

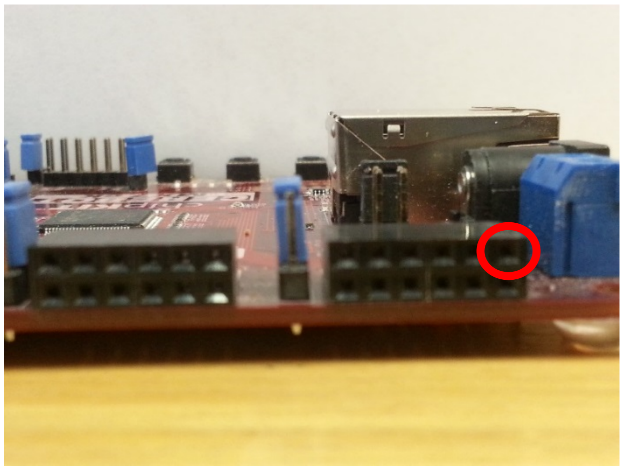
A potentiometer is just a variable resistor. By moving the joystick along either of its axes, you are modifying the resistance in a voltage divider. One of the two voltages in each of the two voltage dividers is the one that is being "reported" as L/R or U/D. The output voltages independently range between the low (usually GND) and high reference voltages (whatever you supply from your Cerebot, probably 3.3V) as you move from "left" to "right" or from "down" to "up" respectively.

Rather than take a picture, I'm going to recommend that you look at your own joystick to familiarize yourself with the pin placement. There is a schematic and documentation on the Scholar site, but for the most part it will just repeat things that I am describing here.

The Cerebot Connector

Next we turn to the Cerebot connector: Here are pictures showing top-down and head on views of the connectors – but once again, it would be a good idea to look at your own Cerebot:





In the picture on the left, the "1" on the right labels the back pin, which corresponds to the top row of the connector. The pins continue in numerical order from right to left across the top row, then from right to left across the bottom row. Notice that the two pins on the left are labeled "GND" and "VCC" – this actually applies to the pins in the top row and the bottom row.

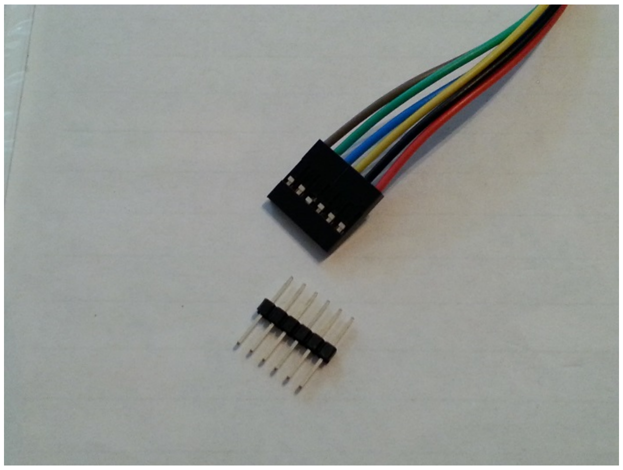
In the picture on the right, I have labeled the position of pin 1 with a red circle. Looking at the connector from this view, the pins have the following positions:

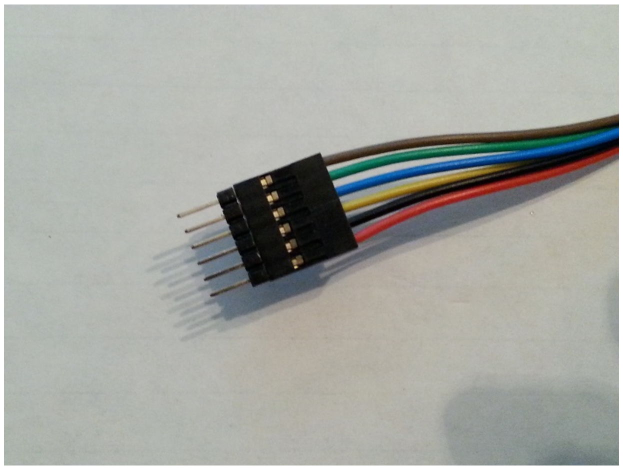
6	5	4	3	2	1
12	11	10	9	8	7

So putting the information in both photos together, pin 5 and 11 are GND pins, and pins 6 and 12 are VCC pins.

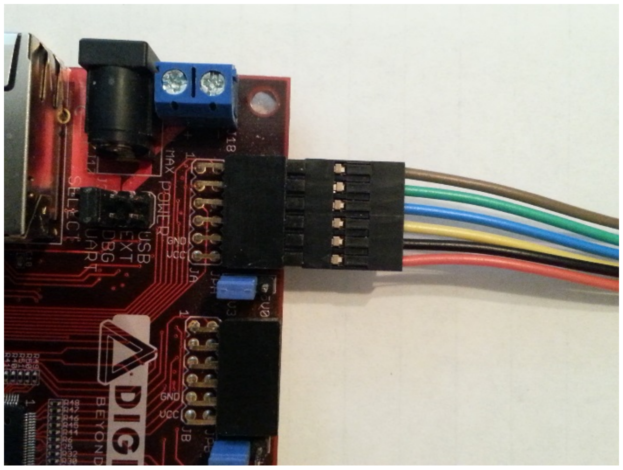
<u>The Medusa Cable</u>

The picture on the left shows the medusa cable, along with female-to-male converter. The picture on the right shows the cable with the converter inserted.





Remember that by convention, red is power and black is ground. This is convenient, because the positions of power and ground on the cable correspond to the positions of power and ground on the connector.



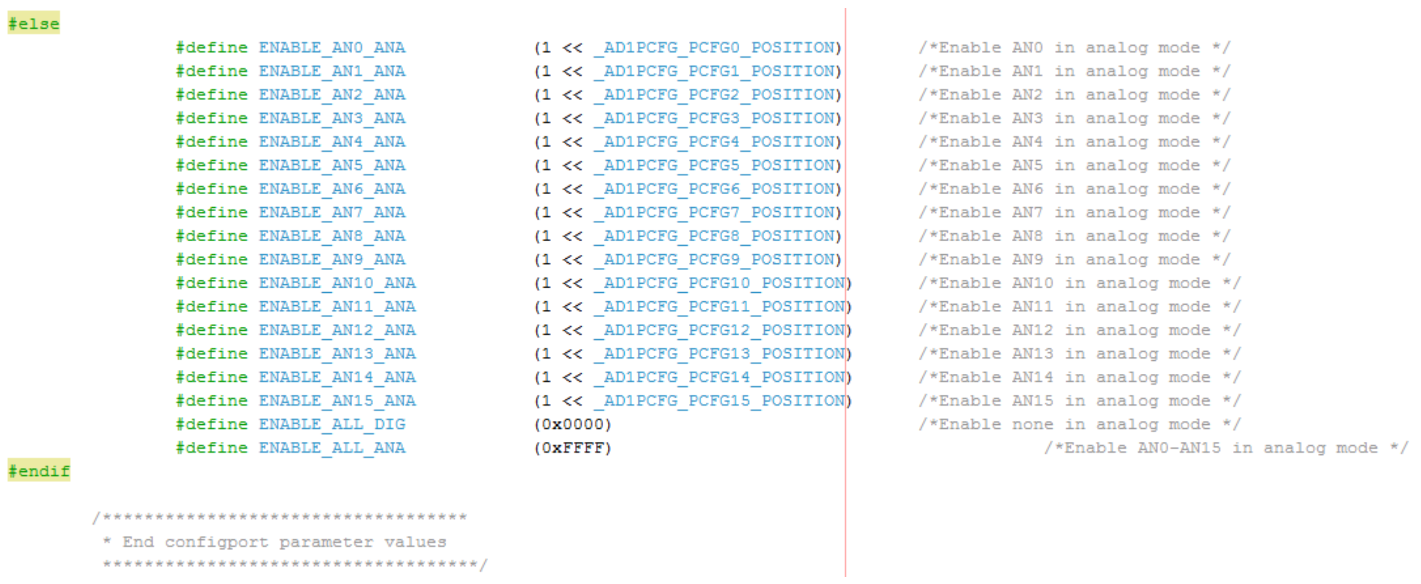
This picture shows the cable connected to the top row of a Cerebot connector. This means that the brown wire is connected to pin 1, the green wire is connected to pin 2, the blue wire is connected to pin 3, and the yellow wire is connected to pin 4. Of course, the black wire is connected to GND (pin 5) and the red wire is connected to pin 6 (VCC).

This is important because we have to figure out which connecter pins represent the input channels of the ADC. You can find this information in the manual.

Using the ChipKIT Pro MZ7 Manual

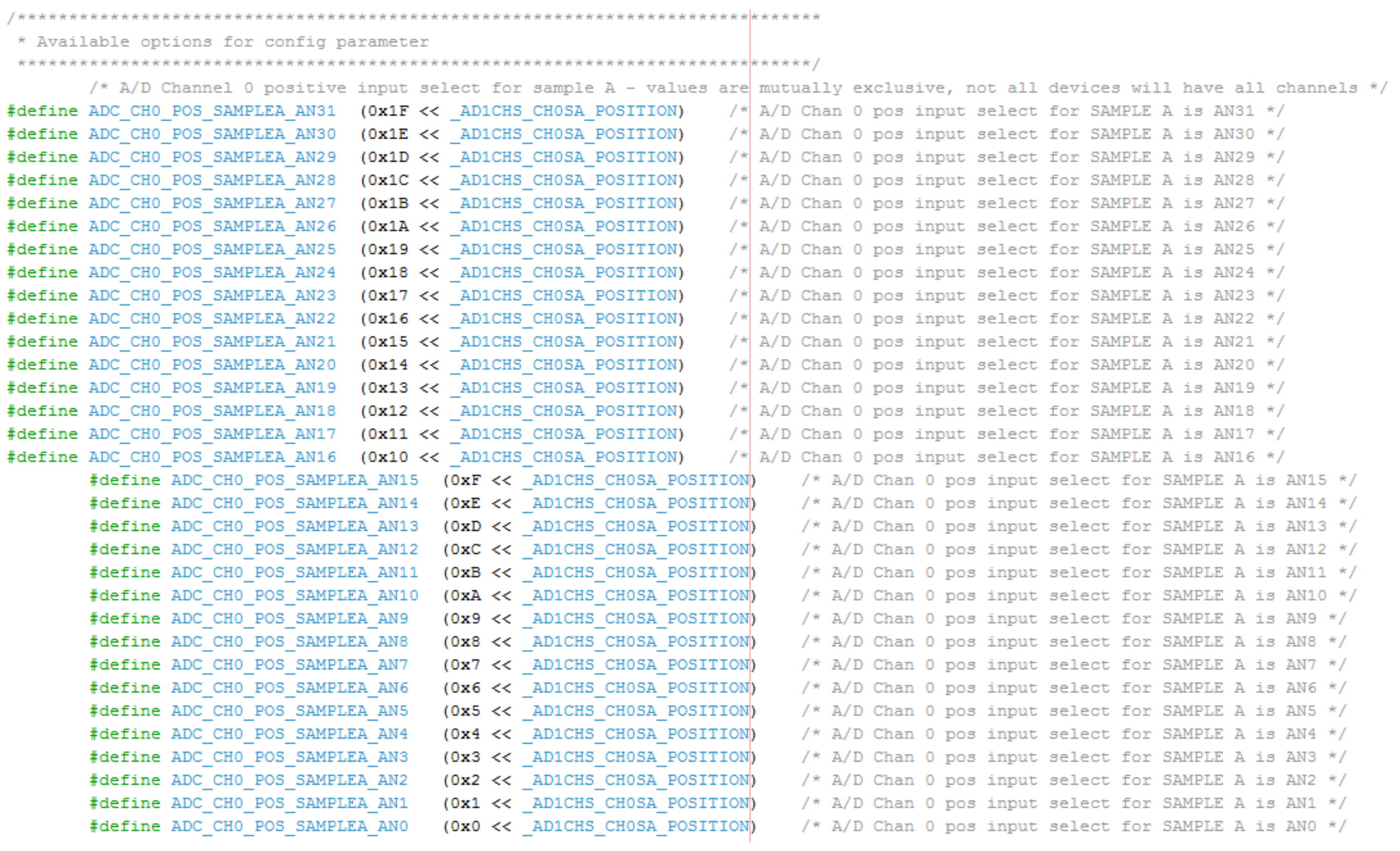
Section 12 of the Manual covers Analog Inputs, and even has a small but helpful section covering certain aspects of the ADC. The problem is that the names that Section 12 uses for the Analog Inputs **are not the same as the names that the Peripheral Library uses to describe its analog input channels.**

Here is a screenshot from the peripheral library file for the ADC. This one happens to show *configport* parameters for the OpenADC10 function:



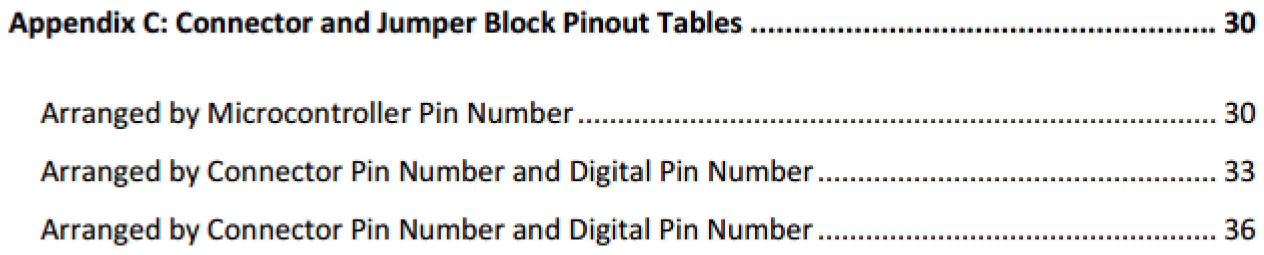
```
#else
                #define ENABLE_AN0_ANA          (1 << _AD1PCFG_PCFG0_POSITION)      /*Enable AN0 in analog mode */
                #define ENABLE_AN1_ANA          (1 << _AD1PCFG_PCFG1_POSITION)      /*Enable AN1 in analog mode */
                #define ENABLE_AN2_ANA          (1 << _AD1PCFG_PCFG2_POSITION)      /*Enable AN2 in analog mode */
                #define ENABLE_AN3_ANA          (1 << _AD1PCFG_PCFG3_POSITION)      /*Enable AN3 in analog mode */
                #define ENABLE_AN4_ANA          (1 << _AD1PCFG_PCFG4_POSITION)      /*Enable AN4 in analog mode */
                #define ENABLE_AN5_ANA          (1 << _AD1PCFG_PCFG5_POSITION)      /*Enable AN5 in analog mode */
                #define ENABLE_AN6_ANA          (1 << _AD1PCFG_PCFG6_POSITION)      /*Enable AN6 in analog mode */
                #define ENABLE_AN7_ANA          (1 << _AD1PCFG_PCFG7_POSITION)      /*Enable AN7 in analog mode */
                #define ENABLE_AN8_ANA          (1 << _AD1PCFG_PCFG8_POSITION)      /*Enable AN8 in analog mode */
                #define ENABLE_AN9_ANA          (1 << _AD1PCFG_PCFG9_POSITION)      /*Enable AN9 in analog mode */
                #define ENABLE_AN10_ANA         (1 << _AD1PCFG_PCFG10_POSITION)     /*Enable AN10 in analog mode */
                #define ENABLE_AN11_ANA         (1 << _AD1PCFG_PCFG11_POSITION)     /*Enable AN11 in analog mode */
                #define ENABLE_AN12_ANA         (1 << _AD1PCFG_PCFG12_POSITION)     /*Enable AN12 in analog mode */
                #define ENABLE_AN13_ANA         (1 << _AD1PCFG_PCFG13_POSITION)     /*Enable AN13 in analog mode */
                #define ENABLE_AN14_ANA         (1 << _AD1PCFG_PCFG14_POSITION)     /*Enable AN14 in analog mode */
                #define ENABLE_AN15_ANA         (1 << _AD1PCFG_PCFG15_POSITION)     /*Enable AN15 in analog mode */
                #define ENABLE_ALL_DIG          (0x0000)                            /*Enable none in analog mode */
                #define ENABLE_ALL_ANA          (0xFFFF)                                        /*Enable AN0-AN15 in analog mode */
#endif

        /************************************
         * End configport parameter values
         ************************************/
```

Here is another screen shot from the same peripheral library file. This one shows *config* parameters for the SetChanADC10 function:



```
/***********************************************************************************
 * Available options for config parameter
 **********************************************************************************/
        /* A/D Channel 0 positive input select for sample A - values are mutually exclusive, not all devices will have all channels */
#define ADC_CH0_POS_SAMPLEA_AN31  (0x1F << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN31 */
#define ADC_CH0_POS_SAMPLEA_AN30  (0x1E << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN30 */
#define ADC_CH0_POS_SAMPLEA_AN29  (0x1D << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN29 */
#define ADC_CH0_POS_SAMPLEA_AN28  (0x1C << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN28 */
#define ADC_CH0_POS_SAMPLEA_AN27  (0x1B << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN27 */
#define ADC_CH0_POS_SAMPLEA_AN26  (0x1A << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN26 */
#define ADC_CH0_POS_SAMPLEA_AN25  (0x19 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN25 */
#define ADC_CH0_POS_SAMPLEA_AN24  (0x18 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN24 */
#define ADC_CH0_POS_SAMPLEA_AN23  (0x17 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN23 */
#define ADC_CH0_POS_SAMPLEA_AN22  (0x16 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN22 */
#define ADC_CH0_POS_SAMPLEA_AN21  (0x15 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN21 */
#define ADC_CH0_POS_SAMPLEA_AN20  (0x14 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN20 */
#define ADC_CH0_POS_SAMPLEA_AN19  (0x13 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN19 */
#define ADC_CH0_POS_SAMPLEA_AN18  (0x12 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN18 */
#define ADC_CH0_POS_SAMPLEA_AN17  (0x11 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN17 */
#define ADC_CH0_POS_SAMPLEA_AN16  (0x10 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN16 */
        #define ADC_CH0_POS_SAMPLEA_AN15  (0xF << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN15 */
        #define ADC_CH0_POS_SAMPLEA_AN14  (0xE << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN14 */
        #define ADC_CH0_POS_SAMPLEA_AN13  (0xD << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN13 */
        #define ADC_CH0_POS_SAMPLEA_AN12  (0xC << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN12 */
        #define ADC_CH0_POS_SAMPLEA_AN11  (0xB << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN11 */
        #define ADC_CH0_POS_SAMPLEA_AN10  (0xA << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN10 */
        #define ADC_CH0_POS_SAMPLEA_AN9   (0x9 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN9 */
        #define ADC_CH0_POS_SAMPLEA_AN8   (0x8 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN8 */
        #define ADC_CH0_POS_SAMPLEA_AN7   (0x7 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN7 */
        #define ADC_CH0_POS_SAMPLEA_AN6   (0x6 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN6 */
        #define ADC_CH0_POS_SAMPLEA_AN5   (0x5 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN5 */
        #define ADC_CH0_POS_SAMPLEA_AN4   (0x4 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN4 */
        #define ADC_CH0_POS_SAMPLEA_AN3   (0x3 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN3 */
        #define ADC_CH0_POS_SAMPLEA_AN2   (0x2 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN2 */
        #define ADC_CH0_POS_SAMPLEA_AN1   (0x1 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN1 */
        #define ADC_CH0_POS_SAMPLEA_AN0   (0x0 << _AD1CHS_CH0SA_POSITION)    /* A/D Chan 0 pos input select for SAMPLE A is AN0 */
```

The name that the Cerebot literature uses to describe ADC channels is AN*x*, where *x* is one of the sixteen channels numbered 0 through 15. (See the schematics in the lecture slides for a representation of how these inputs interact with the ADC. To figure out where to find input pins for these signals on the connectors, we have to make our first trip to the Cerebot manual's Appendix C: *Connector and Jumper Block Pinout*:



(The table of contents contains a misprint. The table that begins on Page 36 actually arranges the pins by Microcontroller Port and Bit Number.)

The last two tables are arguably the most useful. Sometimes you will want to know how to find a signal based on its placement on a Cerebot connector. Other times you will want to find a signal based on the GPIO port and bit number that interface to the same signal. Both tables present both pieces of information; this is also useful, because usually once you know one of the two, you will want to know the other.

You will probably find it useful to search the (actual) table that arranges the pinout by Connector Pin Number and Digital Pin Number; this table begin Page 33. You are trying to find a connector that contains ADC channel inputs. Remember, these are designated as AN*x*. Ideally, you will want to find a connector that has ADC channels on the same row of pins. That way, you can connect your medusa cable to that row of the connector, and have the non-VCC, non-GND wires each provide access to a channel. If you use the information in this table correctly, you will simultaneously know *where to connect your cable to the Cerebot* and *which wires of the cable correspond to the ADC channels you will need to use.*

<u>Making the Final Connections</u>
Note: This section contains an exercise or two for the reader. Look sharp!

Once you have determined which connector to use and have connected your medusa cable to that row of connector pins, you will know which wire corresponds to which ADC input channel. Now we must connect the wires to the joystick. Remember, we care about five pins on the joystick, so we will need to use five wires on the cable.

- The black wire is GND. Even though the joystick has two ground pins, both potentiometers use a common ground. This pin is no problem; connect it to *either* of the GND pins on the joystick.
- The red wire is VCC. The joystick has two pins that must receive VCC: L/R+ and U/D+. One option is to connect the joystick pins with a jumper, and then connect the red wire to either of the (now-connected) pins. I don't recommend this. For now, choose one of the two high-reference pins of the joystick and connect the red wire to that pin.
- *If you have chosen correctly, each of the four remaining wires corresponds to an ADC analog channel.* You will only need two of them. You must connect one of them to L/R and the other to U/D. The choice is not entirely arbitrary – you can choose any two you like, but the two that you choose must end up being the two that you configure when you write your ADC configuration software. Connect one of the two wires you chose to each of the two joystick pins L/R/ and U/D. *You should now take note of which ADC channel is receiving L/R, and which one is receiving U/D*.

- You have two wires left over, and you still need a VCC source for the other high-reference pins. Recall that certain connector pins correspond to GPIO ports and bits. In the same way that we have written GPIO code to light the LEDs, we can write code to configure a bit of an appropriate bit as an output, and then use the wire that is connected to the corresponding connector pin to drive the other high-reference. *Consult the table that you used to determine where to connect your medusa cable. Determine which GPIO ports and bits interface to the two wires that remain unused. Choose one of them, and write code to 1) configure the proper GPIO port and bit as an output, and 2) drive that bit high. Connect the wire carrying that logic-high output to the high-reference pin of the joystick that you have not connected.*

That's it for hardware! Once you have configured the ADC correctly in software, you should have a working interface between the joystick and the Cerebot. This brings us to…

<u>Configuring the ADC using the Peripheral Library Functions</u>
Of the many Cerebot peripherals that I have used, the ADC is probably one of the few that is harder to configure than it is to use. This is because there are a number of configuration functions, and one of them in particular requires many parameters.

Let's start by reviewing what we know about configuration. As it was for the Timer, what we are doing is "building" values for placement into various control registers. But rather than deriving the values and placing them into the registers based on our knowledge of what each bit of a register does and what the register addresses are, we use labels that correspond to the values (generally established via *#define* or *typedef*) to build the values and the peripheral library functions themselves to place the values that we have built.

We build the values by using OR-separated "lists" of these labels. Here is an example from a Timer2 initialization function that I "wrote" – note that all this function does is to call the appropriate timer PLIB function, which then uses a parameter for the period register value that user passes via the function call:

```
void initTimer(int count)
{
      // Set TMR2 frequency = 10MHz/[16*(count + 1)] = [625/(count + 1)] kHz.
      OpenTimer2(T2_ON|T2_IDLE_CON|T2_SOURCE_INT|T2_PS_1_16|T2_GATE_OFF, count);
}
```

This is the structure of OpenTimer2:

```
      OpenTimer2(config, period);
```

If we read enough about the configuration options of Timer2, we can determine that the above function turns on Timer2, allows it to continue running when the processor is idle, uses the internal clock as the timer's source clock, establishes a prescale value of 16:1, and turns off gated timer mode.

This is what we need to do for the ADC. Ideally, we would like to come up with the structure of a function to initialize the ADC. Let's call this function initADC. Prototyped, initADC might look like this:

```
void initADC(void);  // This should indicate that we plan to pass nothing to
                     // initADC, and to have it return nothing.
```

So what does initADC need to do? There are three functions of interest to us in the ADC peripheral library that pertain to configuration. (Actually, the functions are macros created via #define, but that distinction should matter little to us.) Our function initADC should call each of these "functions" in the correct order. Since initADC is a void function, we will likely end up hard-coding all of the parameters in the functions that initADC calls. Let's focus on each function, one at a time.

`SetChanADC10(config)`

In SetChanADC10, we are setting up the channels that the ADC will accept as inputs. Any of the 16 channels can be connected to either of two multiplexers. We can configure the multiplexers to alternate in providing values to the ADC circuit. Since we have two channels that we want to sample, it makes some sense to configure both MUXA and MUXB to receive one of the two channels.

Each of the multiplexers must be configured to supply a "positive" and a "negative" input to the sample-and-hold circuit that supplies the ADC. *In both cases, the positive inputs are the analog input channels that we want to convert. The negative inputs are the low-reference voltage of the ADC. Your SetChanADC10 function should have four parameters in its OR-separated list. Make sure that you provide only one of the channels as SAMPLEA, and the other one as SAMPLEB.*

`OpenADC10(config1, config2, config3, configport, configscan)`

In OpenADC10, we are configuring the ADC. This function has five parameters, and four of the five represent potentially extensive OR-separated lists. As you review the configuration options described below, you might want to have the OpenADC10 function open. The configuration options for OpenADC10 are conveniently located below the function.

(In general, Peripheral Library functions include parameters that contribute values of zero to the control value that we are "building." Sometimes, it is better to be safe and include these parameters, even though leaving them out would have no impact on the configuration. While I will make an effort to clearly describe parameters that you must use, you might note that some of the parameters I describe are ones that contribute zero values. In one or two cases, I will point out items that you must make sure you ***don't*** *include; in these cases, not including these items is as easy as leaving them off the list.)*

- `config1`

  We want to use config1 to make the digital output of the ADC an unsigned integer (**not** a signed integer), use Timer3 period match to end sampling and start conversion (not INT0, which refers to the external INT0 pin), and enable auto-sampling.

  We do not want to use *OpenADC10* ***to turn on the ADC! We have another function for that.***

- `config2`

  We want to use config2 to make sure that the ADC uses its internal VDD and VSS as its voltage references, takes two samples before triggering an interrupt request, divides its output buffer into two 8-word buffers, and alternates between MUXA and MUXB when taking samples. We won't be enabling offset calibration or turning on auto-scan mode.

- `config3`

  I have not noticed a difference between using config3 to establish the ADC internal clock as the clock source and establishing the peripheral bus clock as the clock source. However, the ADC internal clock is better suited for applications where sampling must continue while the processor is idle. The internal clock is not calibrated.

  We want to use config3 to have establish a sampling rate. The sampling rate consists of two components – how many peripheral clock periods represent one time unit and how many such time units represent one sampling period. *The product of the two values represents the sampling period, measured in peripheral clock periods.*

Configure the number of time units in the sampling period by choosing a value for the "Auto Time Sample." Do not choose zero. Configure the number of peripheral clock periods per time unit by choosing a value for the "Conversion Clock." For the a value of 1 time unit = 2 peripheral clock periods (which is the default) I have gotten good results making the sampling period equal to fifteen to twenty time units.

You might want to experiment with these values to see how your joystick handles when it is sampled more frequently or less frequently. The conversion requires a minimum sampling period of 12 time-bases.

- `configport`

  Use configport to enable the ADC input channels that are carrying your L/R and U/D joystick outputs as inputs to the Cerebot.

- `configscan`

  Use configscan to skip all channels during auto-scan. It might make sense that we are going to skip all channels during an auto-scan since we aren't auto-scanning anyway. It might seem senseless to configure this option in light of the fact that we aren't auto-scanning anyway. Either way, you might say we using this option to double down on not auto-scanning any of the channels.

EnableADC10()

Calling this function actually turns on the ADC. In the configuration and enabling sequence, you should call it only after you done all of your configuration.

If you do all of these things correctly, you should be ready to use the ADC.

Interlude: Interrupt Configuration
We are using the ADC as an interrupt-driven device. Specifically, we have configured the ADC to request an interrupt on the occurrence of every second sample that it takes. To ensure that we know when we should be reading the buffer that contains the digital outputs, we must properly configure the interrupt controller to respond to this ADC interrupt. As you did in Problem 2, use int.h and int5xx_6xx_7xx.h to use the right interrupt flags and indicators in the correct interrupt PLIB functions to configure ADC interrupts alongside timer interrupts. Timer interrupts already have a higher natural priority than does the ADC interrupt that indicates that a conversion is complete. Unless you want to change this – and you probably don't – there is no need to get too creative with setting interrupt priorities. If you assign them the same priority, the timer interrupt will still "win" in a "tie."

Reading the ADC output using the Peripheral Library Functions
Since the ADC is interrupt-driven, reading the output will be an element of the ISR. The minimum you will have to accomplish is reading the digital output from the correct buffer. Any other processing that you do with the digital outputs you read should be done in the ISR **if you believe that doing so represents a time-vital task that is sufficiently related to reading the data**. Otherwise, it should be done in main.

One way to make up your mind where to locate the data-processing task is to choose to perform it either in main or in the ISR. If your code causes you to get good results with the joystick, then your choice is probably good. If it doesn't, then you should probably make the other choice.

There are two functions that we need to read the ADC digital output. (They're also #define-established macros, but that's still okay.) One of the two is being used only because we configured the output buffer as two 8-word buffers instead of one 16-word buffer. Since the buffer has been configured in this way, one of the two will be the buffer that the ADC is currently using to save data. Since the ADC is using one of the two buffers to write data, we should make sure that we are reading from the **other one**.

Let’s focus on the two functions of interest.

`ReadActiveBufferADC10()`

ReadActiveBuffer returns a value that indicates which of the two 8-word buffers the ADC is filling with values. In this sense, “active” means *being used by the ADC at the time of the function call*. Since the buffer indicated by the function is being used as the source of the write, we *don’t want to read from it.* Instead, we should read from whichever of the two buffers is *not active*.

ReadActiveBuffer returns 0 if the ADC is currently writing to the buffer containing addresses 0 to 7 (0000 to 0111). It returns 1 if the ADC is currently writing to the buffer containing addresses 8 to 15 (1000 to 1111). *See a correspondence?*

`ReadADC10(bufIndex)`

ReadADC10 reads from the buffer address specified by the argument *bufindex*. This value must correspond to one of the sixteen locations (0 to 15) that the buffer contains. ReadADC10 returns the value read from the address specified as its argument, so you will want to use it as a means of assigning a value to a variable:

```
        digitalValue = ReadADC10(address);
```

When the ADC writes to a buffer in dual-buffer mode, it writes to the first address of that buffer (0 or 8, depending) and continues writing sequentially until it has written the number of samples that triggers an interrupt request. In the case of our configuration, the ADC will have data available in addresses 0 and 1 (the input channel sampled by MUXA will have its digital output in address 0, while the input channel sampled by MUXB will have its digital output in address 1), or it will have data available in addresses 8 and 9 (replace “address 0” and “address 1” respectively with “address 8” and “address 9” in the previous parenthetical description).

So how do we know which buffers perform the read? We should *figure out which buffer is* ***not*** *active, then read from the first two locations of that buffer.* In order, the two reads will provide the value resulting from the conversion of the channel on MUXA, and the value resulting from the conversion of the channel on MUXB.

There’s no need to write a wrapper function to contain this process. Short “naked” code in your ISR to determine the active buffer and read from two addresses in sequence is perfectly fine – in fact, function calls out of the ISR can be a dubious proposition.

If you have done everything here correctly, you will succeed at the task at hand. I have made every effort to describe the process in detail and without skipping any steps. If you have trouble, please try to give me some idea of how far you got (or think you got) in the process. It will be hard to debug any of the code that comes before you see activity based on actually moving the joystick, so you may have to review your steps more than once.