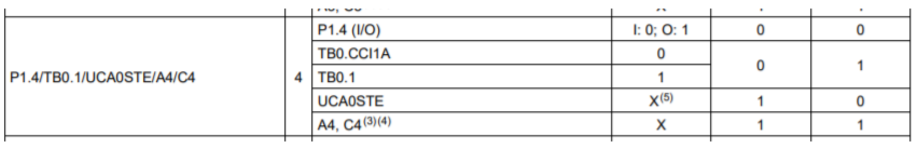
Example 3 – Analog-to-digital Converter

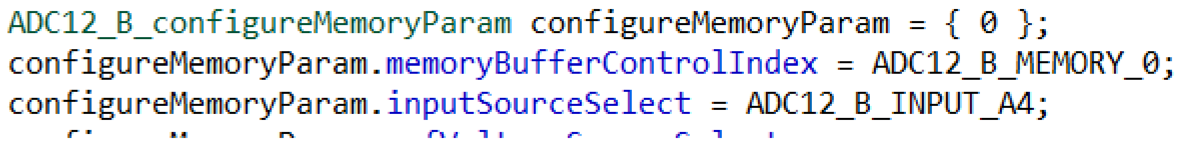
The most popular form of sensor is one that outputs an analog voltage level to represent its data. A light sensor might output 0.01V in darkness and 3.2V in bright light. The analog-to-digital converter (ADC) on the MSP430 samples this voltage level, compares it to a reference such as the 3.3V VCC, and converts it to a digital integer that we can use in our programs. This example project samples one of the analog Grove connectors, turns on LED 1 if the voltage if more than half VCC, and converts the 12-bit integer the ADC outputs to a string format that can be sent to the ESP32.

The example starts the same as the other except for an extra inclusion. The standard input/output library is included, in its minimal form. We use this library to convert the integer value we get from the ADC to a string. Next, we set up the ports. P1.0 (LED 1) is used as an output again, and P1.4 is set as its ternary (third) function. In the datasheet for the MSP430FR5994, we can see that the first peripheral function for P1.4 deals with timer TB0, the second with serial port UCA0, and the third is an analog input.



P1.4/TB0.1/UCA0STE/A4/C4	4	P1.4 (I/O)	I: 0; O: 1	0	0
		TB0.CCI1A	0	0	1
		TB0.1	1		
		UCA0STE	X(5)	1	0
		A4, C4(3)(4)	X	1	1

Next the 12 bit ADC is set up. If you want to find out more about how the ADC is set up, you can look in the driver library files. In summary, we give the ADC an input to sample, a memory to store the value, and reference voltages. We are using port P1.4, which is input A4.



```
ADC12_B_configureMemoryParam configureMemoryParam = { 0 };
configureMemoryParam.memoryBufferControlIndex = ADC12_B_MEMORY_0;
configureMemoryParam.inputSourceSelect = ADC12_B_INPUT_A4;
```

In the main while loop, the delay cycles function is used for delay in place of a timer for simplicity. The conversion is then begun, enabling the interrupt that will trigger when the conversion is completed. A character array is declared to store our string representation of the value. The value is then converted to a string and stored in the buffer we just declared.

Next is the ISR for the ADC. Since many sources can trigger this interrupt, we need to differentiate between them. The different possible sources are commented next to each case in the switch statement, and the case for `MEM0`, which we selected as our memory output, is where our code is. We save the contents of `MEM0` to a global integer variable so that we can access it in the main loop. If the value is greater than `0x7FF` which is half the maximum 12-bit value 2^{12}-1 = 4095, then LED 1 is turned on, otherwise it is turned off.

The last part of the code is the function that will convert the sampled integer to a string (an array of characters terminated will a null character `0x0`). The first step is converting the 16-bit value to a 32-bit value. An unsigned 16-bit integer has a maximum value of 65535 (2^{16}-1), but to get an accurate calculation, we may need to use numbers greater than this maximum, so we need more bits. We then multiply by 100,000 so that it can be divided by 4096. Remember in integer division, the remainder is not considered, so the larger our value is, the more accurate this calculation will be.

Example:

$$value = 3{,}100$$

In normal arithmetic:

$$\frac{3{,}100}{4{,}096} \times 100 = 75.6836\%$$

However, in integer arithmetic, 3100/4096 = 0. If we do the multiplication by 100 first, in integer arithmetic:

$$3{,}100 \times 100 = 31{,}000$$

$$\frac{31{,}0000}{4{,}096} = 75$$

Which is OK, but not as accurate as we might like. To get a more accurate answer, we can multiply by a larger number first:

$$3{,}100 \times 100{,}000 = 310{,}000{,}000$$

$$\frac{310{,}000{,}000}{4{,}096} = 75{,}683$$

This result is much closer! Let’s say we only want two decimal places, but want to keep the precision of a large pre-multiplier. After the division with 4096, we can divide by 10 to get 7,568. Now to get 7,568 to a string we can use the `sprintf` function from stdio.h. This will give us in our buffer:

‘7’	‘5’	‘6’	‘8’	0x0

This is not exactly correct, of course. We need to add a decimal point, and since we know our value will always have two digits after the decimal point, it is fairly simple to add. The last part of the function adds leading zeros if necessary2, then adds the decimal point.