SparkFun Block for Intel® Edison - ADC

The Code:

- This code is in my Github:

Setting up the Block

In order to make this block work, you need to consider several things:

- You can use this block with the Arduino Breakout, the Mini-Breakout or by itself.
- You can use it from python, node, Arduino IDE, C or C++
- Maximum Voltage Level:
  - –0.3 to (3.3) + 0.3
  - I am not sure about this one, check the schematic to be sure: https://cdn.sparkfun.com/datasheets/Dev/Edison/ADC_Block.pdf
- If you need to use the ADC with the Mini-breakout board you will need to cut off the J1 and J2 jumpers and solder the J1pins together.
- The Reference Voltage is configured by software
- If you need to use more than one configuration use the class offered with this sample C++ Library

Version

Make sure that you download the latest Intel Edison Image:

- Intel® Edison Products — Software Downloads

If you don't know how to flash this image you can make use of this short tutorial (Windows, MAC and Linux):

- Intel® Edison Products — Flashing Intel® Edison (wired) - Windows*

If you will use this with the IoTKit Analytics Platform make sure that you are using a the version 1.5.2 or later. In order to know your version and update it follow these steps:

- iotkit-admin -V
- npm update -g iotkit-agent

If you do not have a default installed iotkit-agent run this command:

- npm install iotkit-agent

If your npm is giving you problems try and update it:

- npm install -g npm

Just in case you want to use the MRAA Library (useful to work with the GPIOs, UART, I2C and so on) you can install it with this command. You can communicate with the ADC directly using this library but I used the system commands in order to make it more simple and easier to move to other languages.

- npm install mraa

Compile

To copile the main.cpp file just execute this command:

- g++ main.cpp

This command will create an executable program called "a.out" (default executable target name).
You can change it if you want, follow this reference for more information:

- G++ Compiling

Execute

To execute the program, execute these commands in the linux(yocto) command line (considering your output file is called "a.out"):

- chmod +x a.out
- ./a.out

The Library

Main.cpp

There is a declaration of several variables as ain0_operational status. This variables are used to configure the ADC according to what you want to read from the Analog inputs.
The program has a class named ADC; when you create an ADC object you have to specify its configuration variables. The Default configuration will read only the AN0 comparing it to a Maximum of 4.096V. This main.cpp shows how to use 2 objects to read 2 different analog inputs, you can even declare an object to subtract (compare) these inputs or even change the reference voltage.

The meaning of this values is specified in the Spark_ADC.cpp file and as follows:

* Represents the suggested configuration to only read from A1N0 and the constructor defaults
int os: **Operational status/single-shot conversion start.**
This bit determines the operational status of the device.
This bit can only be written when in power-down mode.
Write status:

- • • • 0x0: No Effect.

- 0x1: Begin single conversion (when in power-down mode). *****

Read status

- 0x0: Device is currently performing a conversion.
- 0x1: Device is not currently performing a conversion.

int imc: Input multiplexer configuration.

These bits configure the input multiplexer.

VIN (AINX minus AINY):

- 0x000: AIN0 - AIN1
- 0x001: AIN0 - AIN3
- 0x010: AIN1 - AIN3
- 0x011: AIN2 - AIN3
- 0x100: AIN0 - GND *****
- 0x101: AIN1 - GND
- 0x110: AIN2 - GND
- 0x111: AIN3 - GND

int pga: Programmable gain amplifier configuration

These bits configure the programmable gain amplifier.

FS:

- 0x000: +-6.144V
- 0x001: +-4.096V *****
- 0x010: +-2.048V
- 0x011: +-1.024V
- 0x100: +-0.512V
- 0x101: +-0.256V
- 0x110: +-0.256V
- 0x111: +-0.256V

int mode: Device operating mode

This bit controls the current operational mode:

- 0x0: Continuous conversion mode. *****
- 0x1: Power-Down single-shot mode.

int rate: Data rate

These bits control the data rate setting:

- 0x000: 128SPS
- 0x001: 250SPS
- 0x010: 490SPS
- 0x011: 920SPS
- 0x100: 1600SPS *****
- 0x101: 2400SPS
- 0x110: 3300SPS
- 0x111: 3300SPS

int comp_mode: Comparator mode

This bit controls the comparator mode of operation.
It changes whether the comparator is implemented as a traditional comparator
or as a window comparator:

• • • • 0x0: Traditional comparator with hysteresis. *****
        • 0x1: Window comparator.

int comp_pol: Comparator polarity

This but controls the polarity of the ALERT/RDY pin:

• • • • 0x0: Comparator output is active low. *****
        • 0x1: ALERT/RDY pin is active high.

int comp_lat: Latching comparator

This bit controls whether the ALERT/RDY pin latches once asserted or clears once
conversions are within the margin of the upper and lower threshold values:

• • • • 0x0: The comparator output is active low. *****
        • 0x1: ALERT/RDY pin is active high.

int comp_que: Comparator queue and disable

These bits perform two functions:

• • • • 0x11: Disable the comparator function and put the ALERT/RDY pin into a high state. *****

Otherwise they control the number of successive conversions exceeding the upper or
lower thresholds required before asserting the ALERT/RDY pin.

• • • • 0x00: Assert after one conversion.
        • 0x01: Assert after two conversions.
        • 0x10: Assert after four conversions

After you specify the configuration bits, you can create an ADC Object.
Within the main 2 ADC objects are created: ain0 and ain1.
In order to configure the object you need to call the function "*set_config_command*".
After the object is configured you can still check the configuration parameters using the function
"*get_config_command*"
In order to read from the ADC using the object configuration the function "*adc_read*" needs to be called.

set_config_command

This function is the one that will set the specific string to configure the ADC according to what is specified. This will receive some binary parameters in this order:

set_config_command(int os, int imc, int pga, int mode, int rate, int comp_mode, int comp_pol, int comp_lat, int comp_que)
An example to declare a value comes as follows:

- int os = 0b0

And the specifications for this value can be checked in this documentation.
You will not need to worry about setting the configuration of the ADC over and over again every time you want to read a different Analog pin or read a different configuration. The configuration string is kept in the object and when you call the "*adc_read*" it will configure it for you before reading it.

adc_read

This command will configure the ADC with the specified parameters; after that, it will read the answer from the ADC, put the reply in the proper order and then return in an integer the value. Keep in mind that you may need to do some math in order to use this data; this is the raw data coming from the ADC but put in order

get_config_command

This function will only return the command that is being used to configure the ADC, it can be helpful to check in real time what you have just configured.

Last Comments

- This library is configured to work with the default factory circuit. It works with the I2C 0x48 address, it can be modified to work with any other address just by changing some lines in the Spark_ADC.cpp. Contact me if you need some guidance
- I could write the Arduino library if you need it. Just ask me
- If you need some help to uderstand how do the objects and classes work, here you can find some quick guide: C++ classes and objects