Using the ADC-001 for the Beaglebone

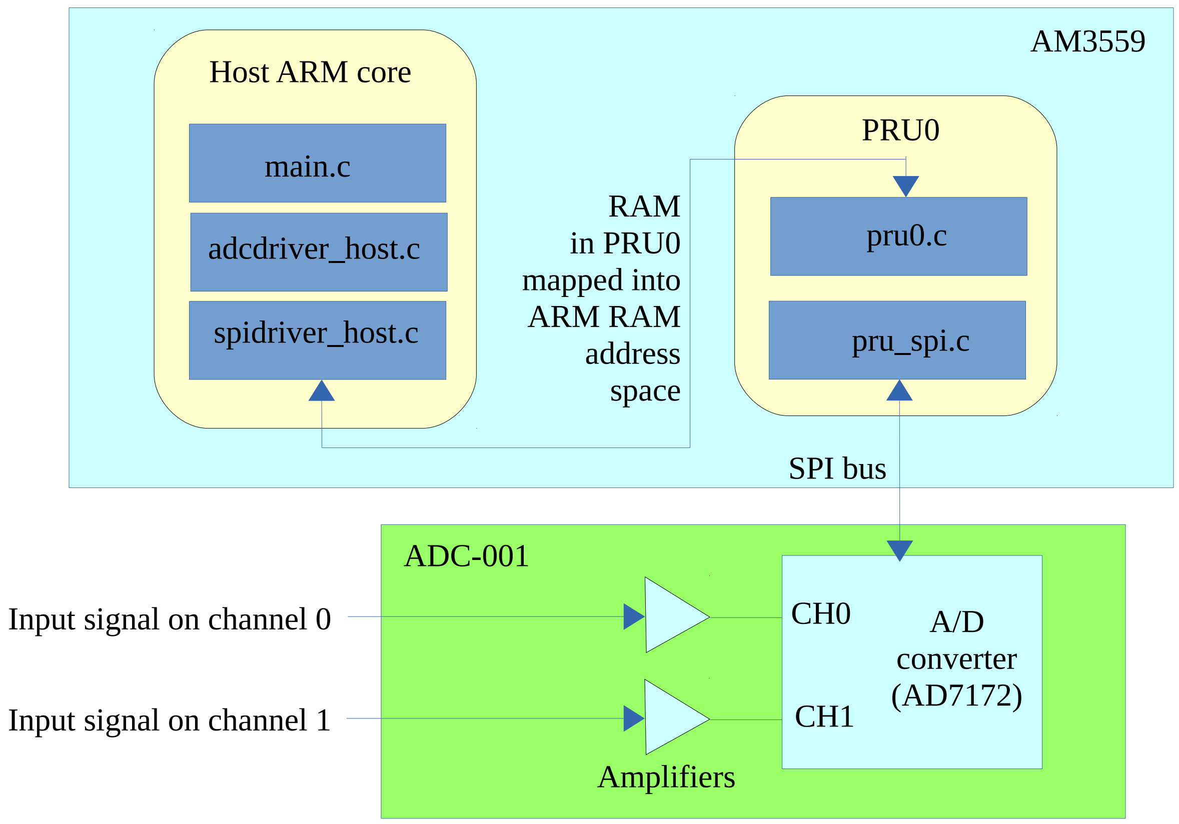
Last revision: 4.11.2017, SDB.

Introduction and overview

This document provides basic information about writing programs which use the A/D shield ADC-001.

Writing your program

A conceptual block diagram of an application using the A/D shield is shown below. The host ARM microprocessor (AM3359) is shown on top. The A/D shield, ADC-001 is shown beneath the ARM. The A/D itself is an Analog Devices AD7172 configured to support two input channels. Communication between the AM3359 and the A/D takes place via a SPI bus.



Different pieces of software functionality running on the ARM live in different files. The different files are shown as dark blue rectangles. The top level code lives in the file `main.c`. If you are a beginner or you have a simple program, your code should be placed into this file, in the function called `main()`. More advanced users may want to spread their program around in different .c files to conceptually organize their work. You may use the existing function `main()` as a template, and adapt it to your needs.

Underneath the main function is a set of drivers for the A/D in the file `adcdriver_host.c`. These drivers expose a simple API which you may call in order to get data from the A/D hardware. Details of the API are given below. Beneath the A/D drivers are other functions which create the bit-banging hardware signals used to communicate with the A/D and other peripherals on the A/D shield. Although you won't need to directly call these functions to take data, the code is available for you to inspect and learn from.

The A/D board is controlled using one of the PRUs which are part of the main processor chip (AM3359). The PRUs are microcontrollers which take care of the bit-banging hardware interactions required to get data from the A/D. Communication between the ARM host CPU and the PRU takes place by reading and writing to a block of shared RAM which lives on the PRU. On the ARM side, the functions in `spidriver_host.c` deal with communication via RAM, while on the PRU0 side, the functions in `pru0.c` handle the communication. At the lowest level, the SPI bit-banging code lives in `pru_spi.c`.

Using the A/D converter from software

When writing your code, your job is to configure the A/D, and then request measured data from it. A basic program flow might look like this:

1. Initialize the A/D hardware using the function `adc_config()`
2. Enter a loop.
3. Perform A/D read using either `adc_read_single()` (to perform single reads from the A/D without concern about exact timing) or `adc_read_multiple()` (to fill a buffer of readings taken at a precise time delay from each other).
4. Process the A/D value(s) read. This might involve performing some signal processing computation on the value(s).
5. Go back to step 2 to perform the next loop iteration.

Termination of the program can occur after a certain number of for loops have been executed, or through receiving a signal (e.g. control-C from the keyboard).

A/D configuration

The following functions are used to configure the A/D and change its settings.

- `void adc_config(void);`

Call this function at the beginning of your program. It first sets up the PRU driver software, then sets up the communication RAM between PRU0 and the host ARM processor, and then performs all the hardware configuration steps required to use the A/D shield.

- `void adc_set_samplerate(int rate);`

This function sets the sample rate of the A/D in continuous sample mode. Here, the variable "`rate`" is an integer which corresponds to the sample frequency in Hz. Only certain sample rate values are allowed. This is because the AD7172 only supports certain sample rates. Refer to the AD7172 data sheet for more details. The allowed sample rates are defined at the top of `adcdriver.h`, and carry descriptive names like `SAMP_RATE_31250`.

- `void adc_set_chan0(void);`

This places the A/D into a mode where it reads channel 0 for all following measurement requests, starting with the next one. When the board comes out of reset, this is the default state.

- `void adc_set_chan1(void);`

This places the A/D into a mode where it reads channel 1 for all following measurement requests, starting with the next one.

Requesting a single measurement

At any time you may request a single measurement from the A/D using the function `adc_read_single()`. The active channel set by the last call to `adc_set_chan*` is the channel read. (The default is channel 0 if no call to `adc_set_chan*` has been made.) The function declaration is

- `float adc_read_single(void);`

The calling pattern is

```
float vmeas;

vmeas =  adc_read_single();
```

When you call this function, it will return the most recent measurement the A/D has made as a floating point number equal to the voltage at the A/D input.

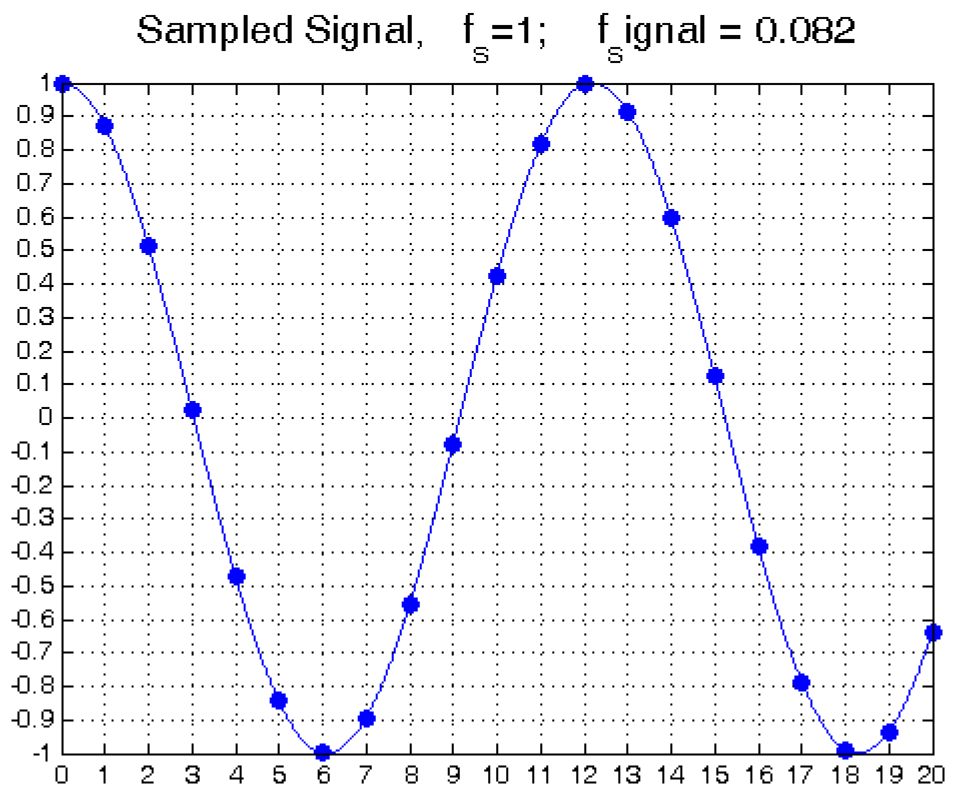
To measure many values, you can place this function call into a for loop. However, due to the indeterminacy of the exact time your function call is executed, the time delay between the measured samples is not guaranteed to be uniform. In some cases this might not matter. For example, if you are measuring a slow-moving signal, such as the temperature of a sample once per second, you may not care about jitter of a few milliseconds. However, if you are measuring a signal requiring uniformly-spaced time steps, such as an audio signal, this kind of jitter is unacceptable. Therefore, do not use this function for applications requiring uniform time steps between samples.

Requesting a vector of measurements

You may also request a buffer full of measured values taken one after the other using a uniform time step using the function `adc_read_multiple()`. A drawing showing such a sampled waveform is shown below. The underlying continuous signal is shown as the blue line, and the sampled points are shown as blue dots. Note that the sample points occur at uniformly-spaced intervals in time.

Certain algorithms, such as the FFT, assume uniformly-spaced samples. Due to the timing jitter which occurs when reading single values, if you want to perform computations requiring uniformly-spaced samples, you must read an entire vector of values prior to processing. The sampling interval of the vector of points is guaranteed to be uniform by the hardware. Once the values are gathered, you may then process the vector all at once before asking for the next vector.

The requirement for uniformly-spaced sampling is the reason that the ADC-001 API supports the two different functions `adc_read_single()` and `adc_read_multiple()`. It is important for you to understand the difference between what the two functions do.



The function declaration is

- `void adc_read_multiple(uint32_t read_cnt, float *volts);`

The calling pattern looks like this (assuming you want to sample 256 points):

```
float volts[256];

adc_read_multiple(256, volts);
```

The function args are the number of points to measure (in this case, 256) and a pointer to a buffer into which to place the measured samples. In the above example, the measured voltage samples are placed into the vector named "`volts`". Data acquisition starts after the function is called, and the function returns when all samples have been measured.

Other functions

- `uint32_t adc_get_id_reg(void);`

This function returns the value held in the AD7172 identification register. The ID is a hard-coded value built into the AD7172 itself. This function simply reads the ID value held in the AD7172. The value returned by this function should be `0xde`. You may use this function as a debug tool if you suspect you are having problems with the A/D. If you call this function and the return is not `0xde`, then it is likely you have a problem with either the A/D itself, or with the communication between the A/D and your software.

- `adc_reset()`

Requests a hardware reset of the AD7172.

- `adc_quit()`

Resets the SPI program running on PRU0, then releases the memory map which allows communication between the PRU and the host ARM processor. This function should be called prior to exiting your program because it releases resources back to the operating system. If you don't release resources after leaving your program, it is possible your program will fail if you run it again.

Building your program

A Makefile is included in the base directory. The Makefile will build all components of the Beaglebone A/D software system. To do a complete build, issue the command

```
make
```

To remove the old executables and other intermediate files, issue the command

```
make clean
```

A good way to make sure you have rebuilt everything is to issue the command

```
make clean && make
```

Running your program

When you run “make”, the main executable is build. The name of the main executable is “main”. To run your program, issue the command

```
./main
```

Note that you must run the program while logged in as “root” user. This is because the program relies on using some system resources to communicate with the PRU, and the system resources used may require “root” permissions for access.