Fast Analog to Digital Conversion for Arduino Boards

The Arduino *analogRead* function is very simple to use but is not terribly fast, especially if you have to read several inputs. A typical *analogRead* call on a Uno or Mega takes about 112 microseconds. If you need to read 3 or 4 inputs, that adds up to 336 or 448 us. If we are willing to give up a few bits of conversion accuracy we can get the ADC in the MCU to get a sample quite a bit faster, and if we are clever with our requirements so that the inputs we need fast responses for get sampled more often than those we care less about, then we can improve overall performance quite a bit.

The application that the code with this article was developed for required sampling the back-EMF from a brushless motor such as you'd find in a radio-controlled car or airplane. I wanted to be able to accurately determine the RPM of the motor with no additional sensors. In addition to measuring the RPM of the motor I also wanted some other analog inputs to measure battery voltage, motor current and the positions of some analog potentiometers. Only the motor phase sense readings (the back-EMF) needed to be fast. The other readings could occur less often, but it was important that they didn't interfere with the high speed readings.

The algorithm used here samples all the high speed inputs first. In my case there was one, but it could be two or more. Once the high-speed inputs have been sampled, one of the slow speed inputs is sampled. On the next cycle, the high speed inputs get sampled again and then the next slow input gets sampled.

So if we have one fast inputs, we can have as many slow inputs as we need without slowing down the fast reads very much. For one fast inputs, each cycle consists of two reads: one fast and one slow.

To illustrate the algorithm, let's say we have the input we want to read most often on A0, and we have three other inputs that we are happy to read slower on A1, A2, and A3. The read sequence would then be:

```
A0, A1, A0, A2, A0, A3, A0, A1, A0, A2, A0, A3, ...
```

To get the speed we want it's not possible to do the reads as part of the normal foreground code (essentially what you do when you call *analogRead*). Instead we set up the MCU hardware ADC to do the reads for us, and generate an interrupt when the analog conversion is finished. We service the interrupt as quick as we can, taking the sampled value and storing it in a global memory variable, after perhaps a little math. Then we tell the ADC hardware to start the next conversion cycle. That still doesn't get us all that much faster unless we also alter the ADC clock prescaler. By making the ADC clock faster, we gain quite a bit of speed (reduced conversion time) at the expense of some conversion accuracy.

Using this technique I got the fast analog read for one input down to about 40 microseconds, so that for one fast input and multiple slow inputs, I got a fast input cycle time of about 40 * 2 = 80 us. If you only have one input to sample then you can read it about every 40 us (about 25 kHz). That is fast enough to sample an audio input or, for me, a brushless motor back EMF signal.

If you don't understand computer interrupts you might like to read Nick Gammon's excellent work: https://www.gammon.com.au/interrupts.

It's worth taking a minute here to look at the ADC conversion timing. Per the ATmega328P specification section 23.4, a normal ADC conversion takes 13 ADC clock cycles. The Arduino boards have 16 MHz clocks, and this is divided down by the ADC prescaler. The register bits that control the prescaler are in Table 23-5 of the spec.

The example code sets only ADPS2 which gives us a division ratio of 16. So we divide the CPU clock down to 1 MHz which is a period of 1 us. 13 clocks for the ADC conversion gives us a theoretical conversion time of 13 us.

In practice we need to respond to the interrupt generated when conversion is complete which takes a little time and then grab the ADC sample and set it up for the next conversion. All of that takes a bit more time. The ex2_fast_a0 example shows this working in practice. The overall conversion time comes out to about 16 microseconds. That's 13 us for the ADC conversion and 3 us for the ISR code.

Setting the prescaler to 8 instead of 16 will half the 13 us conversion time to just 6.5 us. However, there is now significant overhead getting to and from the ISR compared to the ADC conversion time so you won't win a 2X improvement. I found this gave me a total cycle time of about 9.5 us. That's 6.5 us for the ADC conversion and 3 us for the ISR code. That is way faster than the 112 us required for *analogRead*!

As the ADC runs faster its conversion accuracy decreases. You might need to experiment to see if the reduced accuracy works for your needs. For my motor back-EMF measurements a prescaler of 16 turned out to work very well. I used a simple resistor divider to reduce the motor voltage to the 5 V input range of the ADC. I also added a small smoothing capacitor to filter out some noise form the motor ESC.

When we give up using the Arduino functions like *analogRead*, and start programming the MCU registers directly, we also give up some convenience. When I write alternatives like this faster ADC system, I do not try to make the code too general. I try to just make it do what I need in the simplest way possible. I also try quite hard to limit the amount of data storage required since this is very limited on an Arduino Uno. Most of the work I did this project for was actually done on an Arduino Mega board as my program ended up too big for a Uno and I wanted more I/O pins.

The example code

There are three example applications with this article. The first one: ex1_analog_read is just to show how long a single call to *analogRead* takes. Running it will print out something like this on the Serial Monitor:

```
10000 analogRead calls took: 1120124 us. 112 us per call.
10000 analogRead calls took: 1120056 us. 112 us per call.
10000 analogRead calls took: 1120056 us. 112 us per call.
```

EXAMPLE 2

The second example in ex2_fast_a0 is a complete application that runs the ADC with a faster clock. You can chose a prescaler of 16 or 8 in the code. It only reads one input and serves as an example of the best possible case, from a timing point of view. The code toggles a port output pin when it enters the ISR and again when it leaves so that you can see on a scope the overall conversion period (leading edge to leading edge) as well as the amount of time spent servicing the interrupt. I use this technique a lot to measure exactly what my code is doing. You might note that the code does not use *digitalWrite* for the scope output. It's ok to call *digitalWrite* in an ISR, but it's slower than doing direct port writes and we are concerned with wasting the least amount of time possible inside the ISR. Once my code is working I'd remove the code for the port writes and save a few more CPU cycles in the ISR.

For this example the ADC result is simply put in a global variable, but in practice you would likely be doing more processing. For my motor application I ran the ADC outputs through some digital filters and then more code to compute the motor RPM.

The foreground code inside the *loop* function in the example reads the conversion result twice every second and prints it out. There is code to lock out the ISR by disabling interrupts while

the 16-bit value is read. Failure to do this might allow the ISR to update half of the 16-bit value while we are trying to read it.

If you run the example you'll see it print lines of values like this:

```
A0: 611
A0: 724
A0: 459
A0: 607
A0: 783
A0: 783
```

The implementation modifies the AVR MCU registers directly. It will work on Arduino Uno or Mega family boards. If you want to modify the code, you will need to be familiar with the ATmega registers and their use.

EXAMPLE 3

The third example is a more general solution with the fast ADC support code broken out into a C++ class in a .h file and a .ino file so that you can simply add the code to your own application. In this case you derive a class from the FastAdc class and overload the *onFastUpdate* and/or *onSlowUpdate* virtual functions to add your own code. You provide a list of the fast ports to sample, and a list of the slow ports. This example samples one port in the fast list and three in the slow list. You can modify it easily to read more fast ports, or whatever combination you like.

The example code shows how to add some additional code to process the data as it is read from the ADC. In this case it simply finds the maximum value. The foreground code resets the maximum value occasionally. This is just to illustrate how to use the FastAdc class, it's not really a very useful thing to be doing with the data. I also chose to print out the result in mV rather than the raw ADC conversion values.

```
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
Peak reset
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
A0: 3826 mV, Peak: 3826 mV, A1: 3797 mV, A2: 3797 mV, A3: 3797 mV
```

That should be enough to either get you on your way to faster ADC conversions. Just beware that any code that directly programs the MCU registers locks you in to that family of processors (the AVR family in this case), and possibly to a specific processor.