AN-03 Over-sampling an ADC to increase resolution.

April 2003, cws/alm

All analog systems, no matter how clean or stable, contain a noise component. In this application note we make use of this error to actually improve resolution and accuracy.

No matter what resolution A/D converter is used noise will occasionally push the conversion across at least one bit threshold on occasion. By taking statistically significant numbers of conversions, then averaging, we can effectively mulitply the inherent resolution of most analog to digital converters.

Note that more samples are required than a simple addition of their results, ie: a 16-bit result requires more than 64 (2^6) conversions of a 10-bit converter.

This application is only done in forth, not in BASIC at this time. Both example programs can be found (in their current executable form) in the directories ~/amrforth/v6/017/example for Linux, and \amrforth\v6\017\example for Windows.

First let's talk about the no-interrupts version, program #1. On line 4 we create a 'value' of 14 named 'bitsofadc'. This is similar to a constant but can be changed using the word 'to'. For example, '16 to bitsofadc'. We use 'value' here because it only works on the host, won't waste any memory on the target. The value of 14 in this case means we will conditionally compile the 14 bit version of 'adc@'.

On line 6 we define a code word called init-adc. This word initializes the adc for single ended input, sets the gain to 1, turns on continuous tracking, turns bias on, turns temperature sensor off, and enables the internal voltage reference. See the Cygnal data sheet for the f017 for more details, available from the website, www.cygnal.com.

Line 14 checks to see if you asked for 10 bit resolution above, if so the next lines are compiled up to the [then]. This is the natural resolution of the f017. The code word 'adc10@' on line 16 is the simplest way to read the adc. '$90 # ADC0CN' starts the conversion by clearing the interrupt bit, setting the ADCBUSY bit, and enabling the ADC. Then we loop, waiting for bit 5 of ADC0CN to become set. This is the ADC completion interrupt bit. On lines 19 and 20 we read the low and the high bytes of the result and push them onto the data stack.

In order to get 12 bits of resolution we need to add 16 samples then divide by four. See the Cygnal appnote AN018.pdf, "Improving ADC resolution by Oversampling and Averaging", available at http://www.cygnal.com, for more details. The word 'adc@' on line 24 implements a 12 bit version. A 2 is placed on the stack to seed the accumulator. This 2 is equivalent to 0.5 after dividing by 4 at the end of the word. This is a tricky way to round correctly. The 15 for ... next loops 16 times by counting down to 0. Inside the loop we read the 10 bit adc and add it to the total on top of the stack. After the loop terminates we divide by 4 to average the number down to 12 bits resolution.

On line 30 is a similar word to get 14 bits resolution. Since 256 times 1023 is more than 16 bits, we need to use a double precision accumulator. D+ is not part of the default kernel. It can be found in the file 'preamble.fs' in /amrforth/v6/lib/.

Notice that the high level version of the 14 bit adc@ is commented out. Line 32 starts the code version of the same word. In the code version we use registers R4, R5, and R6 to accumulate the sum of the readings. Register R7 is used as a loop counter. We load R4 with 8 in line 34 to help with rounding later. Lines 35-40 loop 256 times, waiting for a conversion to complete and adding the result into R4,R5,R6. Then lines 41-46 shift those registers right by 4 places, dividing by 16. Lines 47-48 push the 16 bit result onto the data stack.

The 16 bit version is very similar to the 14 bit version. Only the loop counters and the rounding seed change.

At line 79 the \S comments out the rest of the file. The words that follow were used to time each of the adc words.

Program #2 uses the ADC completion interrupt to accumulate samples in the background and signal the foreground when it is time to average.

On line 5 we define another 'value', 'START-CONVERSION', which represents the bit pattern that clears the interrupt bit and starts a new conversion. It is a value because it is only needed at assembly time, not at runtime on the target. It gets used at least twice in the program file, so it made sense to name it. Lines 9-16 create byte variables that are only used in assembly language. Once again, this is to save some memory on the target. Each word is an assembler macro that places the address of the appropriate variable on the stack and sets direct addressing mode for the assembler. This is to avoid typing the word direct over and over again.

The word init-adc works similarly to the one in program #1, but it also clears some accumulator variables, starts the first conversion, and enables the interrupt.

The interrupt routine is lines 347-57. Since we do arithmetic in the interrupt, we need to save and restore ACC and PSW. On lines 40-42 we read the ADC result and add it to the accumulator variable, which is 3 bytes. Then we clear the interrupt and start the next conversion on line 44. Line 46 decrements the counter. We're taking 256 samples here so an 8 bit counter is perfect. If it is non-zero then we restore PSW and ACC and leave. If it has counted down to zero, then we move the accumulator bytes into the total bytes, a sort of double buffering. A 1 is moved into the adc-flag to signal the foreground that the samples are ready. PSW and ACC are restored and we leave.

On line 60 is the word adc@. It loops until 256 samples have been accumulated, which in most cases has already been done in the interrupt. Interrupts are disabled while reading the total, so that it can't be changed in the middle of the read. The 3 byte total is read into registers R4, R5, and R6. R7 is used as a loop counter to shift those bytes right by 4 places, dividing by 16. As usual, the 16 bit result is pushed onto the data stack.

The words labeled 'debugging' were used while testing the words above.

```
\ Program #1
\ No interrupts, choice of 10, 12, 14, or 16 bit resolution.
\ See ~/amrforth/v6/017/example/adc017.fs
\ or   \amrforth\v6\017\example\adc017.fs
\ for the current executable version of this program.
\ adc017.fs
\ Using the 10 bit A/D on the f017.

14 value bitsofadc

code init-adc  (  - )
        0 # AMX0CF mov    \ All inputs are single ended.
        0 # AMX0SL mov    \ Select AIN0, single ended.
        $80 # ADC0CF mov  \ $80 SAR=16 gain=1
        $80 # ADC0CN mov  \ $80 Always tracking
        $03 # REF0CN mov  \ Internal Ref., bias on, temp sensor off.
        next c;

10 bitsofadc = [if]
code adc@  (  - n)  [then]
code adc10@  (  - n)
        $90 # ADC0CN mov  \ Start the conversion.
        begin  5 .ADC0CN set? until  \ Wait for conversion complete.
        ADC0L A mov  Apush
        ADC0H A mov  Apush
        next c;

12 bitsofadc = [if]
: adc@  (  - n) 2 15 for  adc10@ +  next  2/ 2/ ;
[then]


14 bitsofadc = [if]
\ : slow-adc14@  (  - n)
\       8 0 255 for  adc@ 0 d+  next  16 um/mod nip ;

code adc@  (  - n)
        A clr  A R5 mov  A R6 mov  A R7 mov
        8 # A mov  A R4 mov  \ Start with 0.5, for rounding later.
        begin   $90 # ADC0CN mov  \ Start the conversion.
                begin  5 .ADC0CN set? until  \ Wait for conversion.
                ADC0L A mov  R4 A add  A R4 mov
                ADC0H A mov  R5 A addc  A R5 mov
                A clr  R6 A addc  A R6 mov
        R7 -zero until
        4 # R7 mov
        begin   C clr
                R6 A mov  A rrc  A R6 mov
                R5 A mov  A rrc  A R5 mov
                R4 A mov  A rrc  A R4 mov
        R7 -zero until
        R4 A mov  Apush
        R5 A mov  Apush
        next c;
[then]

16 bitsofadc = [if]
\ : slow-adc16@  (  - n)
\       32 0 4095 for  adc@ 0 d+  next  64 um/mod nip ;

code adc@  (  - n)
        A clr  A R5 mov  A R6 mov
        32 # A mov  A R4 mov  \ Start with 0.5, for rounding later.
        16 # B mov
        begin   \ A total of 4096 samples accumulated.
                A clr  A R7 mov
                begin   4 .ADC0CN setb  begin  4 .ADC0CN clr? until
                        ADC0L A mov  R4 A add  A R4 mov
                        ADC0H A mov  R5 A addc  A R5 mov
                        A clr  R6 A addc  A R6 mov
                R7 -zero until
        B -zero until
        6 # R7 mov   \ Divide by 64, 2^6.
        begin   C clr
                R6 A mov  A rrc  A R6 mov
                R5 A mov  A rrc  A R5 mov
                R4 A mov  A rrc  A R4 mov
        R7 -zero until
        R4 A mov  Apush
        R5 A mov  Apush
        next c;
[then]

\S Timing tests.

\ 1,000,000 iterations.
: test10  (  - ) 100 for 10000 for  adc@ drop  next next ;

\ 10,000 iterations.
: test12  (  - ) 10000 for  adc12@ drop  next ;

\ 1000 iterations, high level.
: test14slow  (  - ) 1000 for  slow-adc14@ drop  next ;

\ 1000 iterations, coded.
: test14  (  - ) 1000 for  adc14@ drop  next ;

\ 100 iterations, high level.
: test16slow  (  - ) 100 for  slow-adc16@ drop  next ;

\ 100 iterations, coded.
: test16  (  - ) 100 for  adc16@ drop  next ;
```

```
\ Program #2
\ Using an interrupt to accumulate samples.  14 bit resolution only.
\ See ~/amrforth/v6/017/example/adc017int.fs
\ or   \amrforth\v6\017\example\adc017int.fs
\ for the current executable version of this program.
```

```
\ adc017int.fs
\ Using the 10 bit A/D on the f017 in an interrupt routine,
\ over-sampled to get 14 bits of date.

$90 constant START-CONVERSION

\ Variables for use only in assembler definitions.
\ No need for forth executables on the target.
a: adc-flag    [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: adc-counter [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: total1      [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: total2      [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: total3      [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: acc1        [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: acc2        [[ cpuHERE ]] literal direct ;a  1 cpuALLOT
a: acc3        [[ cpuHERE ]] literal direct ;a  1 cpuALLOT

code init-adc  (  - )
        $80 invert # IE anl  \ Disable interrupts globally.
        A clr
        A adc-counter mov  \ 256 samples.
        $08 # acc1 mov  \ Start accumulator with 0.5, for rounding.
        A acc2 mov
        A acc3 mov
        A adc-flag mov  \ Total not ready.
        $00 # AMX0CF mov    \ All inputs are single ended.
```

```
         $00 # AMX0SL mov    \ Select AIN0, single ended.
         $80 # ADC0CF mov  \ $80 SAR=16 gain=1
         $80 # ADC0CN mov  \ $80 Always tracking
         $03 # REF0CN mov  \ Internal Ref., bias on, temp sensor off.
\ enable the interrupt and start the first conversion.
         $02 # EIE2 orl
         $80 # IE orl
         START-CONVERSION # ADC0CN mov
         next c;

label adcint
         ACC push  PSW push
\ Accumulate this sample.
         ADC0L A mov   acc1 A add   A acc1 mov
         ADC0H A mov   acc2 A addc  A acc2 mov
         A clr  acc3 A addc  A acc3 mov
\ Start the next conversion, clear interrupt flag.
         START-CONVERSION # ADC0CN mov
\ Decrement the counter.
         adc-counter dec
\ Leave unless finished accumulating.
         adc-counter A mov  0<> if
                  PSW pop  ACC pop  reti
         then
\ Save total and restart accumulation.
         A clr
         acc1 total1 mov  8 # acc1 mov
         acc2 total2 mov  A acc2 mov
         acc3 total3 mov  A acc3 mov
         1 # adc-flag mov
         PSW pop  ACC pop  reti c;
adcint $7b int!

code adc@  (  - n)
         begin  adc-flag A mov  0<> until  0 # adc-flag mov
         IE push  \ Save the interrupts flag.
         $7f # IE anl  \ Disable interrupts globally.
         total1 A mov  A R4 mov
         total2 A mov  A R5 mov
         total3 A mov  A R6 mov
         IE pop  \ Restore the interrupts flag.
         4 # R7 mov  \ Divide by 16.
         begin    C clr
                  R6 A mov  A rrc  A R6 mov
                  R5 A mov  A rrc  A R5 mov
                  R4 A mov  A rrc  A R4 mov
         R7 -zero until
         R4 A mov  Apush
         R5 A mov  Apush
         next c;

\S ----- debugging ----- /

code IE@  (  - c)  IE A mov  Apush  0push  next c;
code EIE2@  (  - c)  EIE2 A mov  Apush  0push  next c;
code adc-flag@  (  - c) adc-flag A mov  Apush  0push  next c;
code adc-counter@  (  - c)
         adc-counter A mov  Apush  0push  next c;
code total@  (  - ud)
         $7f # IE anl
         acc1 A mov  Apush
         acc2 A mov  Apush
         acc3 A mov  Apush
         A clr  Apush
         $80 # IE orl
```

```
            next c;

```