ember

Application Note 5071
3 November 2011
120-5071-000A

Using the EM35x ADC

This document describes how to use the ADC in the EM35x, gives an overview of the internal design, discusses design considerations, and provides code examples of fundamental usage. Because the example code uses direct register accesses, there is no fundamental requirement on the rest of the Hardware Abstraction Layer and these examples can be tailored to couple tightly with individual applications.

New in This Revision

Initial release.

Contents

Background 2
EM35x ADC Overview 2
Start-up and Sample Times 3
Reference Sources 3
General Usage 5
Configuring the ADC 5
Taking a Measurement....................6
Using Offset and Gain Correction....................6
Measuring Absolute Voltage 7
Calibrating for Absolute Voltage....................8
  Code Sample....................8
Ratiometric Measurements 9
  Code Sample....................9
Differential Measurements.................... 10
Using the DMA.................... 11
  Code Sample.................... 11
Using the ISR 11
  Code Sample.................... 12
General Suggestions for Improving Accuracy 12
Glossary.................... 13

Ember Corporation
25 Thomson Place
Boston, MA 02210
+1 (617) 951-0200
www.ember.com

Background

The EM35x Analog to Digital Converter (ADC) is a first-order sigma-delta converter. While a full treatment of sigma-delta architecture is beyond the scope of this document, it may be helpful to understand the general principles of operation before programming the EM35x ADC.

The ADC works by generating a series of pulses where the pulse density is proportional to the analog voltage at the input. This is effectively a 1-bit quantisation of the input sample, running much faster than the minimum Nyquist sample frequency. This high oversampling rate has the benefit of simplifying the output filter by ensuring the input signal is aliased at frequencies much higher than the bandwidth of interest.

However, taken in isolation, each 1-bit sample from the sigma-delta output has no real meaning. Only by reducing the sample rate and producing multi-bit parallel outputs can each sample be interpreted as a digital representation of the analog input. This process, known as decimation, also explains how setting a longer sample period improves the resolution of the ADC result.

EM35x ADC Overview

The ADC can sample up to 6 signals in either single-ended or differential modes. While the ADC module supports both types of conversion, the input stage always operates in differential mode. Single-ended conversions are performed by connecting one of the differential inputs to VREF/2, while fully differential operation uses two external inputs. Both inputs can be connected to internal sources for calibration purposes.

In addition to sampling external inputs through GPIOs, the ADC can internally sample the 1.8 V supply (VDD_PADSA/2), VREF, VREF/2, and GND.

Note: In the typical application circuit, VDD_PADSA is externally driven by the VREG_OUT output. Therefore, the names VDD_PADSA and VREG are often used interchangeably.

The GPIOs, which can be configured as ADC inputs, are mapped to pins shown in Table 1.

Table 1. ADC External Input Pin

Analog Signal	GPIO	Pin Number
ADC0	PB5	43
ADC1	PB6	42
ADC2	PB7	41
ADC3	PC1	38
ADC4	PA4	26
ADC5	PA5	27

A full description of the ADC registers and GPIO configuration can be found in Ember document 120-035X-000, *EM35x Datasheet*.

Start-up and Sample Times

A 21 µs start-up time is imposed by the analog hardware when the ADC is first enabled, but this does not apply after a configuration change. After the analog hardware startup, when the ADC state is changed to enabled and/or any change is made to the ADC_CFG register, the time until the next result becomes available is:

$$SampleTime\ \ (\mu s) = 2\left(\frac{\#\ Sample\ Clocks}{Clock\ \ Frequency\ (MHz)}\right)$$

After the initial conversion on a given configuration the ADC enters an enhanced stream mode during which it outputs results at twice the one-shot sample rate.

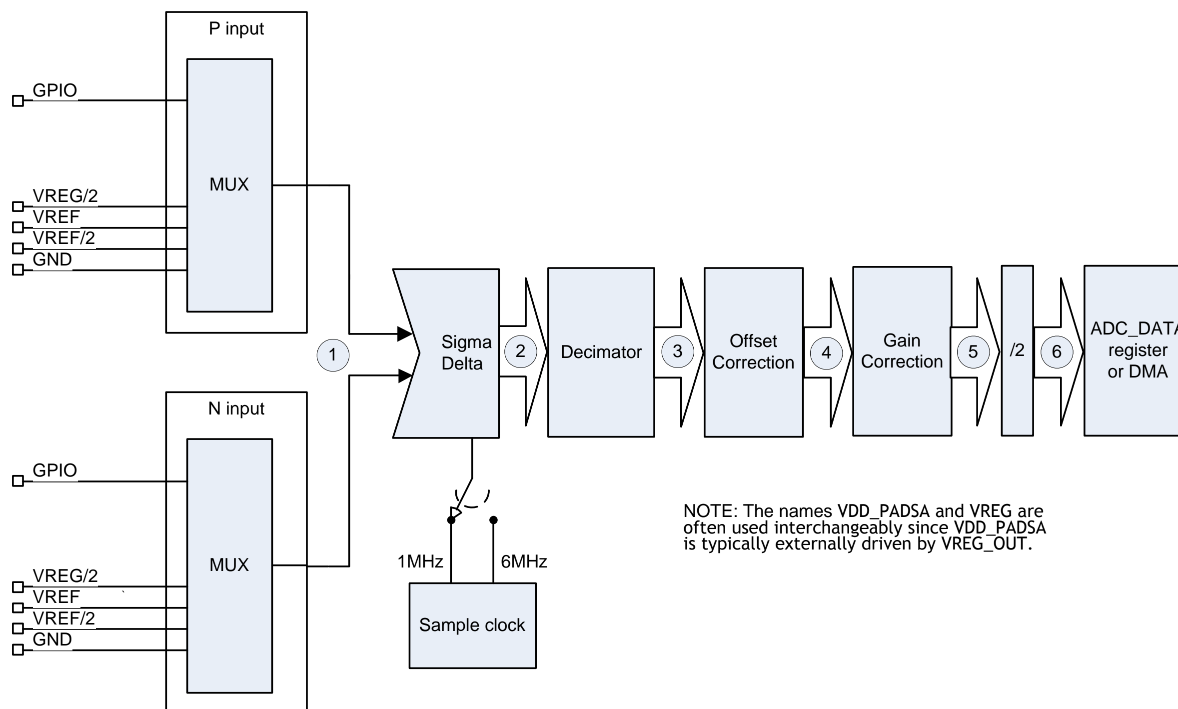
Reference Sources

Figure 1 shows a simplified block diagram of the ADC's internal structure. The two internal voltage references are available to the ADC. Additionally, the VREF internal voltage reference is also available to the ADC divided by 2 and VREF/2 is used as the N input for all single-ended measurements

- VREF
    - 1200 mV nominal
    - VREF/2 is used as the N input for all single-ended measurements
    - Can be calibrated if absolute voltage accuracy is needed
    - Can be used as a stable source for gain and offset corrections

- VREG/2
    - 1800 mV factory trimmed to ± 20 mV, then divided by 2
    - Actual value (1800 mV) is measured and stored in Manufacturing Tokens `(TOKEN_MFG_1V8_REG_VOLTAGE)`

Figure 1. EM35x ADC Simplified Block Diagram



The inputs and outputs in the blue numbered circles are as follows:

1 analog Input

- ±1200 mV Differential
- Single Ended is ±600 mV against 600 mV (VREF/2)

2 Output of Sigma-Delta

- 1 bit @ Sample Clock Rate

3 Output of Decimator

- Maximum resolution is 16 bits every (Sample Clock Period * 4096)
- Single Ended 15 bits (can exceed ±600 mV without clipping)

4 Offset Adjustment

- Calculated from minimum input value
- Single ended scale using GND against VREF/2 measurement

5 Gain Correction

- Calculated so maximum input gives 14 bits full scale (When using 4096)
- Single ended this is +600 mV => +16384 with offset correction

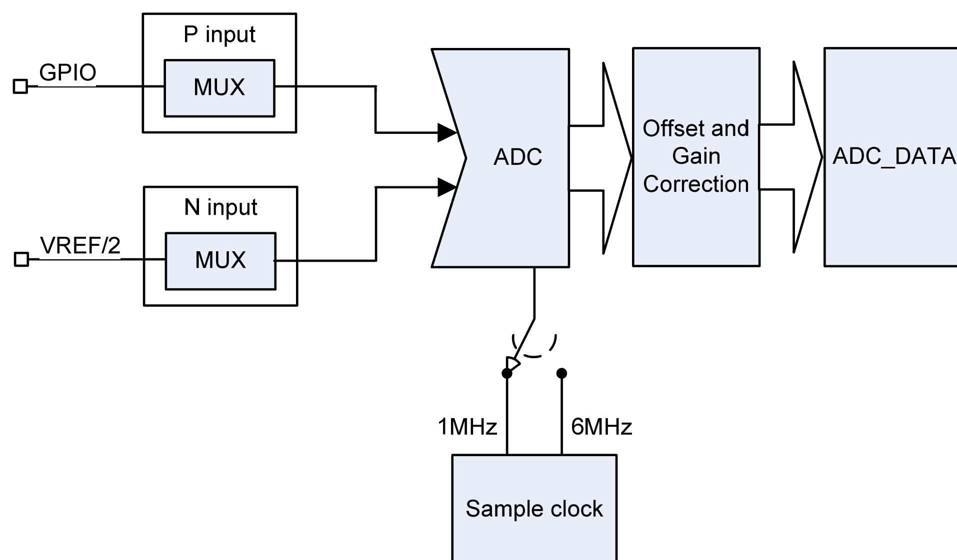
6 Divide by 2

- Ensure measurement stays in range
- Single ended now 14 bits (±8192 without offset correction)

General Usage

In most cases the user will simply want to measure a single input value against the internal voltage reference (VREF/2). This simple scenario is used in the basic descriptions in this document and is illustrated in Figure 2.

Figure 2. General Usage



To clarify and simplify the code provided in this document, the following set of defines are used.

```
#define ADC_INPUT_PB5    0x0   /* ADC0 */
#define ADC_INPUT_PB6    0x1   /* ADC1 */
#define ADC_INPUT_PB7    0x2   /* ADC2 */
#define ADC_INPUT_PC1    0x3   /* ADC3 */
#define ADC_INPUT_PA4    0x4   /* ADC4 */
#define ADC_INPUT_PA5    0x5   /* ADC5 */
#define ADC_INPUT_GND    0x8   /* GND, 0V */
#define ADC_INPUT_VREF2  0x9   /* VREF/2, 0.6V */
#define ADC_INPUT_VREF   0xA   /* VREF, 1.2V */
#define ADC_INPUT_VREG2  0xB   /* VDD_PADSA/2, 0.9 */
```

Configuring the ADC

Before any samples are taken, the ADC registers must be configured with the desired input and clock settings. Also, the required GPIO pins should be properly configured in analog mode. Normally GPIO configuration would be done in the board header file but the code is repeated in this configuration function for convenience. This configuration function is the most basic example intended for taking simple single-ended measurements where the availability of a result is indicated by an interrupt. Because this configuration function is very basic, it could be the starting point for writing more complete software.

```
void adcConfigure(void)
{
  // configure PB5 as analog
  SET_REG_FIELD(GPIO_PBCFGH, PB5_CFG, GPIOCFG_ANALOG);
```

```
  // configure PB6 as analog (this is used for differential measurements)
  SET_REG_FIELD(GPIO_PBCFGH, PB6_CFG, GPIOCFG_ANALOG);

  // configure adc registers default to slow clock for higher input
  // impedance
  // default to best resolution (0x7 = 4096 clock cycles, 14 bit)
  // measure PB5 vs vref/2
  ADC_CFG = ADC_1MHZCLK
            | (0x7 << ADC_PERIOD_BIT)
            | (ADC_INPUT_PB5 << ADC_MUXP_BIT)
            | (ADC_INPUT_VREF2 << ADC_MUXN_BIT);

  // configure interrupts for polling
  INT_CFGCLR = INT_ADC;
  INT_ADCCFG = INT_ADCULDFULL | INT_ADCDATA;

  // start the ADC
  ADC_CFG |= ADC_ENABLE;
}
```

Taking a Measurement

The easiest way to read a single value from the ADC is simply to configure the registers as above, and wait for the interrupt flag. If the ADC has been enabled to take a measurement in a different section of code, be careful to clear the interrupt flags before the desired measurement has occurred/finished.

```
int16s adcMeasure(void)
{
  // clear interrupt flags
  INT_ADCFLAG = 0xFFFF;

  while (!(INT_ADCFLAG & INT_ADCDATA)) { // wait for INT_ADCDATA
  }

  return (int16s)ADC_DATA;
}
```

In the above sample, no gain or offset correction has been applied and the final result will be a value between approximately -8192 (GND) and +8192 (VREF).

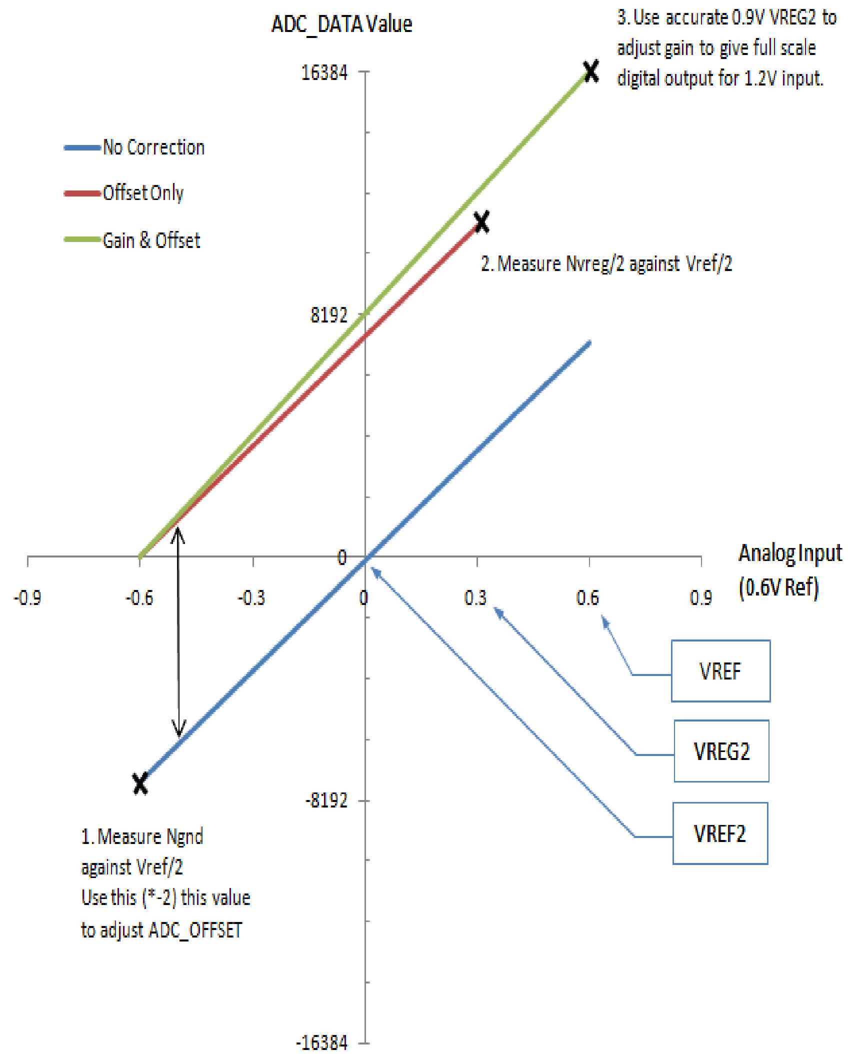
Using Offset and Gain Correction

Although the code above is satisfactory for some purposes, in other circumstances it may be beneficial to use the internal gain and offset corrections in order to make the raw ADC data more readily useful. Using this feature can also ensure that the maximum resolution is available from the ADC – i.e. obtaining maximum ADC_DATA range for 0-1200 mV input.

Measuring Absolute Voltage

Figure 3 shows how the corrections might be used to produce a near-zero ADC output for readings at ground potential, while giving full scale 14-bit output for readings close to VREF.

Figure 3. Gain/Offset Plot



Calibrating for Absolute Voltage

Code Sample

```
void calibrateAbsolute(void)
{
  int16s ngnd;
  int16s vdd;
  int16s nvreg;

  // ensure that the corrections are reset
  ADC_OFFSET = ADC_OFFSET_RESET;
  ADC_GAIN = ADC_GAIN_RESET;

  // read factory trimmed vdd value from token
  halCommonGetToken(&vdd, TOKEN_MFG_1V8_REG_VOLTAGE);

  // take ground reading (ngnd)
  SET_REG_FIELD(ADC_CFG, ADC_MUXP, ADC_INPUT_GND);
  SET_REG_FIELD(ADC_CFG, ADC_MUXN, ADC_INPUT_VREF2);
  ngnd = adcMeasure();

  // multiply ngnd by -2 and write to offset register
  ADC_OFFSET = -2 * ngnd;

  // read 1.8V regulator/2 (nvreg)
  SET_REG_FIELD(ADC_CFG, ADC_MUXP, ADC_INPUT_VREG2);
  nvreg = adcMeasure();

  // adjust gain so 1200mV produces 16384 at adc raw output
  // (32768 corresponds to a gain value of 1.0)
  ADC_GAIN = (int16u)(32768L * (16384L * vdd / 24000L) / nvreg);
}
```

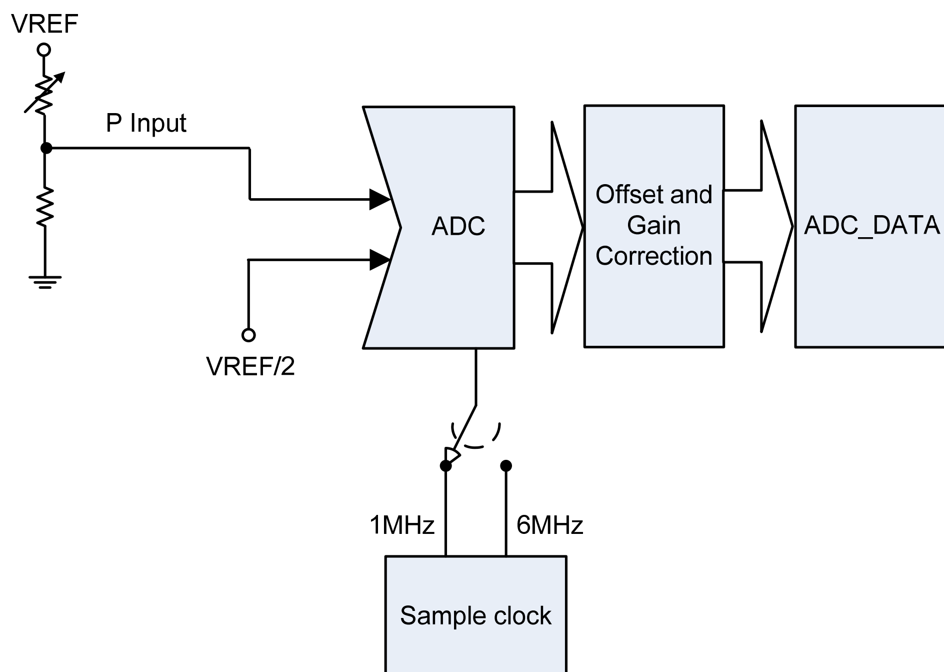
Because calibration requires changing the inputs to the ADC mux, you will need to call your ADC configuration code again, or preferably manually change the selected inputs before taking readings. The raw readings can be converted easily to an absolute voltage:

```
int16s convertToMillivolts(int16s adcdata)
{
  // if calibrated, just scale by nominal vref (1200mV)
  return ((adcdata * 1200L) / 16384L);
}
```

Ratiometric Measurements

A typical configuration is shown in Figure 4.

Figure 4. Ratiometric Measurement Configuration



In this case an absolute measurement is not needed as the sensor is excited by the same voltage as the reference used by the ADC. Therefore there is no need to adjust the gain based on the factory trimmed value of VREF. All that is necessary is to adjust the offset as before, then calculate the gain to produce full scale positive output (16384) when the input is at the reference voltage.

Code Sample

```
void calibrateRatio(void)
{
  int16s ngnd;
  int16s nvref;

  // ensure that the corrections are reset
  ADC_OFFSET = ADC_OFFSET_RESET;
  ADC GAIN = ADC GAIN RESET;

  // read Ngnd
  SET REG FIELD(ADC CFG, ADC MUXP, ADC INPUT GND);
  SET REG FIELD(ADC CFG, ADC MUXN, ADC INPUT VREF2);
  ngnd = adcMeasure();

  // multiply ground reading by -2 and write to offset register
  ADC_OFFSET = -2 * ngnd;

  // read nvref
  SET_REG_FIELD(ADC_CFG, ADC_MUXP, ADC_INPUT_VREF);
  nvref = adcMeasure();

  // adjust gain so Vref produces exactly 16384 at the output
```

```
  ADC_GAIN = (int16u)((32768L * 16384L) / nvref);
}
```

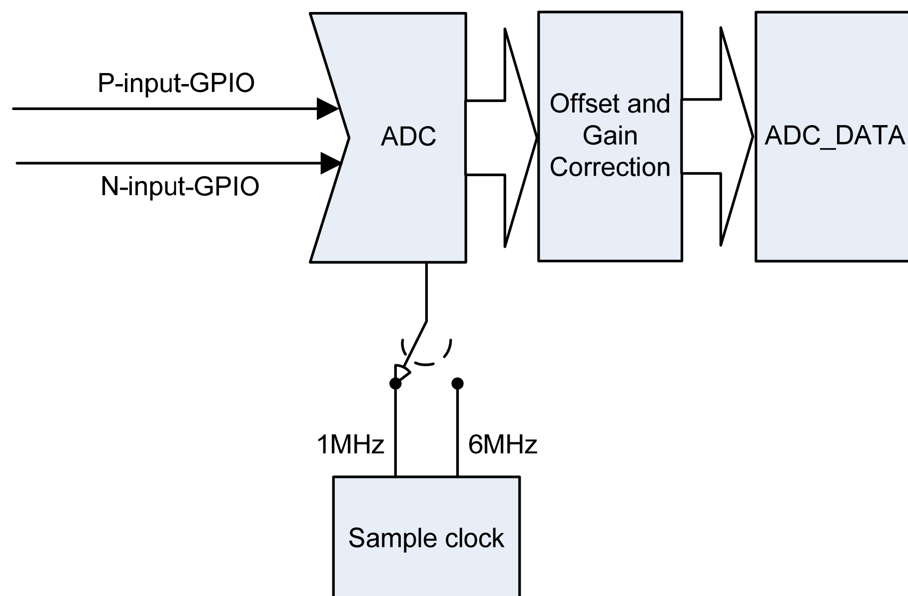
Just as in the absolute calibration case, ratio calibration requires changing the inputs to the ADC mux, so you will need to call your ADC configuration code again or preferably manually change the selected inputs before taking readings. The raw readings can now be converted to a fraction of VREF. In this example, the fraction is expressed as a percentage (100 == VREF).

```
int16s convertToRatio(int16s adcdata)
{
  // ratio measurement - scale by nvref, which is calibrated to be 16384
  return ((adcdata * 100L) / 16384);
}
```

Differential Measurements

A typical differential measurement configuration is shown in Figure 5.

Figure 5. Differential Measurement Configuration



In general, differential measurements rely on monitoring changes between two dynamic inputs and therefore produce relative and not absolute readings. The sample below configures the ADC to measure changes between signals on PB5 and PB6 (any two distinct GPIO pins from Table 1 may be used likewise).

```
int16s adcDiffMeasure(void)
{
  // measure PB6 vs PB5
  SET_REG_FIELD(ADC_CFG, ADC_MUXN, ADC_INPUT_PB5);
  SET_REG_FIELD(ADC_CFG, ADC_MUXP, ADC_INPUT_PB6);

  // clear the ADC interrupts
```

```
  INT_ADCFLAG = 0xFFFF;

  while (!(INT_ADCFLAG & INT_ADCDATA)) { // wait for INT_ADCDATA
  }

  return (int16s)ADC_DATA;
}
```

Using the DMA

The EM35x incorporates a DMA channel to facilitate transferring ADC results from the main ADC_DATA register to RAM. It can be operated in two modes – linear and autowrap:

- In linear mode the DMA continues to write to the buffer until ADC_DMASIZE is reached. The DMA then stops the transfer automatically, setting the INT_ADCOVF flag if a further ADC conversion completes before the DMA is reset
- In auto-wrap mode the DMA cycles through the buffer continually, overwriting older ADC samples as necessary.

Code Sample

```
#define DMA_BUFFER_SIZE     8

// buffer to fill with DMA data
int16s dmaBuffer[DMA_BUFFER_SIZE];

void dmaMeasurementBlocking(void)
{
  // reset dma & define buffer
  ADC_DMACFG = ADC_DMARST;
  ADC_DMABEG = (int32u)dmaBuffer;
  ADC_DMASIZE = DMA_BUFFER_SIZE;

  // load DMA
  ADC_DMACFG = ADC_DMALOAD;

  // clear the ADC interrupts
  INT_ADCFLAG = 0xFFFF;

  while (!(INT_ADCFLAG & INT_ADCULDFULL)) { // wait for INT_ADCULDFULL
  }
}
```

Using the ISR

Although the code above demonstrates configuring the DMA, in most cases the DMA will be used when repeated ADC measurements are needed, and the blocking nature of the code may prevent other important processes from running on the CPU. It is usually desirable to implement any DMA buffer accesses from within the ADC ISR.

Note: Projects built using the Ember AppBuilder tool already have a pre-existing halAdcIsr routine in the HAL that must be deprecated if a custom function is to be implemented.

Code Sample

```
void dmaMeasurementNonBlocking(void)
{
  // reset dma & define buffer
  ADC_DMACFG = ADC_DMARST;
  ADC_DMABEG = (int32u)dmaBuffer;
  ADC_DMASIZE = DMA_BUFFER_SIZE;

  // interrupt on DMA buffer full only
  INT_ADCCFG = INT_ADCULDFULL;

  // clear the ADC interrupts
  INT_ADCFLAG = 0xFFFF;

  // load DMA
  ADC_DMACFG = ADC_DMALOAD;

  // enable NVIC ADC interrupt
  INT_CFGSET |= INT_ADC;
}

void halAdcIsr(void)
{
  // DMA has finished loading the RAM buffer.
  // Process ADC results here, or set a flag for checking in a
  // delayed service routine.  Ember recommends that you do no
  // processing in an ISR that takes longer than 400µs.

  // clear interrupt
  INT_ADCFLAG = 0xFFFF;
}
```

General Suggestions for Improving Accuracy

It is important to keep the source impedance as low as possible – at least low enough that the input impedance of the ADC is a few orders of magnitude larger. As a rough guide, a 10 kΩ source produces an error in the output of around 1%. If there are concerns regarding high source impedance, then it is better to use the 1 MHz sample clock as this increases the input impedance of the ADC to 1 MΩ (it is around 500 kΩ at 6 MHz). Note however that, aside from this effect, there is no inherent difference in accuracy between the two clock sources.

Where the source impedance is large and cannot be reduced, for example when a large value resistor network is needed to keep the measured voltage within the input range of the ADC, it is possible to compensate for this source of error:

- By adding the same impedance (such as resistor divider) to the 1.8 V regulated supply and connecting this to an ADC input. The measured value can then compensate for the impedance error by

  correctedValue = 1800 * adcMeasuredValue/adcVregValue

- Alternatively, since the error is linear and constant, it should be possible to establish what this gain value is and adjust all measurements empirically to compensate.

Using longer conversion times will increase the resolution of the ADC output. The shortest sample time (32 clocks) produces seven significant bits in the output. Each doubling of the number of clocks produces an additional significant bit, up to a maximum of 4096 clocks (14 bits). Regardless of the conversion time, a 16 bit value is always produced, with the significant bits left-aligned within the result.

Finally, in general it is good practice to preserve ADC settings between those used for calibration and those used for measurement. It may also be worthwhile to recalibrate periodically, particularly in those applications where large variations in temperature are anticipated.

Glossary

Absolute: A measurement where a precise value for the analog voltage is desired.

Calibration: In the context of this ADC document, this refers to the process of verifying the value of the reference voltage and applying gain and offset correction to produce absolute ADC measurements.

Decimation: The process of converting a low resolution serial data stream into lower rate parallel data bits.

Differential: A measurement of the variation between two separate input signals.

Oversampling: Quantising an input data stream at a rate significantly higher than the Nyquist sample rate.

Quantisation: Approximating an analog value with a discrete digital value.

Single-Ended: An analog value measured against a stable voltage reference.

Ratiometric: A measurement where the voltage reference is also used to excite the sensor of interest.

Sigma-Delta: A modulation method to used to encode analog signals into a digital bitstream.

After reading this document

If you have questions or require assistance with the procedures described in this document, contact Ember Customer Support at http://www.ember.com/support_index.html.

ember

Copyright © 2011 Ember Corporation.

All rights reserved. Neither this publication nor any part thereof can be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Ember Corporation. This documentation is furnished under license and can be used or copied only in accordance with the terms of such license.

The content of this documentation is furnished for informational use only, is subject to change without notice, and does not represent a commitment or guaranty by Ember Corporation. The statements, configurations, technical data, and recommendations in this document are believed to be accurate and reliable as of the time of publication, but Ember Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT, ARE DISCLAIMED. Users are responsible for their applications and for the use of any products specified in this document.

Title, ownership, and all rights in copyrights, patents, trademarks, and other intellectual property rights embodied in Ember Corporation's Products and any copy, portion, or modification thereof, shall not transfer to Purchaser or its customers and shall remain in Ember Corporation and its licensors.

No source code rights are granted to Purchaser or its customers with respect to any Ember software. Purchaser agrees not to copy, modify, alter, translate, decompile, disassemble, or reverse engineer Ember hardware (including without limitation any embedded software) or attempt to disable any security devices or codes incorporated in Ember hardware. Purchaser shall not alter, remove, or obscure any printed or displayed legal notices contained on or in Ember hardware.

Ember is a trademark of Ember Corporation. All other trademarks are the property of their respective holders.