Application Note 5042
17 June 2008
120-5042-000E

Using the EM250 ADC

This document describes how to use the Analog to Digital Converter (ADC) in the EM250, gives an overview of the ADC API, provides both the general use case and common examples, and discusses design considerations.

Use this document as a general usage guide in conjunction with Ember's *Hardware Abstraction Layer (HAL) API Reference* and the *EM250 Datasheet*. Before reading this document, you should have a familiarity with these documents and ADCs in general.

Topics

EM250

ember

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
www.ember.com

wireless semiconductor solutions

EM250 ADC Overview

The EM250's Analog to Digital Converter (ADC) is a first-order sigma-delta converter sampling at 1MHz with a programmable resolution and conversion rate. A discussion of sigma-delta converter design is outside the scope of this document.

While the *EM250 Datasheet* provides a description of the ADC hardware including register bits and basic functionality, the Hardware Abstraction Layer (HAL) API provides a common and simplified interface to the ADC. The majority of use cases can be quickly and easily implemented using the HAL API and therefore the programmer does not require an intimate knowledge of the register bits.

The ADC has 10 input sources comprised of 4 single-ended inputs, 2 supply monitoring inputs, 2 calibration inputs, and 2 differential inputs. The 2 differential inputs are comprised of the 4 single-ended inputs. The 4 single-ended inputs are available on GPIO[7:4] and named ADC[3:0]. In addition, the calibration input VREF (provided by an internal reference supply) is available as an output on GPIO8 named VREF_OUT.

Note: For full descriptions of the ADC Module hardware and the EM250 GPIO configuration, refer to the *EM250 Datasheet*.

API Overview

The ADC API in the HAL can be considered a standalone API that is part of the *Hardware Abstraction Layer (HAL) API Reference*. The ADC API is contained in two files — the API definition and the implementation.

- `hal/micro/adc.h`
- `hal/micro/xap2b/adc.c`

The file `adc.h` not only includes the appropriate prototypes and type definitions, but it also contains extensive commenting that is used as the source information for the ADC documentation in the *Hardware Abstraction Layer (HAL) API Reference*.

Note: The file *adc.h* provides the generic ADC API for all platforms. However, this document focuses on the aspects of the API relevant to the EM250 ADC.

The ADC module provides five public functions.

- `halStartAdcConversion()`
- `halRequestAdcData()`
- `halReadAdcBlocking()`
- `halAdcCalibrate()`
- `halConvertValueToVolts()`

In addition, there are two documented internal functions:

- `halInternalInitAdc()`
- `halInternalSleepAdc()`

These two internal functions are shown here for completeness. The 'Internal' designation indicates they are required for proper stack operation and should never be called directly by an application. The ADC is required for calibration of the EM250 radio therefore the function `halInternalInitAdc()` must be called from the function `halInit()`. The function `halInternalSleepAdc()` is ultimately called from the function `halPowerDown()`. Refer to the *Hardware Abstraction Layer (HAL) API Reference* for more information on `halInit()` and `halPowerDown()`.

The ADC API provides a collection of three primary data types used in the functions to make the API more readable:

- `ADCUser` – Allows different ADC conversions to be interleaved and the data stored separately.
- `ADCRateType` – Selects the conversion rate to use.
- `ADCChannelType` – Selects the input channel to use.

Details about these data types are described later. A fourth data type, `ADCReferenceType`, is defined and used in some of the functions but this parameter cannot be changed for an EM250 and must always be set to `ADC_REF_INT`.

General ADC Use Case

This section provides a simplified function that illustrates the general use case of the EM250 ADC.

```
void adcRead(void)
{
  int16u data1, data2 = 0;
  int16s volts1, volts2 = 0;

  //Conversion 1
  halStartAdcConversion(ADC_USER_APP,
                        ADC_REF_INT,
                        ADC_SOURCE_ADC2_GND,
                        ADC_CONVERSION_TIME_US_256);
  halReadAdcBlocking(ADC_USER_APP, &data1);
  volts1 = halConvertValueToVolts(data1);
  emberSerialPrintf(0, "Read1 = 0x%2x = %d 10^-4 V\r\n", data1, volts1);

  //<temperature of the EM250 changes>

  //Conversion 2
  halAdcCalibrate(ADC_USER_APP);
  halStartAdcConversion(ADC_USER_APP,
                        ADC_REF_INT,
                        ADC_SOURCE_ADC2_GND,
                        ADC_CONVERSION_TIME_US_256);
  while(EMBER_ADC_CONVERSION_DONE !=
                               halRequestAdcData(ADC_USER_APP, &data2)) {
    //<other work while waiting>
  }
  volts2 = halConvertValueToVolts(data2);
  emberSerialPrintf(0, "Read2 = 0x%2x = %d 10^-4 V\r\n", data2, volts2);
}
```

When a 0.6V signal is applied to ADC2 (GPIO6, chosen with `ADC_SOURCE_ADC2_GND`) and this sample function executes, output on serial port 0 looks similar to the following:

```
Read1 = 0x7EEB = 5971 10^-4 V
Read2 = 0x7EE3 = 5968 10^-4 V
```

This example performs two conversions in slightly different ways. The ADC API does not support continuous samples, so every single conversion must be initiated by the function `halStartAdcConversion()` and the resulting value is read either through `halReadAdcBlocking()` or `halRequestAdcData()`. Use `halRequestAdcData()`whenever the application needs to perform other operations while a conversion finishes. The function `halConvertValueToVolts()` is optional to provide an ADC reading in volts. It performs any necessary calibration and returns the sampled data in units of 10^-4 Volts. Finally, call the utility function `halAdcCalibrate()`whenever the application suspects the EM250's temperature has changed.

This example makes three major assumptions:

- GPIO6 has been configured to operate as analog input ADC2.
- The function `halInit()` has already been called because it calls `halInternalInitAdc()`.
- The Ember 'printf' functionality is available and operating on serial port 0.

halStartAdcConversion()

Since the EM250 ADC API does not support continuous sampling, every conversion must begin with this function. This function accepts four parameters.

- `ADCUser id` – The ADC API supports interleaving measurements and storing these measurements separately. It does this by using the concept of an ADC user. An ADC user is simply an abstract name given to a block of storage area in the ADC. However, this storage area can only maintain a single reading value and configuration data. The ADC user construct allows not only multiple user conversions but also enables the Ember networking stack to interleave operations without disrupting the application (in fact, the application does not need to be aware of the stack's operation). The ADC API predefines two users that are available to applications:
    - `ADC_USER_APP`
    - `ADC_USER_APP2`
- `ADCReferenceType reference` – This parameter must always be set to `ADC_REF_INT` for the EM250 because the EM250 only has an internal reference of 1.2Volts. (Other platforms allow different references.)
- `ADCChannelType channel` – This parameter sets which input is used for taking a sample. The complete channel list is provided in the file `adc.h` and descriptions are provided in the *EM250 Datasheet*. Note that when using an external source (ADC[3:0]), the GPIO corresponding to those sources must be configured for analog input.
- `ADCRateType rate` – This parameter sets the rate of the conversion. Due to the nature of this sigma-delta ADC, the conversion rate directly corresponds to the number of effective data bits. The longer the conversion, the more effective bits, and the higher the sample accuracy. The complete list of rates is provided in the file `adc.h` and descriptions are provided in the *EM250 Datasheet*.

This function returns an `EmberStatus` return code. In many scenarios (such as this code example) the return code can be ignored. In practice it is a good idea to always check return codes to see if ADC reads have started properly.

halReadAdcBlocking()

This function accepts two parameters, `ADCUser` as described earlier and a pointer to a 16 bit variable. The `ADCUser` defines which read buffer to extract data from and the pointer defines where in memory the read data should be placed.

This function obtains the raw reading from the ADC hardware. Without context or calibration (explained later in `halConvertValueToVolts()` below), this data on its own can only be used by applications to monitor changes in readings. As the name implies, this function blocks until an ADC reading is finished. In many applications this is acceptable, but be aware that depending on the chosen conversion rate, this function could block application execution for up to8192 microseconds (two 4096 microsecond conversions).

This function returns an `EmberStatus` return code. In many scenarios (such as this code example) the return code can be ignored. In practice it is a good idea to always check return codes to see if ADC reads have started properly.

halRequestAdcData()

This function is identical to `halReadAdcBlocking()` with the exception that it always returns immediately with an `EmberStatus` return code which should always be checked. If the return code is `EMBER_ADC_CONVERSION_DONE`, the data is valid. Use this function in applications that cannot afford to simply block and wait for ADC data to become available.

halConvertValueToVolts()

This utility function converts the raw value obtained from the ADC hardware into a proper voltage. This function accepts one parameter, a 16 bit value. This input should be the raw reading obtained from the ADC hardware via either `halReadAdcBlocking()` or `halRequestAdcData()`. This function returns a signed, 16 bit value with the units 10^-4 Volts. The value is in the range of -12000 to +12000, representing -1.2000 to +1.2000 volts.

To generate a reading in volts, the ADC must be properly calibrated. If your application does not calibrate the ADC with a call to `halAdcCalibrate()`, the `halConvertValueToVolts()` function will internally call `halAdcCalibrate()`. However, `halConvertValueToVolts()` will only trigger calibration once after ADC initialization.

halAdcCalibrate()

This function accepts one parameter, `ADCUser`, which defines the read buffer being calibrated. An application can call this function anytime to recalibrate. In general, to guarantee results measured in voltage are as accurate as possible, call this function whenever you suspect the temperature of the EM250 changed. (The EM250 does not provide any internal means of determining temperature change.) Since the ADC relies on a constant internal voltage reference, the ADC is sensitive to only temperature change and not power supply voltage change.

Measuring the EM250 Power Supply Voltage

This section describes how to use the ADC to monitor the 2.1-3.6V pad power supply (labeled VDD_PADS on the EM250 Pin Assignment) and builds upon the earlier example in “General ADC Use Case” on page 3. This supply voltage provides the power for the entire chip and in most battery-powered designs directly represents the battery voltage. The ADC provides a momentary reading of the power supply voltage level.

The ADC can actually monitor two power supply voltages, VDD_CORE and VDD_PADS. The focus of this section is on the VDD_PADS supply which is the master 2.1-3.6V power supply to the whole chip. The VDD_CORE supply is the 1.72V core power supply. Since the core power supply is provided by the internal regulator and maintained as a constant, most applications will never need to measure the VDD_CORE supply.

Measuring VDD_PADS is performed in the same manner as measuring any other single-ended source ADC. The only major difference is that the VDD_PADS voltage routed to the ADC is internally divided by 4 to bring the signal within the ADC’s 1.2V limit. Therefore, the application simply needs to scale the reading back up to get a true voltage, as shown in the following code sample:

```
void adcReadVddPads(void)
{
  int16u data = 0;
  int16s volts = 0;

  halStartAdcConversion(ADC_USER_APP,
                        ADC_REF_INT,
                        ADC_SOURCE_VDD_PADS_GND,
                        ADC_CONVERSION_TIME_US_256);
  halReadAdcBlocking(ADC_USER_APP, &data);
  volts = halConvertValueToVolts(data);
  volts = (volts/10)*4; //shift decimal point right, and scale up
  emberSerialPrintf(0, "VDD_PADS = 0x%2x = %d 10^-3 V\r\n", data, volts);
}
```

When this code executes, it produces an output on serial port 0 that looks similar to:

```
VDD_PADS = 0x8D35 = 2920 10^-3 V
```

This result shows that VDD_PADS (the overall chip supply voltage) is currently at 2.920V. There are only two differences between this function and the sample function `adcRead()` shown on page 3. In this example:

- The function `halStartAdcConversion()` uses ADC_SOURCE_VDD_PADS_GND as the input channel for measurement. This channel selects the VDD_PADS source.

- The line of code, `volts = (volts/10)*4;` is added after converting the reading to volts. Since the VDD_PADS supply is internally scaled down by 4, the code must multiply it by 4 to scale it back up to proper volts. Additionally, since volts is only a 16 bit signed integer, the code divides by 10 so that the variable does not overflowed and the result can be easily interpreted as millivolts.

Depending on the application, the measurement of VDD_PADS may not need to be in a human readable form. In some scenarios, the application may be simply looking for the VDD_PADS supply voltage to drop below a known threshold (learned through testing). In this case, the raw reading may be used and the costly mathematics can be avoided.

The effectiveness of battery monitoring depends on the type of batteries in use. For example, alkaline batteries exhibit a consistently declining slope of supply voltage over time, so for alkaline batteries this monitoring is useful. However, lithium ion batteries have a fairly constant voltage supply over most of their lifetime with very steep drop-off in voltage prior to full drainage. Monitoring lithium ion batteries would not be as useful. Calculating remaining battery life is outside the scope of this document.

Differential Measurements

The majority of use cases for the ADC involve a single-ended measurement, such as reading a temperature sensor or measuring the EM250 power supply voltage. Some designs require making a differential measurement between two ADC inputs. The EM250 supports two differential sources for the ADC.

- The differential between analog inputs ADC0 and ADC1 (GPIO4 and GPIO5)
- The differential between analog inputs ADC2 and ADC3 (GPIO6 and GPIO7)

In general, most differential measurements rely on calculating changes between two different readings and are therefore relative values. A differential measurement is shown in the following example:

```
void adcReadDiff(void)
{
  int16u data = 0;

  halStartAdcConversion(ADC_USER_APP,
                        ADC_REF_INT,
                        ADC_SOURCE_ADC2_ADC3,
                        ADC_CONVERSION_TIME_US_256);
  halReadAdcBlocking(ADC_USER_APP, &data);
  emberSerialPrintf(0, "Reading = 0x%2x\r\n", data);
}
```

When this code executes it produces an output on serial port 0 that looks similar to:

```
Reading = 0xBF21
```

The only significant difference between this function call and the general example, `adcRead()`shown on page 3, is that this example uses `ADC_SOURCE_ADC2_ADC3` as the input channel for the measurement. Because differential measurements are relative, this sample function does not attempt to provide a voltage but simply provides the raw data reading. This input channel selects ADC2 (GPIO6) and ADC3 (GPIO7) as ADC inputs and takes a measurement of their difference. Of course GPIO6 and GPIO7 must both be configured as analog inputs.

Unfortunately, interpreting the meaning of differential measurements can become quite tricky, especially if a true voltage is desired. Relative values are easier to work with since they normally don't require "calibration" as with the `halAdcCalibrate()` function. Calibration in a differential scenario refers to determining the relationship between a differential measurement and an absolute voltage. The simplest method of performing this type of calibration is to first treat both inputs (ADC0 and ADC1 or ADC2 and ADC3) as single-ended inputs and measure them independently – as described earlier in this document. Doing so allows calculating the absolute voltage of both inputs and then determining the voltage differential between the two inputs.

Advanced Example

This section builds upon the "General ADC Use Case" on page 3, and provides a more advanced example of using the EM250 ADC to read a temperature sensor.

Note: This example reads the temperature sensor on the Ember EM250 Breakout Boards (P/N: 710-0455-000). Breakout Board revisions prior to A2 do not have circuitry to step down the voltage to below 1.2V, so developers wishing to use the onboard temperature sensor or any analog input with ranges greater than 1.2V will need to add divider circuitry to perform voltage translation.

The following example is derived from Ember's "Sensor-Sink" example application. This example is not meant to be an exhaustive discussion of the "sensor" application, but simply provides a context for using the ADC. The sensor application code may be found in your installation under this subdirectory:

`app/sensor/sensor.c`

The actual sensor demonstration application can be compiled and run. However, this example is not complete and is not intended to be compiled and run as is. The following code is extracted from `sensor.c` and has been significantly reduced.

```
//  sensor.c
//  sample app for Ember Stack API

int main(void)
{
  //Initialize the hal
  halInit();                                                  // 1.

  // event loop
  while(TRUE) {
      applicationTick(); // check timeouts, buttons, flash LEDs
  }
}

// applicationTick - called to check application timeouts, button events,
// and periodically flash LEDs
static void applicationTick(void)
{
    // *********************************************/
    // if we are gathering data (we have a sink)
    // then see if it is time to send data
    // *********************************************/
    if (mainSinkFound == TRUE) {
      if (sendDataCountdown == SEND_DATA_RATE) {              // 2.
        halStartAdcConversion(ADC_USER_APP2,
                              ADC_REF_INT,
                              SENSOR_ADC_CHANNEL,
                              ADC_CONVERSION_TIME_US_256);
      }
      sendDataCountdown = sendDataCountdown - 1;
      if (sendDataCountdown == 0) {
        //emberSerialPrintf(APP_SERIAL, "sending data...\r\n");
        sendDataCountdown = SEND_DATA_RATE;
        sendData();                                           // 3.
      }
    }

  }
}
```

```
// The sensor reads (or fabricates) data and sends it out to the sink
// There are four types of data that are sent:
// - Temperature data as BCD (binary coded decimal) - the default
// - Temperature data as a value
// - ADC Volts reading
// - random data
// The type of data sent depends on the dataMode variable.
void sendData(void)
{
  int16u data;
  int16s fvolts;
  int32s tempC;
  EmberStatus status;
  EmberMessageBuffer buffer = 0;
  int8u i;
  int8u sendDataSize = SEND_DATA_SIZE;

  switch (dataMode) {
    default:
    case DATA_MODE_RANDOM:
      // get a random piece of data
      data = halCommonGetRandom();
      break;
    case DATA_MODE_VOLTS:
    case DATA_MODE_TEMP:
    case DATA_MODE_BCD_TEMP:
      if(halRequestAdcData(ADC_USER_APP2, &data) ==
                              EMBER_ADC_CONVERSION_DONE) {      // 4.
        data = data / SENSOR_ADC_PLATFORM_ADJUSTMENT;
        fvolts = halConvertValueToVolts(data);                  // 5.
        if (dataMode == DATA_MODE_VOLTS) {
          data = (int16u)fvolts;
        } else {
          tempC = voltsToCelsius(fvolts) / 100;
          if (dataMode == DATA_MODE_TEMP) {
            data = (int16u)tempC;
          } else {
            data = toBCD((int16u)tempC);
          }
        }
      } else {
        data = 0xFBAD;
      }
      break;
   }

#ifdef DEBUG
  emberDebugPrintf("sensor has data ready: 0x%2x \r\n", data);
#endif
}

// Function to read data from the ADC, do conversions to volts and
// BCD temp and print to the serial port
void readAndPrintSensorData()                                   // 6.
{
  int16u value;
  int16s fvolts;
  int32s tempC, tempF;
  int8u str[20];
  EmberStatus readStatus;

  emberSerialPrintf(APP_SERIAL, "Printing sensor data...\r\n");
  halStartAdcConversion(ADC_USER_APP, ADC_REF_INT,
                        SENSOR_ADC_CHANNEL,
                        ADC_CONVERSION_TIME_US_256);
  emberSerialWaitSend(APP_SERIAL);
  readStatus = halReadAdcBlocking(ADC_USER_APP, &value);
```

```
   value = value / SENSOR_ADC_PLATFORM_ADJUSTMENT;

   if( readStatus == EMBER_ADC_CONVERSION_DONE) {
     fvolts = halConvertValueToVolts(value);
     formatFixed(str, (int32s)fvolts, 5, 4);
     emberSerialPrintf(APP_SERIAL, "ADC Voltage V = %s\r\n", str);

     tempC = voltsToCelsius(fvolts);
     formatFixed(str, tempC, 5, 4);
     emberSerialPrintf(APP_SERIAL, "ADC temp = %s celsius, ", str);

     tempF = ((tempC * 9) / 5) + 320000;
     formatFixed(str, tempF, 5, 4);
     value = toBCD((int16u)(tempF / 100));
     emberSerialPrintf(APP_SERIAL, "%sF %2xF)\r\n", str, value);
     emberSerialWaitSend(APP_SERIAL);

   } else {
     emberSerialPrintf(APP_SERIAL, "ADC read error: 0x%x\r\n",
                                                           readStatus);
     emberSerialWaitSend(APP_SERIAL);
   }
 }
```

The details of this code make more sense inside the full application, so only the basic algorithm is explained here.

1. The `main()` function must call `halInit()` which initializes the ADC.

2. The `applicationTick()` function, which is periodically triggered from the main loop, checks if the node is inside of a network and if a simple timer has expired. This timer indicates that the sensor should send fresh data. Since an ADC conversion is not instantaneous, before the timer completely expires the timer first triggers `halStartAdcConversion()` which will initiate the reading of the ADC.

3. After the timer has fully expired, the ADC conversion is expected to be complete and the node will send data by calling `sendData()`.

4. If the appropriate data mode is selected, `sendData()` will read the result of the ADC conversion using the function `halRequestAdcData()`. This function is used instead of `halReadAdcBlocking()` because if the ADC were not able to complete the conversion for some reason, the entire application will be able to continue operating.

5. After data is read, `sendData()`converts the raw data into whatever form is desired using helper utility functions. At the end of `sendData()`, the application can do whatever it needs to with the data, such as transmitting it across the network.

6. Finally, the function `readAndPrintSensorData()` shows a simple means for an end-user to manually acquire a data reading. This function is written to be invoked by a user, start a conversion, read the data in a blocking manner, convert the data to a human readable format, and present the data to the user through an Ember-style ‘printf’.

Design Considerations

Consider the following design issues:

- To reach the lowest current consumption possible during deep sleep, reconfigure any GPIO to digital input if it was configured as analog input for use by the ADC (GPIO[7:4], designated by the alternate functions ADC[3:0].)
- The EM250 is not capable of accepting an external reference voltage, but can expose the internal reference voltage. GPIO8 can operate as an alternate function called VREF_OUT. This alternate function puts the pin in the analog mode (similar to configuring GPIO[7:4] as ADC[3:0]). Unlike ADC[3:0], VREF_OUT is an analog output exposing the internal ADC reference voltage, 1.2V.
- Since the ADC module's input signal range is -1.2V to 1.2V, all external inputs (ADC[3:0]) must employ proper voltage division to guarantee they do not exceed this range.

After reading this document

If you have questions or require assistance with the procedures described in this document, please contact an Ember support representative at support@ember.com.

ember

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
www.ember.com

wireless semiconductor solutions