Application Report
SLAA475A – October 2010 – Revised March 2019

MSP430x09x Analog Pool: Feature Set and Advanced Use

Sebastian Fritz *MSP430 Europe*

ABSTRACT

The MSP430x09x is the first native 0.9-V device in the portfolio of the MSP430 family. It is designed to be supplied from a single coin cell with a voltage range between 1.65 V and 0.9 V. While the MSP430x09x is almost identical to other MSP430™ microcontrollers (MCUs) with respect to digital features and peripherals, the MSP430x09x introduces a new analog module named the analog pool (A-Pool). In addition to a general overview of the various modules found within the MSP430x09x family, this application report provides a detailed discussion of the analog pool.

Contents

List of Figures

List of Tables

Trademarks

MSP430 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

1 MSP430x09x Overview

The MSP430x09x allows the user to target applications such as motor control, IR communications, and power monitoring. The main differences between the MSP430x09x and other MSP430 MCUs are:

- Native 0.9-V device
- No memory that holds information over a power cycle
- Software ROM module to support loading code to and from external memory
- JTAG pins shared with I/O functionality
- Programmable analog module (A-Pool)
- Emulation mode for ROM code development
- Password protected ROM code

The MSP430x09x consists of several modules (see Figure 1). Two 16-bit timers are available and make it possible to have capture/compare functionality on every port pin. The compact clock system lets you set the correct application frequency. In addition, an analog pool (A-Pool) module is implemented. The A-Pool contains an 8-bit DAC, comparator, and surrounding logic. The A-Pool can be configured to support higher-level functions such as an 8-bit ADC or SVM. Furthermore, 11 I/O pins are available, four of which are used as JTAG communication pins by default. As a software module, the bootloader found in the MSP430L092 allows loading the application code from external memory.

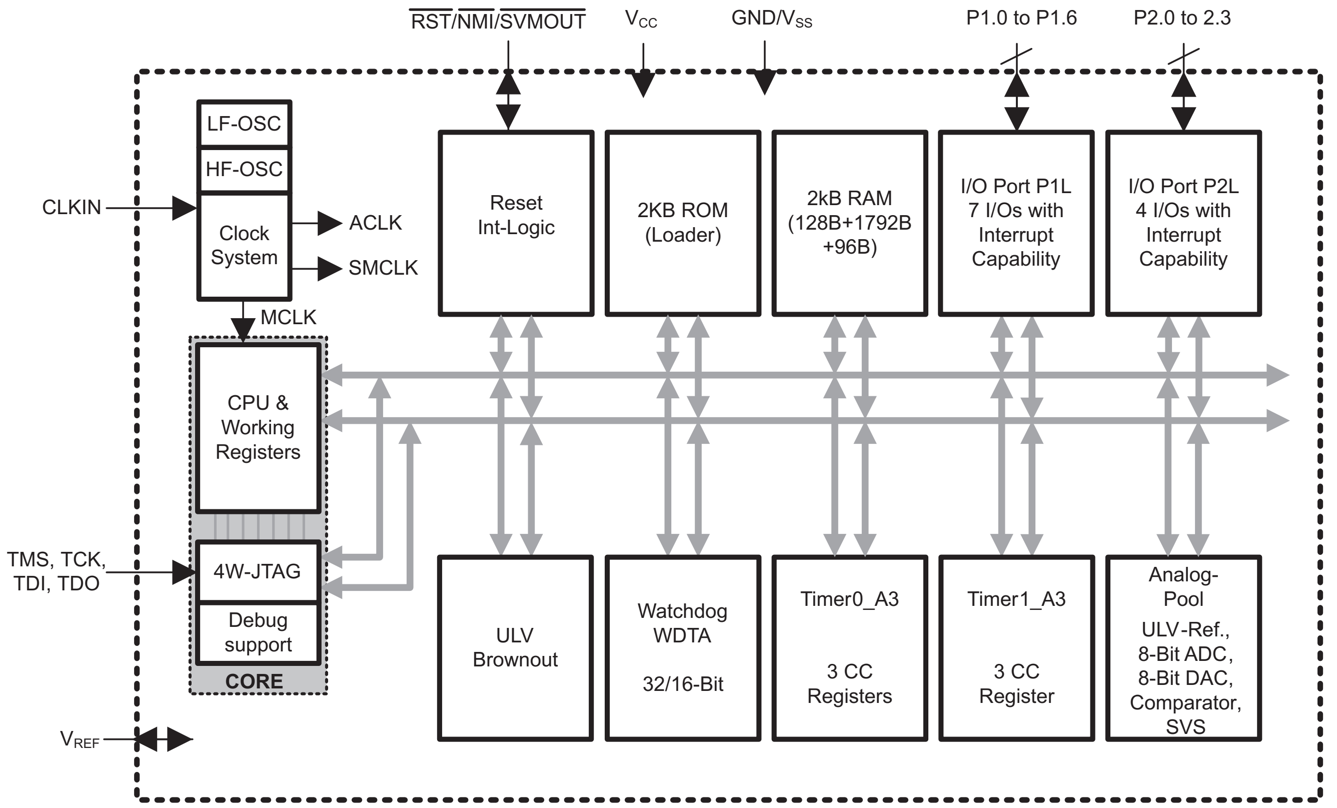

Figure 1. MSP430L092 Functional Block Diagram

2 Analog Pool (A-Pool)

The A-Pool module in the MSP430x09x supports several analog functions at 0.9 V, depending on the user software. The reference voltage for the A-Pool is supplied by an internal 256-mV reference or by an external reference that can be input through a port pin. The clock frequency can be selected from several clock sources and can also be divided within the A-Pool. The A-Pool provides these analog functions:

- Comparator
- 8-bit elementary DAC
- 8-bit ADC
- Supply voltage monitor (SVM)
- Temperature sensor
- Ultra-low-voltage reference

The core of the A-Pool is a comparator and two multiplexers. The multiplexers are used to select the signals for the positive and negative inputs of the comparator. In addition, a counter and specific start and stop logic are used for the ADC and DAC operations. The inputs to the A-Pool can be divided to select the correct voltage range. Table 1 shows the different levels for the input channels.

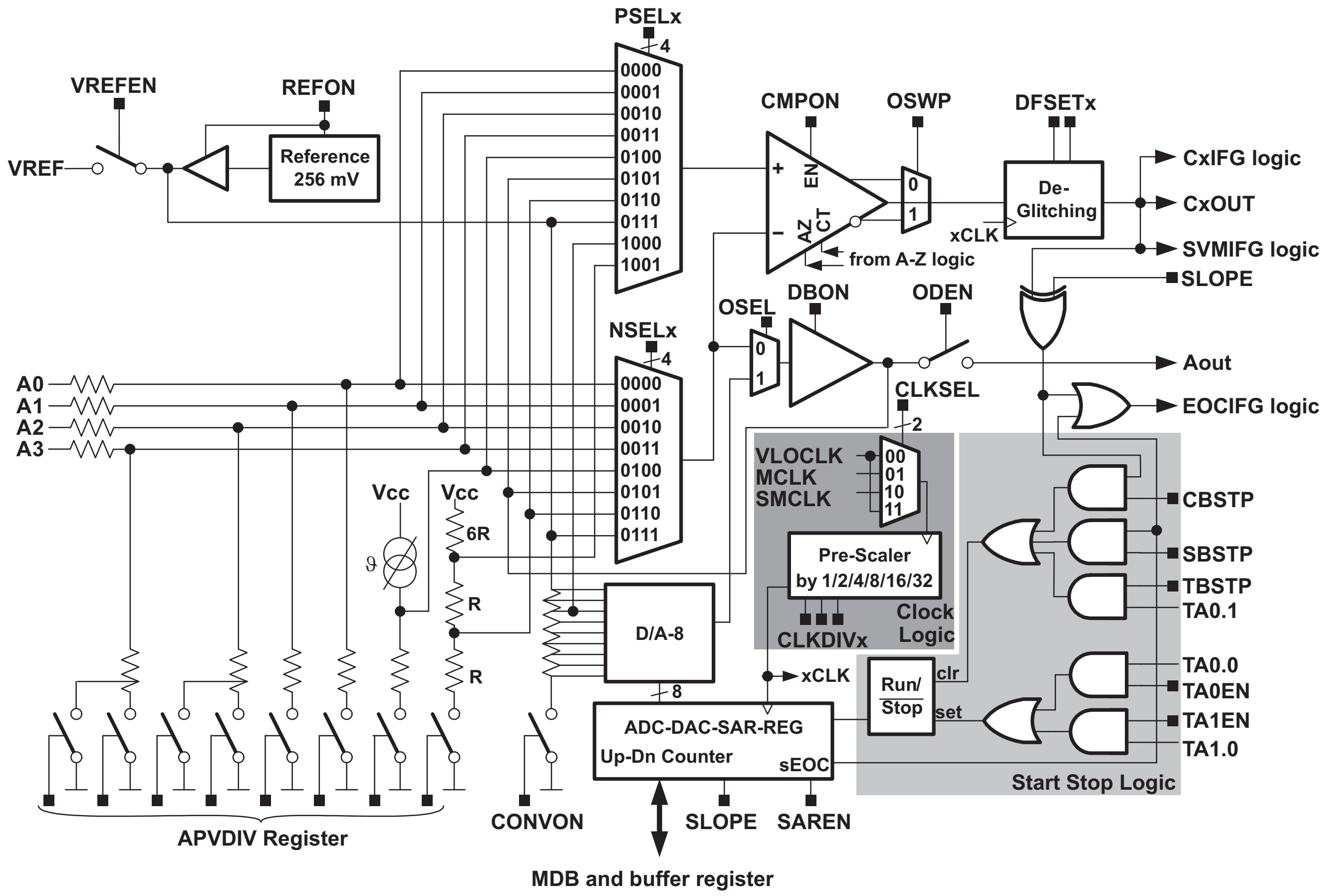

Figure 2. Functional Block Diagram of the Analog Pool

The A-Pool can be a powerful tool when used to implement and execute analog functions. To enable better performance and to avoid unintended behavior, the recommendations in this application report should be taken into account.

2.1 Input Dividers

The inputs to the A-Pool can be divided to select the correct voltage range (see Table 1). For measuring the supply voltage, an internal divider is also implemented. The inputs A0 to A4 have programmable dividers included to allow the measurement of a higher voltage range. The A0 and A1 input channel voltage range can be expand to 500-mV input level; the A2 and A3 input channel voltage range can be expand up to 2-V input level.

Table 1. Voltage Range per Channel

Input Channel	250-mV Input Range AxDIVx = 0	1-V Input Range AxDIVx = 1	2-V Input Range AxDIVx = 2
A0	✓	—	—
A1	✓	—	—
A2	✓	✓	✓
A3	✓	✓	✓

2.2 Internal Reference

Because of the internal reference update mechanism, the VREF can slightly change over time. To avoid this behavior and have better performance, add a 220-nF capacitor at the VREF pin. This increases the accuracy of the A-Pool module and should be used for high-performance measurements.

2.3 Starting and Stopping the A-Pool

The A-Pool is controlled by the user software. Depending on the analog function, the A-Pool can be started and stopped in several ways.

To start the A-Pool in a comparator mode the first time or after making any changes to the comparator settings, it is recommended to use the following code.

```
APCTL = APPSEL_x+APNSEL_x;      // Make new settings for comparator input
APCNF |= LCMP+CMPON;            // Trigger filter again +
                                // Start comparison
```

Depending on the filter settings, the comparison starts at one of these two lines. With a filter setting of 0 (DFSET = 0) the comparison starts immediately after the write of APCTL. In this case, the second line is not needed. With any other filter setting, the comparison starts with the execution of the second line, because the filter settings must be updated again. This can be done by a dummy write to the LCMP bit, which is bit 4 in the APCNF register.

The following code shows a comparator example with the internal DAC as the first input and an external voltage as the second input. The sample code also changes the inputs for the comparator to show the appropriate restart mechanism. A toggle on CxOUT output can be observed when a voltage higher than 0 is provided to the A1 input.

```
#include "msp430l092.h"
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR = BIT3;                             // Set CxOUT to port pin
  P1SEL0 = BIT3;
  P1SEL1 = BIT3;
  APINT = 0x00;                             // Set compare value to 0
  APCNF = CMPON+APREFON+DBON+CONVON+DFSET_2; // Comparator on +
                                            // Reference on +
                                            // Buffer on +
                                            // Conversion on +
                                            // Filter on
  while(1)
  {
    APCTL = APNSEL0+APNSEL2+OSEL;             // Set DAC as pos input +
                                            // Set A0 as neg input +
                                            // Switch DAC to multiplexer
    APCNF |= LCMP+CMPON;                      // Trigger filter again +
                                            // Start comparison
    APCTL = APPSEL0+APPSEL2+OSEL;             // Set DAC as neg input +
                                            // Set A0 as pos input +
                                            // Switch DAC to multiplexer
    APCNF |= LCMP+CMPON;                      // Trigger filter again +
                                            // Start comparison
  }
}
```

The DAC functionality is enabled by setting the CONVON bit to 1. Any change in the APINT or APFRACT register triggers an immediate change in the analog voltage that is either provided to the AOUT pin or used internally, depending on the A-Pool configuration.

The ADC functionality is enabled either by setting the RUNSTOP bit to 1 or by using external trigger sources for starting the ADC. The RUNSTOP bit can be used for a direct software start trigger and is located in the APCTL register. To avoid any unpredictable behavior, TI recommends always writing the APCTL register word-wise or with a bit set on the RUNSTOP register. Arithmetical or logical operations on the APCTL register, even with an unaffected RUNSTOP bit, can trigger unexpected behavior. The code snippets in this application report always use a word-wise write to the APCTL register.

For stopping an ADC conversion, any one of several methods can be used. A hard stop can be done either by setting the RUNSTOP bit to 0 or by using external triggers. In addition, the ADC stops at the measured value when the CBSTP bit is set to 1, or the ADC can be stopped at the highest or lowest value when SBSTP bit is set to 1, depending on the selected slope.

2.4 Comparator Function

To use the A-Pool as a comparator, the multiplexer must be configured correctly and the comparator itself must be enabled. Two flags (CRIFG and CFIFG) are generated for rising and falling edge detection, respectively, when the comparator is in clocked mode. Clocked mode is used when AZCMP bit is set to 1 or the AZSWREQ function is used. In CTEN mode, no rising or falling edge flags are generated, because the flag generation logic is disabled. To observe the comparator status in this mode, the CxOUT bit can be used.

A small digital filter is implemented for deglitching purposes. The user can decide between several filter options, from no filtering up to a majority vote of three out of five samples. The filtering consumes up to five additional comparator clocks and should be selected properly depending on the user application.

The comparator output signal itself is routed out to a dedicated port pin and can be used for external observations or as an input for additional hardware. To be more flexible, it is also possible to switch out the inverted comparator signal. Based on this signal, several flags such as the EOC flag or the rising and falling edge flags are generated (see Figure 3).

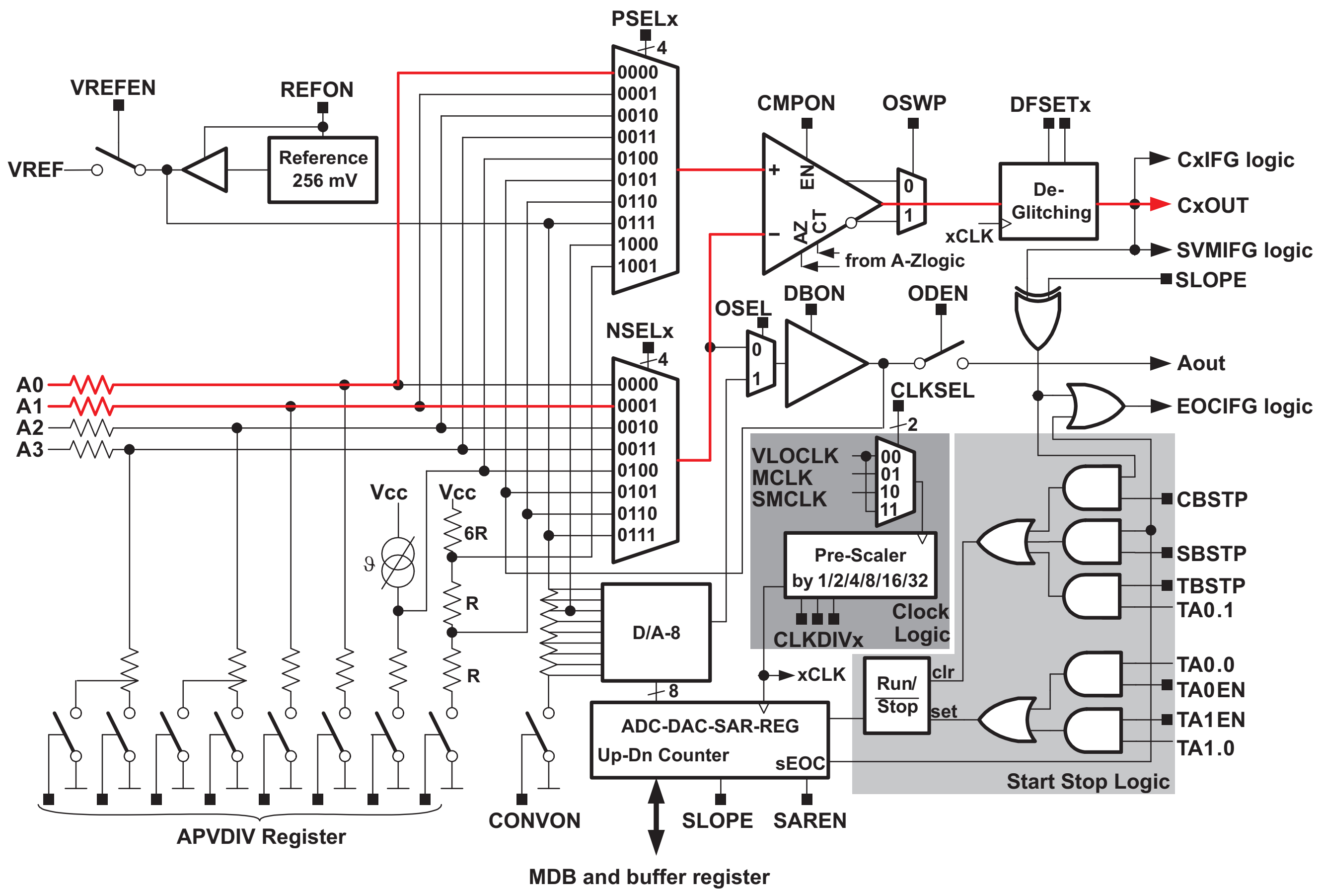

Figure 3. Example of Comparator Function

This following code shows how to configure a simple comparator function with A0 and A1 as inputs.

```
APCTL = APNSEL0;   // Set A0 as pos. input and A1 as neg. input
APCNF = CMPON;     // Switch on comparator
APOMR = AZCMP;     // Set comparator to clocked zero compensated long term comparison
```

Within this comparison mode it is possible to use the CRIFG and CFIFG flags for interrupt generation and comparator output change reaction.

The comparator is used in the clocked auto-zero compensation mode. This means that every two clock cycles the comparator offset is eliminated within the auto-zero phase. During this time, the comparator does not generate any output. To avoid this dead time, the comparator can be switched into a continuous enable mode, as shown in the following code.

```
APCTL = APNSEL0;   // Set A0 as pos. input and A1 as neg. input
APCNF = CMPON;     // Switch on comparator
APOMR = CTEN;      // Set comparator into continuous time mode for all operations
```

Using this mode, it is not possible to use the CRIFG and CFIFG flag, because the comparator is not in a clocked mode. To observe the output of the comparator, the COMPOUT bit can be used.

A clocked mode can be established in several ways. First would be the use of the auto-zero compensation mode, which is the default mode after a reset. Second would be the software request for an auto-zero phase. Third would be an ADC conversion with the implemented SAR logic.

2.5 8-Bit DAC Function

Within the MSP430x09x, an 8-bit DAC is implemented to generate voltages between 0 and 256 mV. The reference voltage for the DAC is the 256 mV generated internally from the reference module. The DAC voltage either can be used internally as an input for the comparator selected with the multiplexers or can be switched out on the AOUT pin. To set a voltage, the APINT register must be filled with the correct value, and the output buffer select (OSEL) and DAC buffer enable signal (DBON) bits must be set. To output the analog voltage on the port pin, it is necessary also to set the output driver (ODEN) bit. Any change of the APINT value is directly visible on the generated analog voltage.

A simple DAC code example follows. On the AOUT pin, a rising wave can be observed. Figure 4 shows the used paths through the A-Pool.

```
#include "msp430l092.h"
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = DBON+CONVON+APREFON;       // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APCTL = ODEN+OSEL;                 // Set DAC output to pin +
                                     // Select output buffer
  while (1)
  {
    APINT = APINT + 1;               // Increment APINT value
  }
}
```

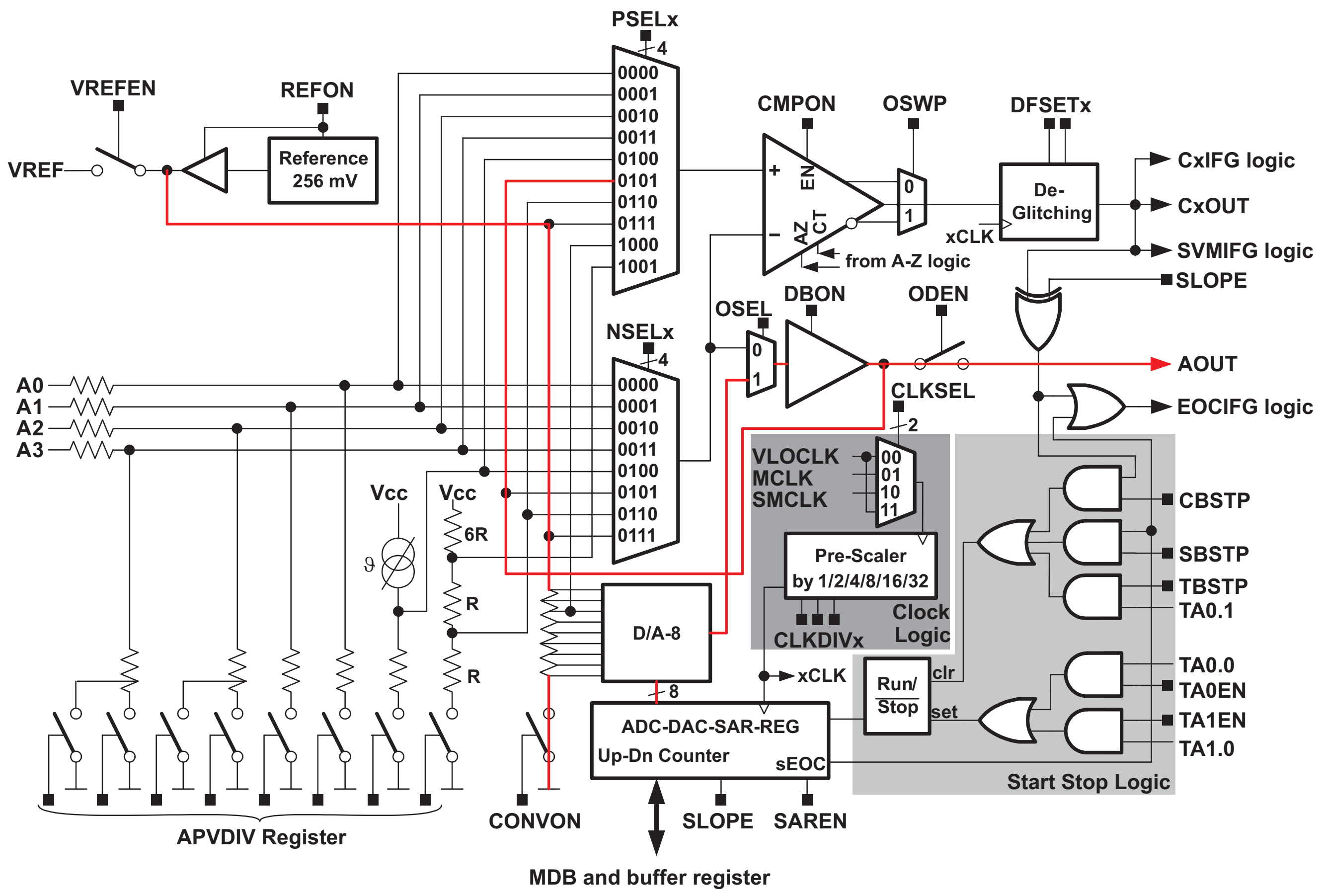

Figure 4. Example of DAC Function

2.6 8-Bit ADC Function

The MSP430x09x can be configured to perform one of two different types of 8-bit A/D conversions: a SAR logic is implemented or a ramp generator with the APINT counter register can be used. With both methods, the result is stored in the APINT register. For the DAC, comparator, and multiplexer blocks, the APINT register is used. One comparator input is the signal to be measured, and the other signal is the DAC output. To observe the internal signal that is used for the A/D conversion, the path to the AOUT pin can be enabled by setting the OSEL, DBON, and ODEN bits to 1.

2.6.1 ADC Conversion Using Ramp

The APINT counter starts at the current APINT value and counts either up or down, depending on the status of the SLOPE bit. When the comparator output signal changes, an end-of-conversion signal is generated and can be used to stop the counter (see Figure 5). The value in the APINT register is then the measured value of the input signal. When using the ramp generator method, four different methods are possible to recalculate internal offsets and errors that are coming from the A-Pool module itself. For example, the comparator has always a little offset, which can create either higher or lower result values. In addition, an overdrive error can occur out of the internal logic of the A-Pool. With a high A-Pool input frequency, the module might not react fast enough to stop the counter correctly.

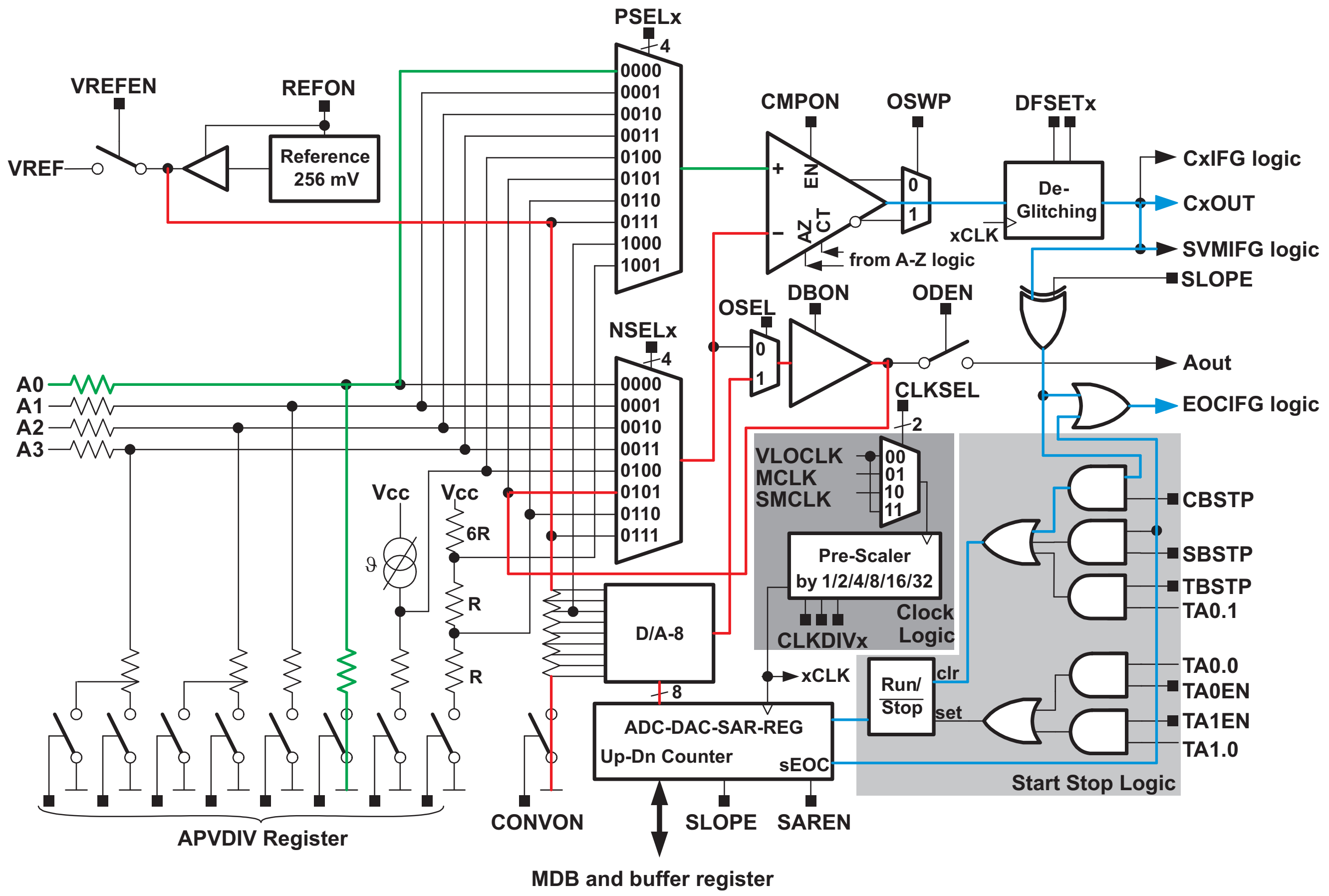

Figure 5. Example of ADC With External Signal and Counter Stop Logic

2.6.1.1 ADC Conversion Without Error Compensation

An ADC conversion for digital filtering can be implemented with a single ramp conversion. This measurement method should be preferred when an exact accuracy is not so important and if the general voltage range of the input signal should be identified. The counter starts at 0 and counts up to the measured voltage. In this method, no error compensation is included. The following example code shows a single ramp conversion from A0 in a 500-mV range. A single conversion is done, and the value is saved in the result variable.

```
#include "msp430l092.h"
unsigned char result;
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APVDIV = A0DIV;                    // Set 500mV input range
  APINT = 0x00;                      // Clear ADC-DAC-REG
  APIE |= EOCIE;                     // Enable end of conversion interrupt
  _BIS_SR(GIE);                      // Switch on global interrupts
  APCTL = CBSTP+SBSTP+APPSEL0+APPSEL2+OSEL;
                                     // Set DAC buffer output to PSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
  APCTL |= RUNSTOP;                  // Start conversion
  while (1);
}
#pragma vector=APOOL_VECTOR          // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))    // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result = APINT;           // Save value in variable
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.6.1.2 ADC Conversions With Overdrive Compensation

The APINT or APFRACT counter that is used to save the result does not stop immediately after the comparator changes its state. Especially with a high-frequency A-Pool input clock, a significant error may occur. To avoid this counting error, it is possible to use the ADC in an overdrive-compensation mode, as shown in the following code.

```
#include "msp430l092.h"
unsigned char result[2];              // result array
unsigned char i = 0;                  // counting variable
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APVDIV = A0DIV;                     // Set 500mV input range
  APINT = 0x00;                       // clear ADC-DAC-REG
  APIE |= EOCIE;                      // Enable end of conversion interrupt
  _BIS_SR(GIE);                       // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP;
                                     // Set DAC buffer output to PSEL +

                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reverence +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
  APCTL |= RUNSTOP;                   // Start conversion
  _BIS_SR(LPM0);                      // Go to LPM0
  APINT = APINT + 0;                  // Add an offset for counting
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP+SLOPE;
                                     // Set DAC buffer output to PSEL +

                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reverence +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
                                     // Switch to falling slope +
  APCTL |= RUNSTOP;                   // Start conversion
  _BIS_SR(LPM0);                      // Go to LPM0
  asm("nop");
  while(1);
}
#pragma vector=APOOL_VECTOR           // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))     // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result[i++] = APINT;                // Save value in result array
           __bic_SR_register_on_exit(CPUOFF); // Exit LPM0
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

Both values are stored in the result array for later use. The user application must make sure that the start value of counting is in the correct range. The addition of a fixed number to the APINT or APFRACT register can create problems when the measured value is near the upper or lower counting border. The user application must avoid an overflow of the APINT or APFRACT register.

2.6.1.3 ADC Conversions With Offset Compensation

The comparator itself can have an internal offset. This offset can cause a significant error in the results. To compensate for this error, the ADC can be used in an offset-compensation mode. In this mode, it is necessary to make two measurements with changed inputs. The following code shows how to implement this.

```
#include "msp430l092.h"
unsigned char result[2];              // result array
unsigned char i = 0;                  // counting variable
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APVDIV = A0DIV;                    // Set 500mV input range
  APINT = 0x00;                      // Clear ADC-DAC-REG
  APIE |= EOCIE;                     // Enable end of conversion interrupt
  _BIS_SR(GIE);                      // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP;
                                     // Set DAC buffer output to PSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  APINT = 0x00;                      // clear ADC-DAC-REG
  APCTL = APNSEL0+APNSEL2+OSEL+CBSTP+SBSTP+OSWP;
                                     // Set DAC buffer output to NSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
                                     // Inverted comparator output is used +
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  asm("nop");
  while(1);
}
#pragma vector=APOOL_VECTOR          // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))    // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result[i++] = APINT;      // Save value in result array
           __bic_SR_register_on_exit(CPUOFF); // Exit LPM0
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.6.1.4 ADC Conversions With Overall Compensation

This compensation includes all compensation methods previously mentioned and allows the most accurate measurement. The internal offset and the overdrive error are handled by the following code.

```
#include "msp430l092.h"
unsigned char result[4];              // result array
unsigned char i = 0;                  // counting variable
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APVDIV = A0DIV;                      // Set 500mV input range
  APINT = 0x00;                        // clear ADC-DAC-REG
  APIE |= EOCIE;                       // Enable end of conversion interrupt
  _BIS_SR(GIE);                        // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP;
                                       // Set DAC buffer output to PSEL +

                                       // Enable DAC buffer +
                                       // Enable conversion +
                                       // Enable reverence +
                                       // Select output buffer +
                                       // Enable Comparator based stop +
                                       // Enable Saturation based stop +
  APCTL |= RUNSTOP;                    // Start conversion
  _BIS_SR(LPM0);                       // Go to LPM0
  APINT = APINT + 0;                   // Add an offset for counting
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP+SLOPE;
                                       // Set DAC buffer output to PSEL +

                                       // Enable DAC buffer +
                                       // Enable conversion +
                                       // Enable reverence +
                                       // Select output buffer +
                                       // Enable Comparator based stop +
                                       // Enable Saturation based stop +
                                       // Switch to falling slope +
  APCTL |= RUNSTOP;                    // Start conversion
  _BIS_SR(LPM0);                       // Go to LPM0
  APINT = 0x00;                        // clear ADC-DAC-REG
  APCTL = APNSEL0+APNSEL2+OSEL+CBSTP+SBSTP+OSWP;
                                       // Set DAC buffer output to NSEL +
                                       // Select output buffer +
                                       // Enable Comparator based stop +
                                       // Enable Saturation based stop +
                                       // Inverted comparator output is used +
  APCTL |= RUNSTOP;                    // Start conversion
  _BIS_SR(LPM0);                       // Go to LPM0
  APINT = APINT + 0;                   // Add an offset for counting
  APCTL = APNSEL0+APNSEL2+OSEL+CBSTP+SBSTP+OSWP+SLOPE;
                                       // Set DAC buffer output to NSEL +
                                       // Select output buffer +
                                       // Enable Comparator based stop +
                                       // Enable Saturation based stop +
                                       // Inverted comparator output is used +
                                       // Switch to falling slope +
  APCTL |= RUNSTOP;                    // Start conversion
  _BIS_SR(LPM0);                       // Go to LPM0
  asm("nop");
  while(1);
}
#pragma vector=APOOL_VECTOR            // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))      // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result[i++] = APINT;        // Save value in result array
           __bic_SR_register_on_exit(CPUOFF); // Exit LPM0
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.6.1.5 Windowed ADC Conversion

If the voltage range of the input voltage is known, the ADC can also be used in a windowed mode. The counting of the ADC starts at a specific value and decreases the runtime of the ADC. For a comparison of the conversion methods, see Section 2.6.4. For an input voltage in a range of 250 mV to 500 mV, the following code can be used.

```
#include "msp430l092.h"
unsigned char result;
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reverence
  APVDIV = A0DIV;                    // Set 500mV input range
  APINT = 0x80;                      // Start measurement at 250mV
  APIE |= EOCIE;                     // Enable end of conversion interrupt
  _BIS_SR(GIE);                      // Switch on global interrupts
  APCTL = CBSTP+SBSTP+APPSEL0+APPSEL2+OSEL;
                                     // Set DAC buffer output to PSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  asm("nop");
  while (1);
}
#pragma vector=APOOL_VECTOR          // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))    // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result = APINT;           // Save value in variable
           __bic_SR_register_on_exit(CPUOFF);// Exit LPM0

           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.6.2 ADC Conversion Using SAR

In addition to the ramp method, a SAR conversion is also implemented. Within 8 clock cycles, the SAR implementation reaches the measured value with an accuracy of 1 bit. The SAR logic does not compensate for any internal offsets of the A-Pool module. The following code shows the settings to use the SAR logic.

```
#include "msp430l092.h"
unsigned char result;                // Result variable
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                      // Enable conversion +
                                      // Enable reference
  APVDIV = A0DIV;                     // Set 500mV input range
  APIE |= EOCIE;                      // Enable end of conversion interrupt
  APOMR |= SAREN;                     // Enable SAR logic
  _BIS_SR(GIE);                       // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+RUNSTOP;
                                      // Set DAC buffer output to PSEL +
                                      // Select output buffer +
                                      // Start conversion
  _BIS_SR(LPM0);                      // Go to LPM0
  asm("nop");
  while(1);
}
#pragma vector=APOOL_VECTOR           // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))     // Add offset to PC and delete flag
  {
    case  0: break;
    case  2: result = APINT;          // Save value in variable
             break;
    case  4: break;
    case  6: break;
    case  8: break;
    default: break;
  }
}
```

2.6.3 Multiple ADC Conversions

The A-Pool can sample up to four external analog voltages with individual voltage dividers for each input channel. The conversions are done sequentially, and the user applications must select the sample channel. The APINTB register can be used to save the last sampled value during the conversion of the next channel.

The following code shows the sample of three channels (A0 to A2) with different input dividers.

```
#include "msp430l092.h"
unsigned char result[3];             // result array
unsigned char i = 0;                 // counting variable
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON+EOCBU;
                                     // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference +
                                     // Enable EOC buffer
  APVDIV = A0DIV+A1DIV+A2DIV0;       // Set A0 to 500mV input range
                                     // Set A1 to 500mV input range
                                     // Set A2 to 1V input range
  APINT = 0x00;                      // Clear ADC-DAC-REG
  APIE |= EOCIE;                     // Enable end of conversion interrupt
  _BIS_SR(GIE);                      // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP;
                                     // Set DAC buffer output to PSEL +
                                     // Set A0 to NSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  APINT = 0x00;                      // clear ADC-DAC-REG
  APCTL = APPSEL0+APPSEL2+APNSEL0+OSEL+CBSTP+SBSTP;
                                     // Set DAC buffer output to PSEL +
                                     // Set A1 to NSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
                                     // Inverted comparator output is used +
  result[0] = APINTB;                // Save first value
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  APINT = 0x00;                      // clear ADC-DAC-REG
  APCTL = APPSEL0+APPSEL2+APNSEL1+OSEL+CBSTP+SBSTP;
                                     // Set DAC buffer output to PSEL +
                                     // Set A1 to NSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
                                     // Inverted comparator output is used +
  result[1] = APINTB;                // Save second value
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  result[2] = APINTB;                // Save third value
  APCTL |= RUNSTOP;                  // Start conversion
  _BIS_SR(LPM0);                     // Go to LPM0
  asm("nop");
  while(1);
}
#pragma vector=APOOL_VECTOR          // A-Pool interrupt service routine
```

```
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))    // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: __bic_SR_register_on_exit(CPUOFF); // Exit LPM0
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.6.4 Comparison Between Different Measurement Methods

To compare the different methods regarding accuracy and speed, a few basic conditions must be taken into account. The parameters for the following comparisons are:

- Because both program execution time and measurement time are relevant for the comparison, the program and ADC are supplied with the same clock speed.
- The input value that is measured with the different methods is in the middle of the measurement range. This means the correct value is 0x80 in integer mode.
- The internal reference is used. One count step equals 1 mV.
- The Program Execution Clock Cycles column in Table 2 shows all clock cycles that are necessary to execute the measurement. No interrupt service routines or additional overhead, such as clock module settings, are included. The program execution cycle counter was determined by the cycle counter of the IDE.
- The compared programs are not using filters.
- All observations and measurement are done at room temperature.
- The treated errors are only errors that can occur because of the design structure. In addition, the ADC values can be changed by noise.

Table 2 shows comparison of the conversion methods regarding execution speed and measurement accuracy.

Table 2. Comparison Between Different Measurement Methods

Measurement Method	Maximum Error	Program Execution Clock Cycles	Measurement Only Clock Cycles ± Error	Overall Clock Cycles Needed
Ramp without compensation	±3	44 (70)	128 ± 3	169 to 175
Ramp with offset compensation	±1	68 (150)	2 × (128 ± 1)	322 to 326
Ramp with overdrive compensation	±2	68 (142)	2 × (128 ± 2)	320 to 328
Ramp with overall compensation	±0	138 (288)	(2 × (128 ± 2)) + (2 × (128 ± 1))	692 to 704
Windowed ramp without compensation (starting point 0x70)	±3	44 (71)	16 ± 3	57 to 63
Windowed ramp with overall compensation (starting points 0x70 and 0x90)	±0	152 (288)	(2 × (16 ± 2)) + (2 × (16 ± 1))	210 to 222
SAR logic	±1	39	8	47

2.6.5 Error Dependencies

The errors themselves have different dependencies, such as the reference voltage or the A-Pool clock.

The overdrive error is dependent on the A-Pool clock frequency. A high frequency increases the possibility that an overdrive error occurs. With the highest clock frequency, the maximum error is 1 voltage step.

The internal reference is refreshed on a regular time base. Between the refresh cycles, the reference can have a slight drift, which can have an impact on the measurement. The update frequency of the reference is selected to meet the specification limits shown in the data sheet.

Depending on the input divider for the signal, the resolution of the signal is limited. The accuracy changes with divider settings. Table 3 shows these dependencies.

Table 3. Dependencies of Divider-to-Signal Accuracy

Divider Setting	Maximum Accuracy of Input Signal
1	1 mV
2	2 mV
4	4 mV

2.7 SVM Function

The A-Pool can be used to implement a supervision voltage monitor function. The integrated V_{CC} divider provides the V_{CC} divided by 8 or divided by 4 for better observation accuracy. A nearly full battery can be easily observed by the V_{CC} divided by 8. Lower battery voltages should be observed with the V_{CC} divided by 4 to get a higher resolution of the supply voltage.

The customer has the choice between a comparator-based and an ADC-based implementation. Using the ADC, the application can measure the internal V_{CC} voltage and can take actions depending on the measured value. The comparator-based solution compares the desired voltage level with the divided V_{CC} voltage.

To generate an SVM, the divided V_{CC} input must be connected to the comparator. Furthermore, the VCCDIVEN bit must be set to 1 to enable the internal V_{CC} ladder. The measured value shows the current V_{CC} value divided by the selected divider. The following code shows a simple V_{CC} observation using the comparator functionality. An SVMIFG flag is generated when the voltage falls below the selected limit. Unlike the SVM functions known from other MSP430 devices, this SVM implementation triggers only one event at the moment when the V_{CC} crosses the selected limit. If the V_{CC} level is below this limit, no additional SVM flag is generated by the logic.

```
#include "msp430l092.h"
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR |= BIT0;                     // Indicates VCC crosses SVM level
  APOMR |= CTEN;                     // Enable CTEN mode
  APINT = 163;                       // Set voltage level 1300mV / 8 = 163mV
  APVDIV |= VCCDIVEN;                // Enable VCC divider
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APCTL = APPSEL2+APPSEL1+APNSEL2+APNSEL0+OSEL;
                                     // Set voltage divider to PSEL +

                                     // Set DAC output to NSEL +
                                     // Select output buffer
  SFRIFG1 &=~ SVMIFG;                // Clear SVM flag
  APCNF |= CMPON;                    // Start comparison
  while(1)
  {
    if (SFRIFG1 & SVMIFG)            // Check if SVM flag is set
    {
      P1OUT ^= BIT0;                 // Indicates VCC crosses SVM level
      SFRIFG1 &=~ SVMIFG;            // Clear SVM flag
    }
  }
}
```

2.8 Use of Multiple Features

The A-Pool is not limited to providing only one of the available functions in an application. The user can decide when to provide a specific function for the application. For this, the timers or a software-based solution can be used. The following code shows a simple A/D conversion and SVM monitoring with software triggers. The observed V_{CC} voltage is 1.25 V, and the measured channel is A0 with 500-mV range.

```
#include "msp430l092.h"
unsigned char result;
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR |= BIT0;                     // Indicates VCC crosses SVM level
  APVDIV |= A0DIV+VCCDIVEN;          // Enable VCC divider
                                     // Set 500mV input range
  _BIS_SR(GIE);                      // Switch on global interrupts
  APCNF = CMPON+DBON+CONVON+APREFON; // Enable comparator on +
                                     // Enable DAC buffer +
                                     // Enable conversion +
                                     // Enable reference
  APOMR |= CTEN;                     // Enable CTEN mode
  APIE |= EOCIE;                     // Enable end of conversion interrupt
  while(1)
  {
    APINT = 156;                     // Set voltage level 1250mV / 8 = 156mV
    APCTL = APPSEL2+APPSEL1+APNSEL2+APNSEL0+OSEL;
                                     // Set voltage divider to PSEL +

                                     // Set DAC output to NSEL +
                                     // Select output buffer
    SFRIFG1 &=~ SVMIFG;              // Clear SVM flag
    APCNF |= CMPON;                  // Start comparison
    if (SFRIFG1 & SVMIFG)            // Check if SVM flag is set
    {
      P1OUT ^= BIT0;                 // Indicates VCC crosses SVM level
      SFRIFG1 &=~ SVMIFG;            // Clear SVM flag
    }
    APINT = 0x00;                    // Clear ADC-DAC-REG
    APCTL = CBSTP+SBSTP+APPSEL0+APPSEL2+OSEL;
                                     // Set DAC buffer output to PSEL +
                                     // Select output buffer +
                                     // Enable Comparator based stop +
                                     // Enable Saturation based stop +
    APCTL |= RUNSTOP;                // Start conversion
    _BIS_SR(LPM0);                   // Go to LPM0
  }
}
#pragma vector=APOOL_VECTOR          // A-Pool interrupt service routine
__interrupt void APOOL_ISR(void)
{
  switch(__even_in_range(APIV,8))    // Add offset to PC and delete flag
  {
  case  0: break;
  case  2: result = APINT;           // Save value in variable
__bic_SR_register_on_exit(CPUOFF);   // Exit LPM0
           break;
  case  4: break;
  case  6: break;
  case  8: break;
  default: break;
  }
}
```

2.9 Temperature Measurements With the A-Pool

Like other MSP430 ADCs, the A-Pool has an integrated temperature sensor that can be used to observe the current temperature. To do so, select one of the ADC measurement methods (uncompensated, compensated ramp, or SAR) and set the internal temperature sensor to one of the comparator inputs. In addition, the temperature sensor itself must be enabled by setting the TMPSEN bit to 1. The measured ADC value represents the current temperature.

2.10 Fractional and Integer Number Use

The A-Pool can perform calculations using either unsigned integer or fractional numbers. For this, the user has the choice between the APINT and APFRACT registers, which are both mapped to the same logic 16-bit register. The APINT register represents the lower 16 bits of this register, and the APFRACT register represents the upper 16 bits. Writing to one of these registers also enables the preferred mode.

2.11 APINTB and APFRACTB Use With ATBU and EOCBU

These two registers are implemented for advanced use. In combination with the EOCBU bit, a measured value can be stored in a buffer for later treatment. This is useful when multi-channel ADC conversions are being performed, for example, to handle the newest value while the next ADC conversion is running in parallel.

In addition, the registers can be also used to provide new data on a regular time base. Using the ATBU bit and the capture compare register of the Timer_A0 (TA0CCR1), the new data can be written from the buffer to the count register on a regular basis. This can be used, for example, to provide new data to the DAC before the new analog value is needed.

2.12 A-Pool Trigger Sources

The A-Pool has three trigger sources. The triggers can be used to provide specific analog voltages on a dedicated time and can be also used to stop any A-Pool activity.

For stopping the A-Pool, it is possible to use the Timer_A0 capture/compare register 1 (CCR1). To do so, the stop signal must be generated by the application, and the TBSTP bit must be set to one.

Two timer capture compare registers (Timer_A0 CCR0 and Timer_A1 CCR0) can be used to start the A-Pool. To do so, the TA0EN and TA1EN bits must be set accordingly.

The following code shows how to start and stop the A-Pool with Timer_A0. The starting and stopping ADC ramps can be observed on the AOUT pin. The ADC ramp does not stop at a specific voltage level, because no SBSTP or CBSTP bits are set.

```
#include "msp430l092.h"
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  TA0CCR0 = 600;                        // Set start value to 600
  TA0CCR1 = 300;                        // Set stop value to 300
  TA0CCTL0 = OUTMOD_3;                  // Set CCR0 outmode to Set/Reset
  TA0CCTL1 = OUTMOD_3;                  // Set CCR0 outmode to Set/Reset
  TA0CTL = TASSEL_2+MC_1+TACLR;         // Set SMCLK to timer clock source +
                                        // Set timer to up mode +
                                        // Clear timer count register
  APCNF = CMPON+DBON+CONVON+APREFON+TA0EN;
                                        // Enable comparator on +
                                        // Enable DAC buffer +
                                        // Enable conversion +
                                        // Enable reference +
                                        // Enable TimerA0 start
  APCTL = APPSEL0+APPSEL2+OSEL+ODEN+TBSTP;
                                        // Set DAC buffer output to PSEL +
                                        // Select output buffer +
                                        // Enable output driver +
                                        // Enable timer based stop
  while (1);
}
```

2.13 Filtering ADC Conversions With Digital Filters

For ADC conversions that use the ADC ramp mechanism, it can make sense to use the internal digital filter to get more stable values. To do so, the digital filters must be triggered each time the measurement method is changed. Similar to the comparator function, the application must provide an additional write to LCMP of the APCNF register. The following code shows how this can be done.

```
#include "msp430l092.h"
void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  APCNF = CMPON+DBON+CONVON+APREFON+DFSET_1;
                                       // Enable comparator on +
                                       // Enable DAC buffer +
                                       // Enable conversion +
                                       // Enable reference +
     ...
     .
  ...
  _BIS_SR(GIE);                        // Switch on global interrupts
  APCTL = APPSEL0+APPSEL2+OSEL+CBSTP+SBSTP;
                                       // Set DAC buffer output to PSEL +

                                       // Enable DAC buffer +
                                       // Enable conversion +
                                       // Enable reverence +
                                       // Select output buffer +
                                       // Enable Comparator based stop +
                                       // Enable Saturation based stop +
  APCNF |= LCMP;                       // Trigger filter again
  APCTL |= RUNSTOP;                    // Start conversion
  _BIS_SR(LPM0);                       // Go to LPM0
  ...
```

As a side effect, the filter can increase the deviation of the measured values depending on the selected filter settings. This must be taken into account when using these measured values. These filter methods also manage any possible deviation from the filter itself.

When the SAR logic is used for measurement, filtering is not useful, because of the way the SAR logic works. Wrong measurement values are the consequence of using filters in combination with the SAR logic.

3 Summary

The modules of the MSP430x09x enable the user to cover a great bandwidth of applications. With the A-Pool, the MSP430x09x contains a very powerful module that can be easily used for several kinds of analog operations. The use is highly software based, and the internal components make the use easy. In addition, the A-Pool is the first analog circuit that is designed for this low voltage range and that delivers accurate values and results. Because of its features, the A-Pool can be used in many customer applications, especially for single-battery applications without any external boost functionality necessary.

4 References

1. MSP430L092, MSP430C09x mixed-signal microcontrollers data sheet
2. MSP430x09x Family User's Guide

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from October 21, 2010 to March 18, 2019	Page
• Format updates and editorial changes throughout document	1