Application Report

SPRA786–January 2010

Programming the TMS320VC5509 ADC Peripheral

ABSTRACT

This application note demonstrates the procedure used to program the TMS320VC5509 ADC peripheral. Basic operation of the ADC consists of the ADC initialization and conversion of an arbitrary voltage into a digital value. This operation is illustrated using the ADC module provided in the chip support library (see TMS320C55x Chip Support Library API Reference Guide (SPRU433A)). Section 3 provides an example that allows you to determine which key was pressed on a keypad. Pressing a key generates DC voltage, and the ADC converts the DC voltage into its digital representation.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www-s.ti.com/sc/techlit/spra786.zip.

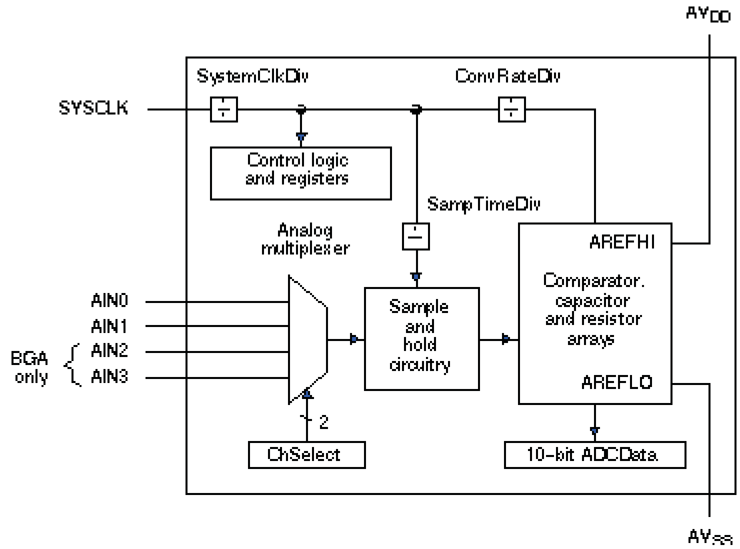
Contents

List of Figures

1 Introduction

The ADC module (Figure 1) converts an analog input signal to a digital value for use by the DSP. The ADC can sample one of up to four inputs (AIN0-AIN3) at a time, and generates a 10-bit digital representation (ADCData) of the samples. The maximum sampling rate of the ADC is 21.5 KHz. This performance makes the ADC suitable for sampling analog signals that change at a slow rate. For example, the ADC could be used to either monitor the voltage drop across a potentiometer on a user interface panel, or to sample the voltage on a battery monitoring circuit. The ADC is not intended to be used as the source of the main data stream for the DSP.

Figure 1. ADC Block Diagram



The ADC is based on a successive approximation architecture that achieves very low power consumption. A sample and hold feature is employed to help produce evenly spaced samples.

The ADC uses external reference voltages to allow isolation of the conversion process from other system supply voltage planes. Three programmable clock dividers are included to allow flexibility of choice among DSP input clocks.

For detailed information on the ADC, please refer to the ADC chapter of the *TMS320C55X DSP Peripheral Users Guide* (SPRU317).

2 Basic Operations

The ADC peripheral requires two basic operations:

- The first requires setting the ADC sampling frequency.
- The second requires reading the data value.

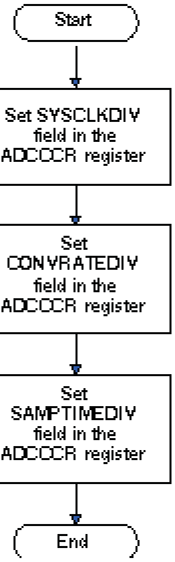
These operations are implemented using the CSL ADC_setFreq() and ADC_read() functions (see Figure 2). Generally, you should call the ADC_setFreq function to configure the sampling rate before calling ADC_read. For a complete description of these functions, please refer to the *TMS320C55X Chip Support Library API Reference Guide* (SPRU433A). CSL source code is provided as part of Code Composer Studio (CCS).

ADC_setFreq(): Figure 2 illustrates a block diagram and API description for the ADC_setFreq function. This function takes three parameters as input and configures the ADC sampling frequency. These parameters include the SYSTEMCLKDIV, the SAMPTIMEDIV, and the CONVRATEDIV. Please refer to Section A.1 for a detailed example on setting the sampling rate.

Initially the function sets the SYSCLKDIV field in the ADCCCR register, which divides down the system clock to obtain the ADC clock. Next, the function sets the CONVRATEDIV field in the ADCCDR register with the value given in order to divide down the ADC clock to generate the conversion clock. Last, it sets the SAMPTIMEDIV field in the ADCCDR register to configure the sample and hold time.

Calling Convention	void ADC_setFreq(int cpuclkdiv, int convratediv, int sampletimediv);

Figure 2. Block Diagram for ADC_setFreq Function



ADC_read(): Figure 3 shows a block diagram of the ADC_read function. This function is used to perform analog to digital conversions and is generally called after setting the sampling frequency with the ADC_setFreq function.

The ADC_read(), as input, points to the location where digital data will be stored, the number of conversions to perform, and the channel number from which to read. Initially, the function sets the desired channel number in the CHSELECT field of the ADCCR register.

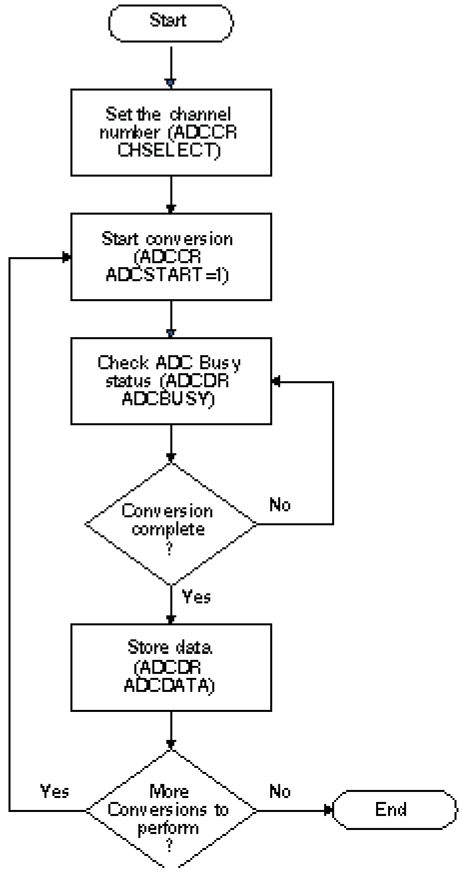
Next, in order to start the conversion, the function sets the ADCSTART bit of the ADCCR register. Upon completion of this step, the function verifies that the conversion has completed by polling the ADC busy bit (ADCBUSY) in the ADCDR register. If the conversion is still being performed (ADCBUSY = 1), it continues to poll until the process is complete. If the conversion is complete, the function reads the digital data from the ADCDATA field in the ADCDR register.

The function loops back to the ADCSTART bit setting and continues to do so until there are no more conversions to perform.

Calling Convention

```
void ADC_read(int channelnumber,
              Uint16 date,
              int length);
```

Figure 3. Block Diagram for ADC_read Function



3 Keypad Read Example

This example allows you to determine which key was pressed on a keypad. Pressing a key generates DC voltage, and the ADC converts the DC voltage into its digital representation. This example can be used on the Spectrum Digital C5509 EVM board.

3.1 ADC Keypad Example

The ADC keypad example code is shown below.

Figure 4. ADC Keypad Read Example Code

```
#include <csl.h>
#include <csl_adc.h>
#include <stdio.h>

Uint16 samplestorage[2]={0,0};
int sysclkdiv=2,convratediv=0,sampletimediv=79;
int counter=0,index=0;

int channel=1,samplenumber=2;
{
main()
CSL_init();

  ADC_setFreq(sysclkdiv,convratediv,sampletimediv);

while (1) {

  ADC_read(channel,samplestorage,samplenumber);

  for (counter=0;counter<samplenumber;counter++) {
  samplestorage[index]=(samplestorage[index]&0x0FFC);
  index++;
  }
  switch (samplestorage[1])

  {
  case 0x0150 :
  printf("You pressed the SW1 button\n");
  break;
  case 0x00F8 :
  printf("You pressed the SW2 button\n");
  break;
  case 0x017C :
  printf("You pressed the SW3 button\n");
  break;
  case 0x00BC :
  printf("You pressed the SW4 button\n");
  break;
  case 0x01B8 :
  printf("You pressed the SW5 button\n");
  break;

  }
}
#endif
}
```

Include CSL header files.

Initialize array to store digital data.

Initialize sampling frequency parameters.

Initialize channel number to read from and the number of conversions to perform.

Initialize the CSL library.

Perform 2 (samplenumber) conversions and store digital data into samplestorage array.

Mask lower 2 bits.

Determine which key was pressed and print to screen.

3.2 Code Description

Figure 4 illustrates the example code used to determine which key was pressed on a keypad. This determination is achieved by using the ADC to convert a DC voltage generated by pressing a key into its digital representation. The keypad can generate a voltage in the range of 0 - 3.3 V, and the ADC has 10 bits of resolution; therefore, there are 1024 possible values, with a range of 0x000 to 0x3FF, that can be generated.

Step 1: Sampling frequency and channel number value selection

Initialize various parameters that are used to perform the conversion. First, initialize the samplestorage array which will store the sampled values:

```
Uint16 samplestorage[2] = {0,0};
```

It is necessary to set up the three parameters that will be passed to the ADC_setFreq function to configure the sampling rate. These settings allow the ADC to have a sampling rate of 21.5 khz, which is the maximum sampling frequency for the ADC.

```
int sysclkdiv = 2,convratediv = 0,sampletimediv =7 9;
```

Next, initialize the two variables that determine the desired channel number to be read from and the number of conversions to perform (2).

```
int channel = 1,samplenumber = 2;
```

Step 2: Setting the Sampling frequency

Using the arguments from Step 1, call the ADC_setFreq function to set the ADC sampling frequency to 21.5 Khz.

```
ADC_setFreq(sysclkdiv,convratediv,sampletimediv);
```

Step 3: Data conversion

Perform conversions using the ADC, call ADC_read with the number of conversions = 2, the samplestorage array, and channel number 0.

```
ADC_read(channel,samplestorage,samplenumber);
```

For various reasons, the least significant bits of the converted values will vary. This includes the type of power source that is used. To work around this issue, mask the lower two bits of the digital value. This will assure consistent results.

```
for (counter=0; counter<samplenumber; counter++) {
samplestorage[index]+(samplestorage[index]&0x0FFC);
index++;
}
```

Step 4: Testing

Given that the range of values generated are from 0x000 to 0x3FF, and the voltage ranges from 0 to 3.3 V, if the voltage that each key generates when pressed is known, it will determine the corresponding digital value that is generated.

For example, a voltage of 3.3 V will generate a digital value of 0x3FF, whereas a voltage of 0.0 V will generate a value of 0x000. The SW1 key on the Spectrum Digital C5509 EVM board produces a value of 1.08 V. The corresponding theoretical digital value can be determined by using the following procedure:

1. Divide the generated voltage (1.08 V) by the maximum voltage (3.3 V).
2. Multiply the result by 1024 (the maximum number of values for this 10-bit ADC).
3. Convert the result to a hexadecimal number.

   Example:
   Step 1: 1.08/3.3 = 0.328125
   Step 2: 0.328125 * 1024 = 336
   Step 3: 336 (decimal) = (0x150 (hexadecimal)

A switch statement is used to determine the exact button pressed by comparing the theoretical and actual values. The determined value is printed to the screen.

```
switch (samplestorage[1])
   {
   case 0x0150 :
   printf("You pressed the SW1 button\n");
   break;
   case 0x00F8 :
   printf("You presed the SW2 button\n");
   break;
   case 0x017C :
   printf("You pressed the SW3 button\n");
   break;
   case 0x00BC :
   printf("You pressed the SW4 button");
   break;
   case 0x01B8 :
   printf("You pressed the SW5 button");
   break;
   }
```

Appendix A ADC Conversion Example

A.1 Conversion Example Instructions

The clock dividers are used to derive the total conversion time from the system clock within the DSP. The following example illustrates, through the programming of these clock dividers, how to obtain the maximum sampling frequency in a system where the DSP operates at 144 MHz. Once the ADC sampling rate has been programmed, the DSP begins using the ADC for sampling analog inputs.

1. Divide down the system clock to generate the main clock to the ADC (ADC clock). In order to minimize the power consumption of the ADC, it is desirable to program the ADC clock to lowest possible frequency.
   Program the ADC clock to 4 MHz. To obtain the 4 MHz value, the system clock of 144 MHz must be divided by 36. An 8-bit divider, SystemClkDiv bits in ADCCR, is provided.

   ADC Clock = (System Clock) / (SystemClkDiv + 1)

   ADC Clock = (144 MHz) / (SystemClkDiv + 1)

   ADC Clock = (144 MHz) / (35 + 1) = 4 MHz

Figure 5. ADC Clock Control Register (ADCCR)

15–9	8	7–0
Reserved	IDLEEN	SystemClkDiv = 0010 0011

The 4-MHz ADC clock is now divided to generate the two components of the total conversion time.

2. Divide down the 4-MHz ADC clock to generate the conversion clock.
   Program the conversion clock rate divider to generate the maximum possible conversion clock frequency of 2 MHz. To obtain the 2-MHz conversion clock frequency, the ADC clock must be divided by the lowest value. A 5-bit divider, ConvRateDiv bits in ADCDR, is provided. *ADC Conversion Time = 13 x (1 / (2 MHz)) = 6.5 µs*

   ADC Conversion Clock = (ADC Clock) / (2 x (ConvRateDiv + 1))

   ADC Conversion Clock = (4 MHz) / (2 x (ConvRateDiv + 1))

   ADC Conversion Clock = (4 MHz) / (2 x (0 + 1)) = 2 MHz

   ADC Conversion Time = 13 x (1 / ADC Conversion Clock)

   The actual conversion time of this clock is 13 cycles; therefore, the conversion time is 6.5 ms.

Figure 6. ADC Clock Divider Register (ADCDR)

15–8	7–4	3–0
SampTimeDiv	Reserved	ConvRateDiv = 0000

3. Program the clock divider for the sample and hold time. The sample and hold time must be greater than or equal to 40 µs.
   Program the sample and hold time to 40 ms. An 8-bit divider, SampTimeDiv bits in ADCDR, is used in conjunction with the conversion rate divider to obtain the sample and hold time from the ADC clock.

   ADC Sample and Hold Period =

   (1 / (ADC Clock)) / (2 x (ConvRateDiv + 1 + SampTimeDiv))

   = (1 / (4 MHz)) / (2 x (0 + 1 + SampTimeDiv))

   = 250 ns x (2 x (0 + 1 + 79)) = 40 µs

Figure 7. ADC Clock Divider Register (ADCDR)

15–8	7–4	3–0
SampTimeDiv = 0100 1111	Reserved	ConvRateDiv = 0000

4. The final results of this example are summarized (see Figure 8). The total conversion time is composed of a 40-µs sample and hold time plus a 6.5-µms conversion time. A new conversion can begin every 46.5 µs, giving a maximum sampling rate of 21.5 KHz.

Total Conversion Time = (Sample and Hold Period) + (Conversion Period)

Total Conversion Time = 40 µs + 6.5 ms = 46.5 µs

Sampling Frequency = 1 / (Total Conversion Time)

Sampling Frequency = 1 / 46.5 µs = 21.5 KHz

Figure 8. Summary of Total Conversion Time Example