AN1711
APPLICATION NOTE

SOFTWARE TECHNIQUES
FOR COMPENSATING ST7 ADC ERRORS

INTRODUCTION

The purpose of this document is to explain in detail some software techniques which you can apply to compensate and minimise ADC errors. The document also gives some general tips on writing software for the ADC. For a list of related application notes that contain other useful information about ADCs see section 6 on page 39.

This document provides some methods of calibrating the ADC. Some ADC errors like Offset and Gain errors can be cancelled using these simple software techniques. Other errors like Differential Linearity Error and Integral Linearity Error are associated with the ADC design and cannot be compensated easily.

The example software provided with this application note is explained in brief in section 5 on page 35.

Rev. 1.0

Table of Contents

1 GENERAL SOFTWARE CONSIDERATIONS 4

1.1 CHECKING FOR FADC (MAX) SUPPORTED BY THE DEVICE 4

1.2 SELECTING CONVERSION CHANNEL (AND STARTING CONVERSION) . . . 5

1.3 POLLING FOR END OF CONVERSION 5

1.4 ADC CONVERSION RESULT FORMAT 5

1.5 READING THE ADC CONVERSION RESULTS 6

1.6 ENTERING HALT MODE 6

1.7 USING A TIMER TO MAKE PERIODIC CONVERSIONS 6

1.8 USING A 10-BIT ADC AS AN 8-BIT ADC 7

1.9 COMBINED REGISTER FOR CONTROL BITS AND LSB OF CONVERSION RESULT 7

1.10ZOOMING TO LOW VOLTAGE SIGNALS 7

2 SOFTWARE TECHNIQUES 8

2.1 AVERAGING TECHNIQUE 8

2.2 AVERAGING BY QUEUE 11

2.3 HISTOGRAM TECHNIQUE 13

2.4 NOISE FILTERING ALGORITHM 17

3 REDUCING SYSTEM NOISE 19

3.1 INSERT “NOP” WHILE CHECKING FOR EOC 19

3.2 USING SLOW MODE 20

3.3 USING WAIT MODE 20

3.4 EXECUTING CODE FROM RAM 22

4 CALIBRATING THE ADC 24

4.1 CALIBRATION ISSUES 24

4.2 CALIBRATION METHODS 24

4.2.1 Use accurate voltage reference 24

4.2.2 Use of external DAC 27
4.2.3 Maintaining a Lookup table 27
4.2.4 Linear compensation 28
4.2.5 Zone compensation 29
4.2.6 Autocalibration for Offset and Gain errors 30
4.2.7 Calibration for Errors using 2 different zones 33

5 SOFTWARE 35

5.1 FILE PACKAGE 35

5.1.1 ADC_tech.h 36
5.1.2 ADC_tech.c 36
5.1.3 Main.c 36

5.2 DEPENDENCIES 36

5.3 GLOBAL VARIABLES 36

5.4 INTERRUPTS 37

5.5 CODE SIZE AND EXECUTION TIME 37

6 RELATED DOCUMENTS 39

1 GENERAL SOFTWARE CONSIDERATIONS

This section gives some basic guidelines for programming the ADC.

General Procedure

- Check for f_{ADC} (max) supported by the device
- Select the conversion channel (and start conversion)
- Poll for End of Conversion
- Read the ADC conversion results

Special Procedures

- Entering HALT mode
- Using a timer with the ADC to perform periodic conversions

Other Special Features

Not all ST7 ADCs have the same features, refer to the datasheet of the ST7 product you are using for specific information. Depending on the device, you may need to apply these tips in your ADC software:

- Using 10-bit ADC as 8-bit ADC
- Handling Control Bits located in same register as Data LSBs
- Zooming to low voltage signals with embedded amplifier

1.1 CHECKING FOR F_{ADC} (MAX) SUPPORTED BY THE DEVICE

Before configuring the ADC and starting any conversions, you need to check the f_{ADC} maximum supported by the device. This value is documented in the ST7 datasheets, you should refer to the ADC electrical characteristics section.

For example:- Some devices have a SPEED bit for working at $f_{CPU}/2$ but the f_{ADC} (max) is 2 MHz. For ST7 devices, f_{CPU} can be up to 8 MHz. In this case you cannot utilize the SPEED bit, because it will boost f_{ADC} to 4 MHz, which is greater than the allowed maximum (f_{ADC} (max) =2 MHz).

If f_{CPU} is 4 MHz or lower (in Run or Slow mode), you can use the SPEED bit to run the ADC at $f_{CPU}/2$, and still respect the 2MHz. f_{ADC} (max).

Some devices support f_{ADC} (max) = 4MHz. It is thus necessary to check the electrical characteristics before configuring the ADC.

1.2 SELECTING CONVERSION CHANNEL (AND STARTING CONVERSION)

The ADC control register provides control bits for selecting the conversion channel. Whenever you change the channel or write in the control register, the ADC conversion starts again (if the ADC is already enabled). The voltage is sampled from the selected channel.

There is no stabilization time required by the ADC after changing the conversion channel and starting conversion. Please refer to the datasheets.

1.3 POLLING FOR END OF CONVERSION

The ADC status register has an EOC bit which is for notifying the end of conversion. In some ST7 devices this bit is named COCO for “conversion complete”.

Once the ADC is enabled, the conversion is started in continuous mode (except in ADCs with single-conversion feature). When you check and find that the EOC bit is set, the data is available in the data registers (see next section).

1.4 ADC CONVERSION RESULT FORMAT

The conversion result of the ADC is available in the ADC data registers. In devices with an 8-bit ADC, an 8-bit register generally called ADCDR, is available for reading the conversion result.

In a 10-bit ADC, 2 registers are available for reading the 10-bit result. The most significant 8 bits are available in a register called ADCDRH and the 2 least significant bits are available in the other register generally named ADCDRL.

This requires reading the ADCDRL and then ADCDRH. The 10-bit result is obtained by left-shifting ADCDRH by 2 bits and then ‘OR’ing the value of the 2 bits ADCDRL, read previously into a variable. All ST7 10-bit ADCs use the same format, which makes it easy to port software from one device to another. Please take care that the 2-bits of ADCDRH are not lost when left-shifting the register by 2 bits. Please refer to the datasheet for the conversion result format.

1.5 READING THE ADC CONVERSION RESULTS

It is recommended to read the ADCDRL first and then the ADCDRH. When the ADC is in continuous mode, the EOC is set at the end of conversion and a new conversion is started again (unless the ADC supports one-shot conversion).

The ADC conversion results are not latched on ADCDRL and ADCDRH. This means that if, between reading ADCDRL and ADCDRH, there is an interrupt which takes a lot of time (more than the ADC conversion time), then the software will read the ADCDRL from one conversion and the ADCDRH from another conversion. It is thus recommended to disable interrupts before reading the conversion results from ADCDRL and ADCDRH and then enable interrupts again.

However, if you are reading the ADC registers (ADCDRL and ADCDRH) in a peripheral interrupt subroutine, for example, if you are reading the registers in a timer interrupt or external interrupt sub-routine then, do not disable and enable the interrupts.

The “Enable Interrupt” instruction in an interrupt subroutine (in concurrent interrupt mode) will enable interrupts and cause a nested mode interrupt.

1.6 ENTERING HALT MODE

It is always recommended to shut down the ADC before entering the HALT mode. When exiting from HALT mode, put the ADC ON again. The stabilization time for the ADC, after exiting from HALT is specified in the datasheet.

1.7 USING A TIMER TO MAKE PERIODIC CONVERSIONS

Some applications may have special requirements for ADC conversion. For example, in an audio application, you may need to sample an audio signal of maximum 3 kHz. You can choose to sample 6K samples per second or higher (12K samples/s or 24K samples/s). The ST7 ADCs do not have a feature for doing this.

In this case it is recommended to use the timer and configure it to generate 6K interrupts (or 12K/ 24K depending on the design) per second. In your timer interrupt routine, the ADC conversion results can be read and stored.

This kind of configuration is easy to use because of the very fast conversion time of ST7 ADCs and the built-in continuous conversion feature.

1.8 USING A 10-BIT ADC AS AN 8-BIT ADC

The EOC bit is cleared only when you read the ADCDRH. You can use the ADC in 8-bit mode if you do not need 10-bit ADC resolution. Thus there is no need to read the ADCDRL register.

Here is the software flow:

1. Check for EOC
2. If EOC is set, read the ADCDRH. This clears the EOC bit.
3. Do not read the ADCDRL. It is not mandatory to read this register.

1.9 COMBINED REGISTER FOR CONTROL BITS AND LSB OF CONVERSION RESULT

Some ST7 devices have a single register containing both the 2 least significant bits of the conversion result and by some control bits in the rest of the register. In this case you need to mask the control bits to filter the 2 bits of the ADC conversion result.

1.10 ZOOMING TO LOW VOLTAGE SIGNALS

Some the ST7 devices (for example ST7LITE) have a built-in amplifier to amplify the input signal. A control bit is available to switch the amplifier ON.

It is thus possible to zoom for lower voltages by switching the ADC amplifier ON and then switching it OFF for higher voltages.

This is very useful for interfacing sensors directly connected to the ADC inputs.

2 SOFTWARE TECHNIQUES

2.1 AVERAGING TECHNIQUE

Averaging is a simple technique where you sample an analog input several times and take the average of the results. This technique is helpful in eliminating the effect of noise on the analog input or wrong conversion.

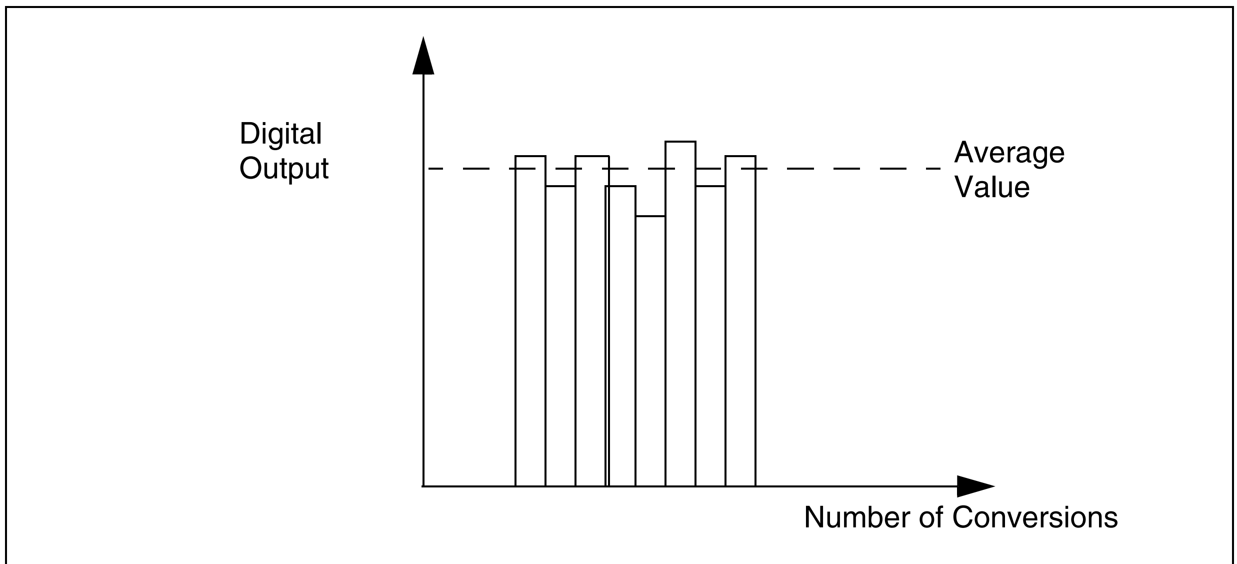
As we take the average of several readings, these readings must correspond to the same analog input voltage. You should take care that the analog input remains at the same voltage during the time period when the conversions are done. Otherwise you will add digital values corresponding to different analog inputs and introduce errors.

In other words the analog input should not change in-between the different readings considered for the averaging.

It is better to collect the samples in multiples of 2. This makes it more efficient to compute the average because you can do the division by right-shifting the sum of the converted values. This saves CPU time and code memory needed to execute a division algorithm.

For example take 8,16, 32 samples etc., and then take the average.

Figure 1. Graphical representation of Averaging technique

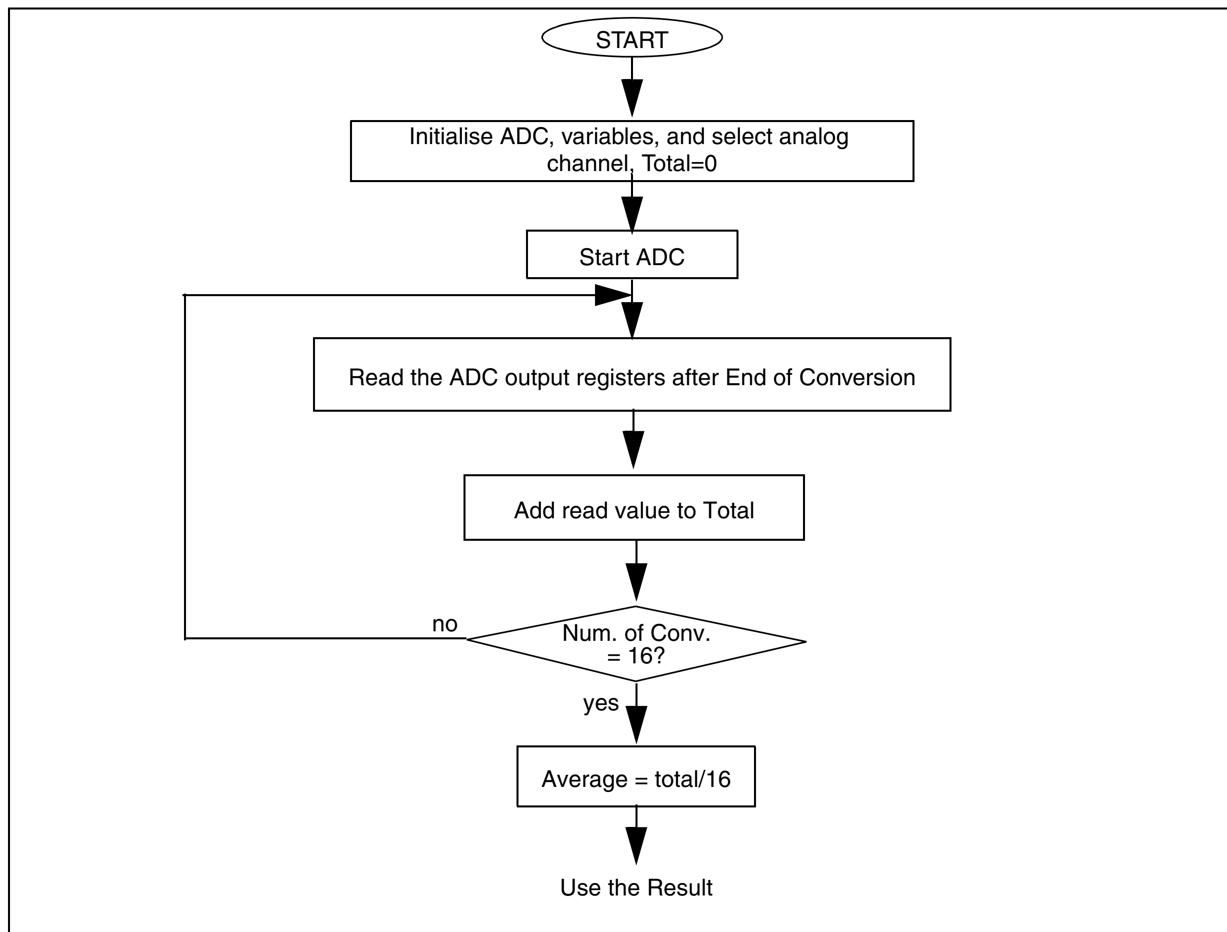


Practical measurement

To obtain the results, this averaging technique is used to measure the voltage on one of the microcontroller's analog input pins. A total of 16 conversions is taken and the average is calculated. This is done in a loop in the firmware.

A switch connected to a port pin can be used to inform the software to send the data to a host PC for display via the SCI communications interface. The port pin used for the switch must be configured as input. The firmware checks if the switch is pressed (0 is read if switch is pressed and 1 if open), the ADC readings are then sent to the PC using RS232 communication. You can use the HyperTerminal application to display the results.

Figure 2. Averaging Algorithm

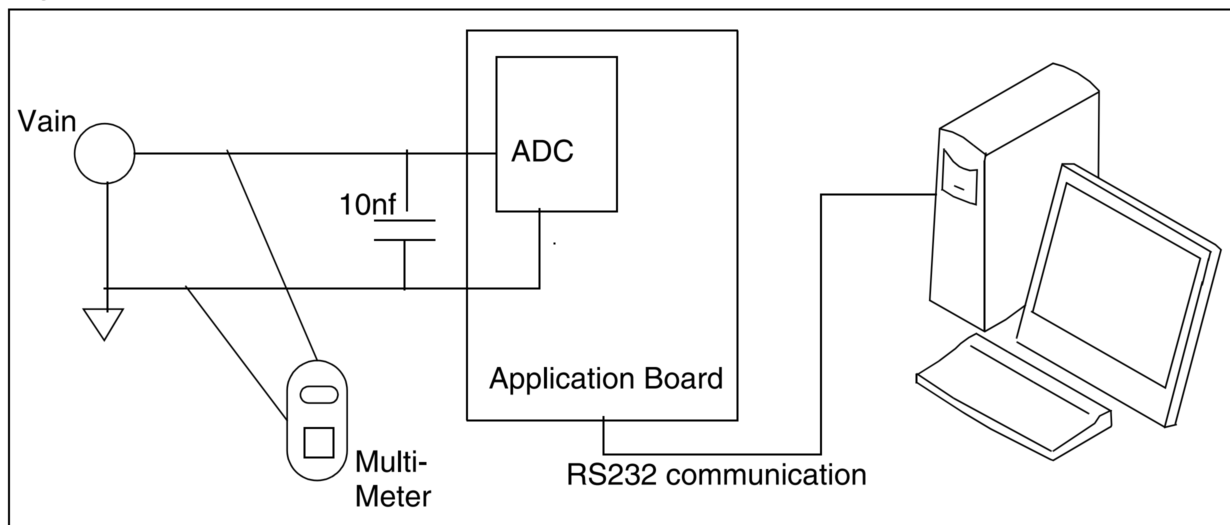


Total conversion time = (number of samples*ADC conversion time)+ computation time.

Computation time = time taken to read the results, add them together and calculate the average by dividing the total by the number of samples.

There is a tradeoff between the total conversion time and number of samples used for averaging, depending on the analog signal variations and time available for computation.

Figure 3. Hardware Setup



The following results are obtained:

V_{AREF}, V_{DD}= 4.950 V

Table 1. Averaging Results

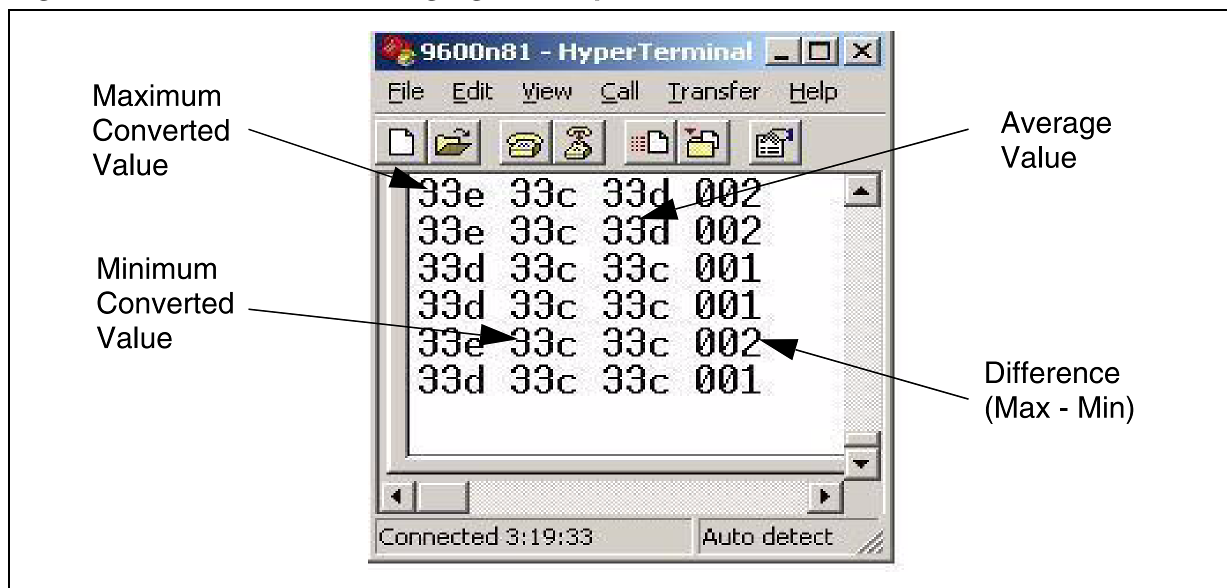
Vin	Ideal results	Maximum value obtained	Minimum value obtained	Average
0.5 V	103	103	101	102
1.0 V	206	206	205	205
1.5 V	310	310	309	309
2 V	413	415	414	414
2.5 V	516	516	516	516
3 V	620	621	620	620
3.5 V	723	725	724	724
4 V	826	831	829	830
4.5 V	930	935	934	934
4.95	1023	1023	1023	1023

Tips:

1. It is always better to take the average of 16 samples rather than to take only one conversion result. If you take a single conversion it can be erroneous because of noise.
2. It is recommended to always keep the analog path as short as possible between the source of the analog voltage and the ADC inputs.

3. Even when you connect a multimeter to the analog input signal, it may introduce noise if the probes of the multimeter are not shielded. Hence, they act like antennae. The results vary by several (6 to 7) LSBs. The average value is always near to the center of the variations.

Figure 4. Illustration of Averaging technique



Comments

1. Averaging is the most popular technique because it requires only a little extra RAM space.
2. The disadvantage is, values affected by noise and the least occurring values (outliers) affect the average.

2.2 AVERAGING BY QUEUE

This technique is Averaging of a LIFO queue (Last-in, first-out). The queue is maintained by using an array to fill the ADC results. To do this:

– Maintain a variable which contains an index for array.

– After every new conversion, overwrite the converted value at the index and then take the average and increment the index.

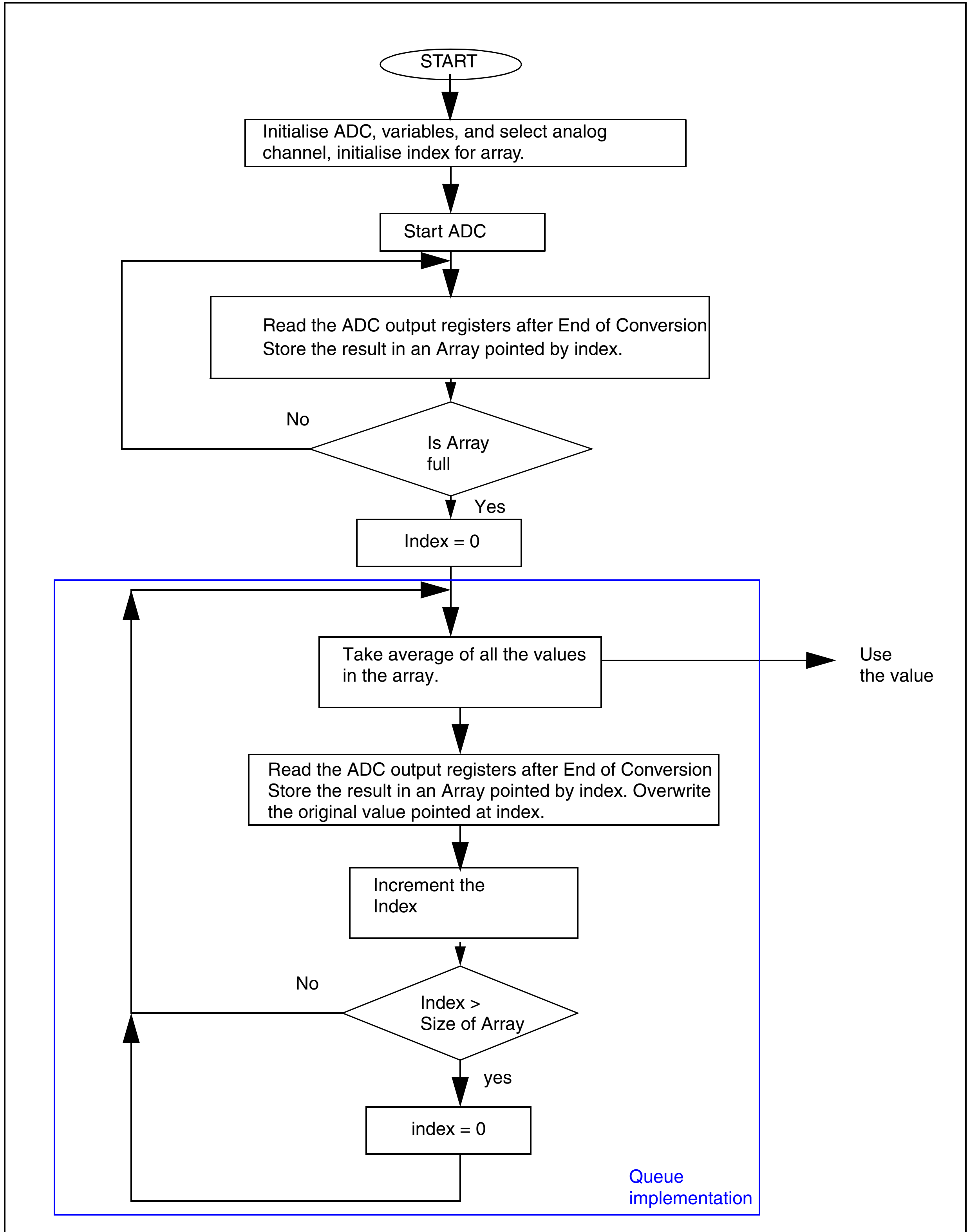
– Once the index reaches the end of array, reset the index to the start of array.

Thus, for every conversion you fill the queue and overwrite the old values with the new values.

This technique is useful when the application cannot wait for the required number of conversions because the conversion time is too long. The time required to calculate the average should be less than the time required to make the total number of conversions.

This technique can be used for slowly varying ADC signals. for example, battery monitoring.

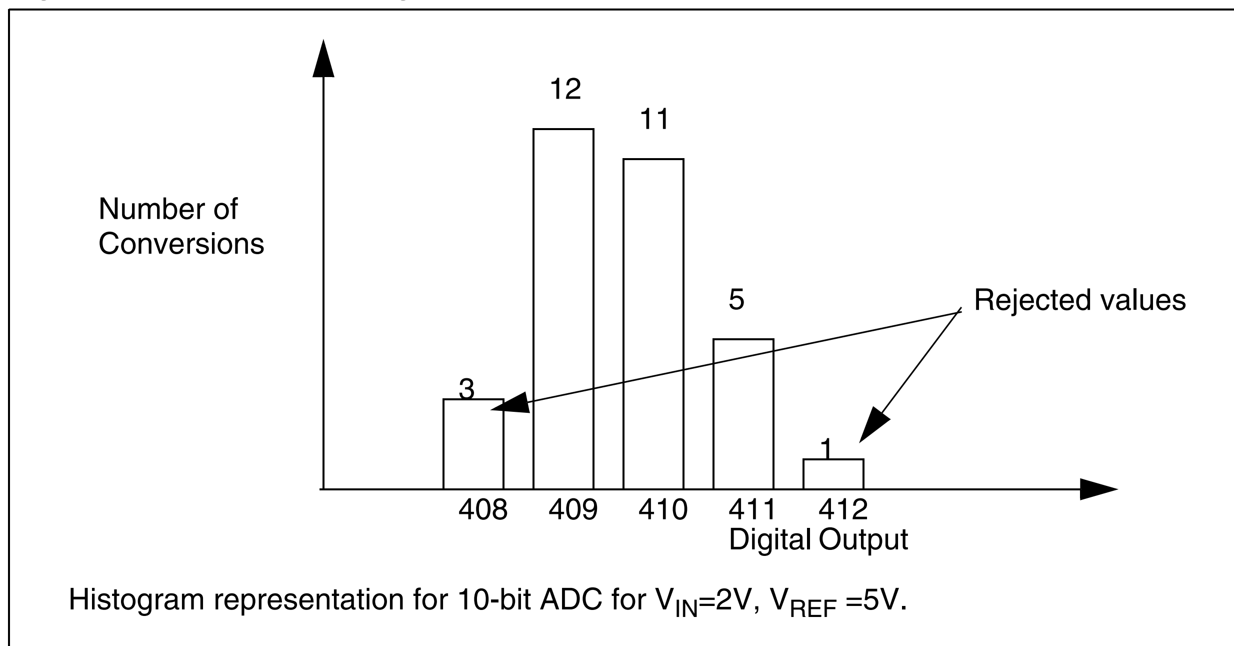
Figure 5. LIFO implementation of averaging



2.3 HISTOGRAM TECHNIQUE

A histogram arranges several conversion results in increasing order and calculates the number of occurrences of each value. The technique is to discard the least occurring values. We assume that the least occurring values are the effect of a momentary disturbance in the application.

Figure 6. Overview of Histogram technique



Histogram representation for 10-bit ADC for V_{IN}=2V, V_{REF} =5V.

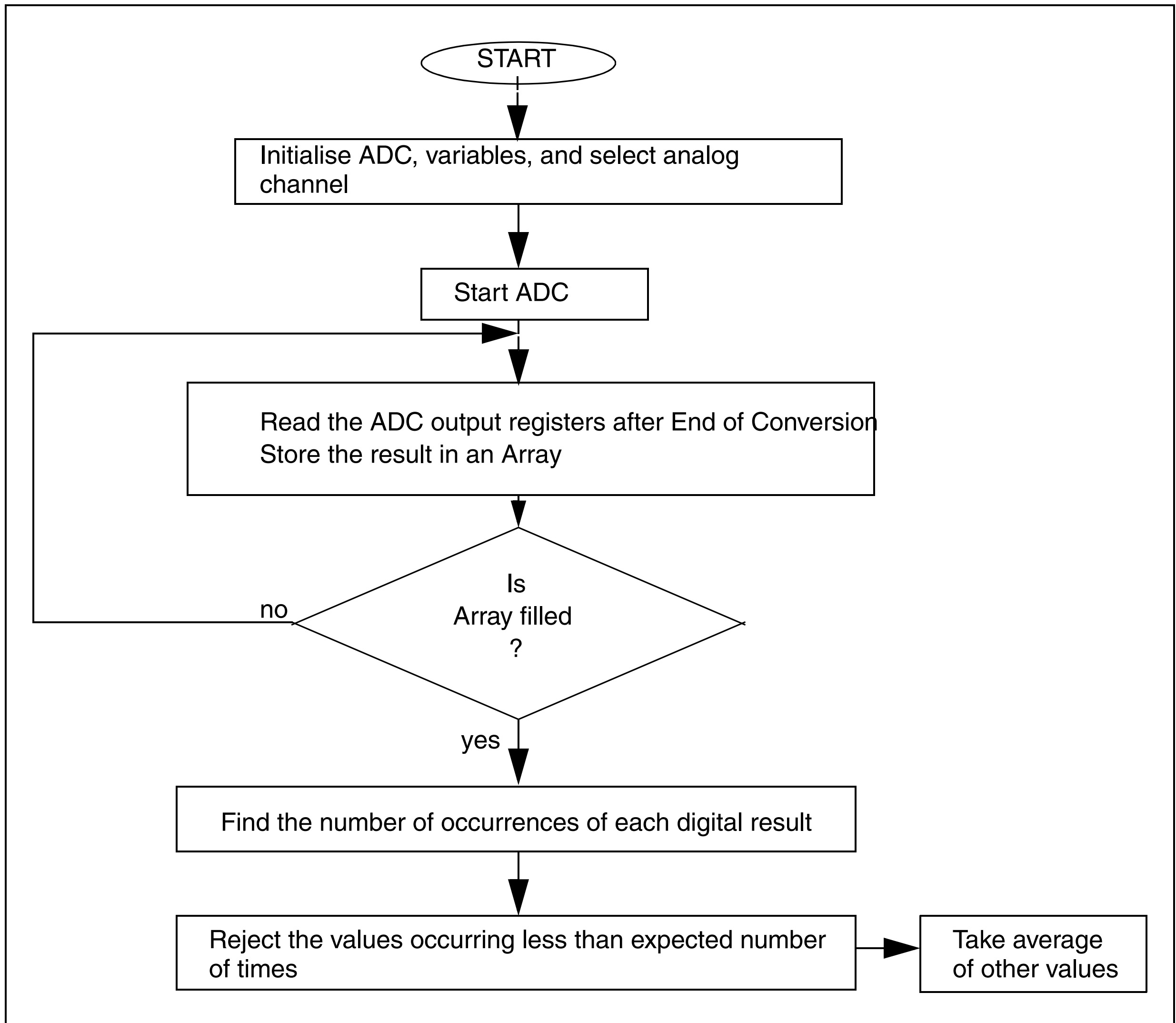
Example

Figure 6 shows an example histogram. It gives the number of occurrences of a conversion value for V_{IN}=2V, V_{AREF} =5V. The histogram shows 5 different values received after 32 conversions. Each output value is arranged in a bar chart showing the number of occurrences. (for example value=409 received 12 times out of 32 conversions).

The principle is to reject the values with the smallest number of occurrences, assuming them to be result of noise etc.

This technique has limitations in practical usage. The disadvantage is more RAM requirement because all the results are stored in an array. It also requires lot of time for post processing the data. The processing of data involves finding the total number of occurrences of any value and then discarding the data which occurred less than a specified number of time.

Figure 7. Histogram technique Flowchart



Practical measurement

You can test this technique using a hardware setup similar to the one shown in Figure 8. The eliminated value was also captured on HyperTerminal. Only last eliminated value was displayed on HyperTerminal.

A Switch connected at a port pin was used to inform the software to send the data on SCI communication. The port pin was configured as input and when, the switch was pressed ‘0’ is read on this pin, the ADC readings were then sent to the PC using RS232 communication. These readings were read on the HyperTerminal and noted.

A function generator is used in lab to introduce noise in the neighbouring pin of analog signal being converted so as to generate the noisy conversions. This neighbouring pin is also configured as floating input.

Figure 8. Hardware setup

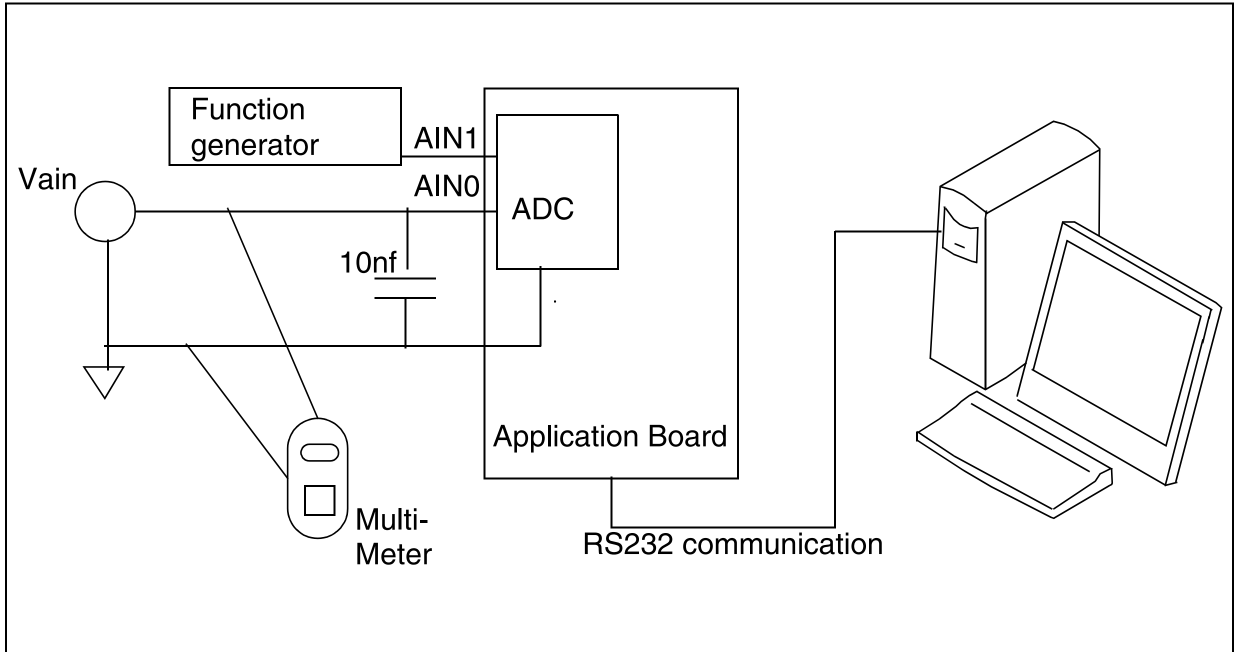
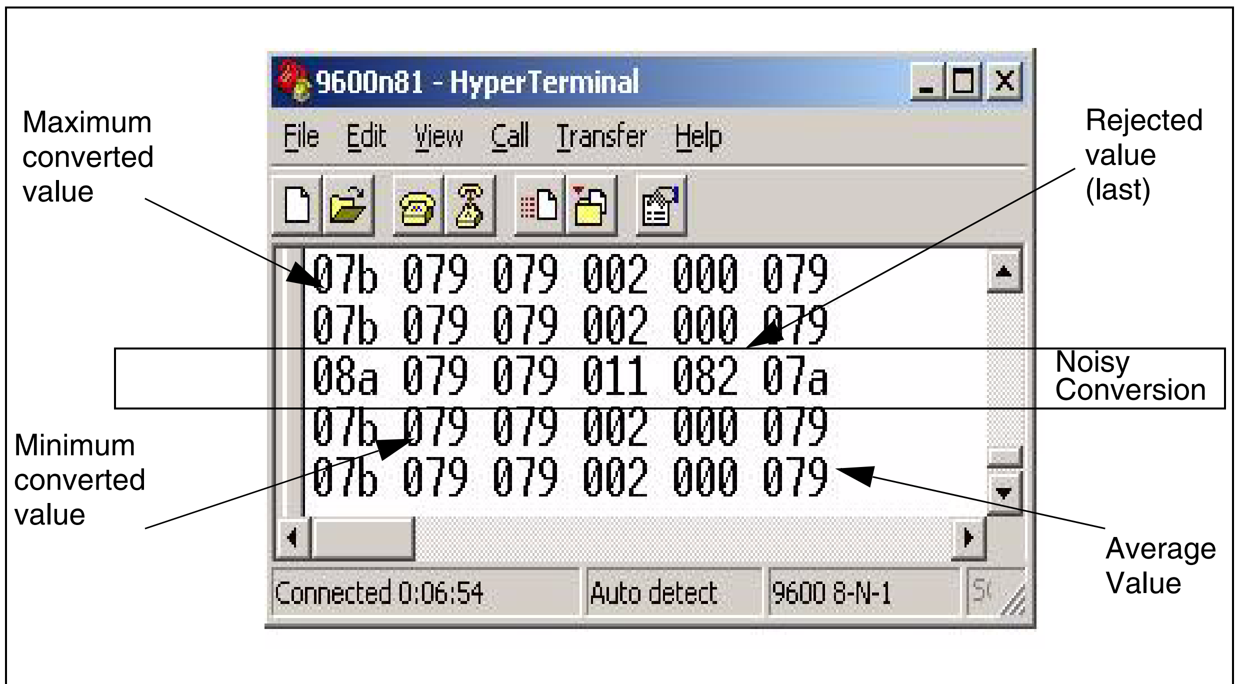


Figure 9 shows how the data is displayed. A noisy conversion is shown and it can be noted that the result of the histogram technique is better than the averaging technique.

Figure 9. Illustration of histogram technique



Comments

1. The Histogram technique requires lot of RAM for storing the data.
2. It takes a lot of time to process the data to find the outlier conversion results caused by noise.
3. This technique is mostly used to analyze ADC performance and not in the final application because of the above disadvantages.
4. If the outlier results caused by noise are symmetrically placed around the expected result then the averaging technique can be as good as the histogram technique.
5. If the results received do not vary much, then it is better to use averaging.

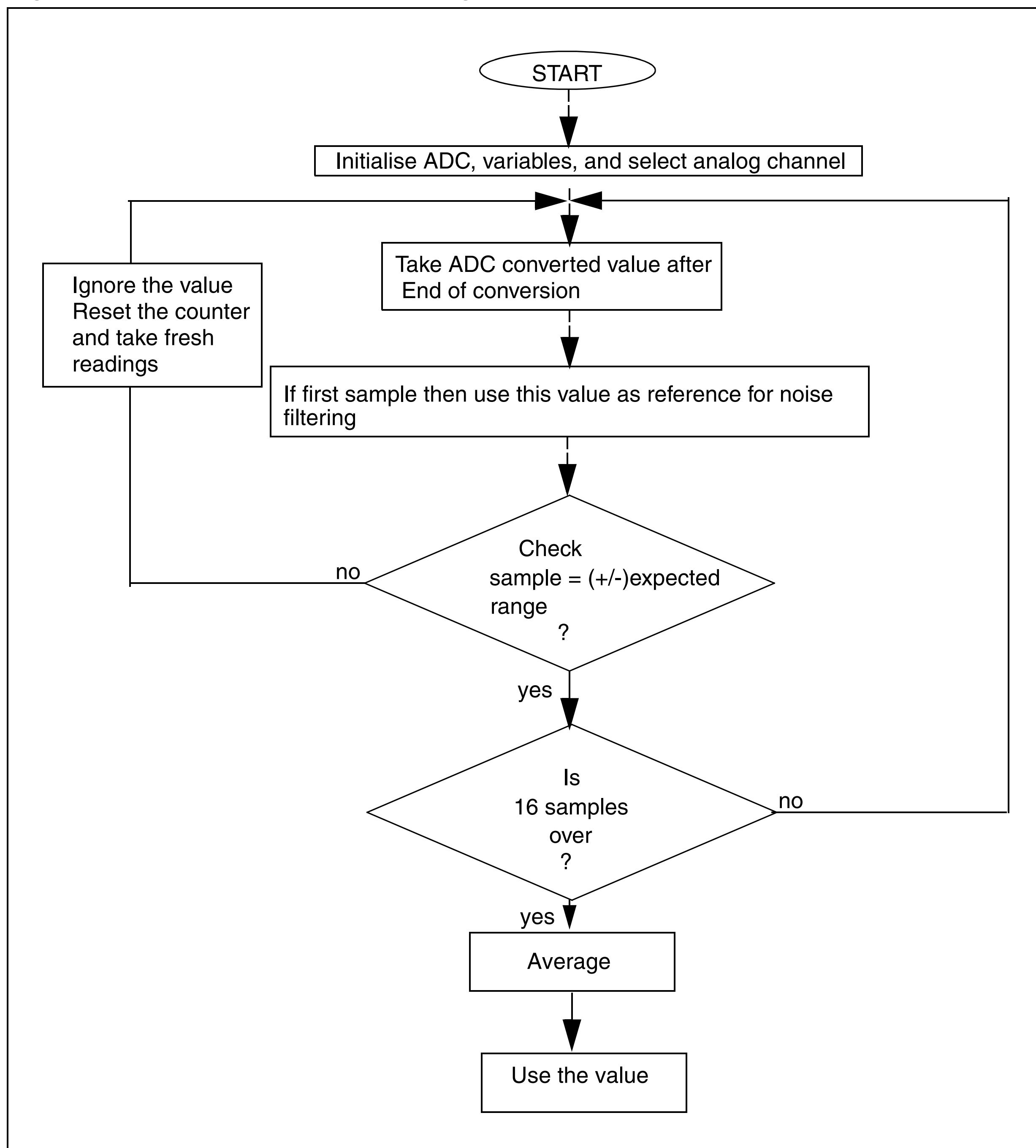
Algorithm improvement

The histogram technique algorithm can be further improved by not storing the converted values. Instead, an array can hold the number of occurrences of each result as an offset from the first digital value. If the first conversion is disturbed by noise, then the array size limitation will be a problem.

2.4 NOISE FILTERING ALGORITHM

During ADC conversions it is possible that a small number of results are received which are totally different from the majority of values. These are called outliers. These may be the result of induced noise in the system. Outliers should be filtered so that the average result is not affected by them. The flowchart shows how this algorithm works.

Figure 10. Flow chart for noise filtering



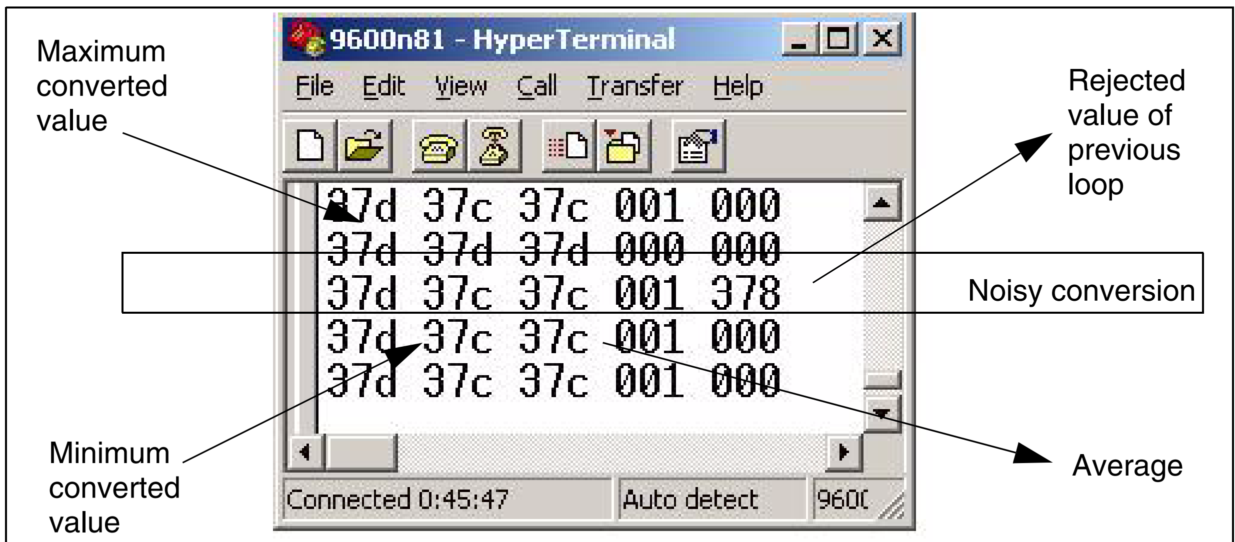
Practical measurements

You can test how this works in practice using a hardware setup like that shown in Figure 8. The objective is to eliminate the errors resulting from noise in the analog input pin of the microcontroller. The loop takes 16 conversions and finds if the values received are within the range. If the values are out of range, a new conversion cycle starts, ignoring the results received and resetting all the variables.

A switch connected to port pin can be used to trigger the software to send the conversion results to the PC using RS232 communication. The results can be displayed on the PC screen using the HyperTerminal as shown in Figure 11.

You can use a function generator to introduce noise in the pin next to the analog signal being converted and to generate conversion errors.

Figure 11. Illustration of noise filtering



Comments

1. The code eliminates the outlier values even if the first conversion in the loop is noisy.
2. Note that the conversion loop is restarted when a noisy value is received, so if the environment is very noisy then it is very important to define the proper range for filtering the results.
3. In noisy environments, the conversion loop could take too much time, making endless conversions and rejecting values. You can avoid this by putting a timeout variable in your program.
4. This technique filters the same way as the histogram technique but it is more efficient in terms of RAM usage and execution.
5. This technique is more accurate than averaging because it takes care of outlier results, however it uses more RAM and takes more execution time for the same number of conversions.

3 REDUCING SYSTEM NOISE

ADC conversion results are the ratio of the input voltage to the reference voltage. If there is noise in the reference voltage, then the results may be not be accurate. Both hardware and software design are responsible for reducing the noise.

The execution of code generates some non-negligible noise on the internal power supply network of the microcontroller. To filter this noise, the V_{DDA} (or V_{AREF}) and V_{SSA} analog supply pins are available on the microcontroller package so you can connect a capacitor filter to these power supply pins to filter high frequency noise. Sometimes these power supply pins may not be available on the package with low pin count.

In any case, you can reduce the generation of internal noise by applying some software techniques and making use of the microcontroller's power saving modes.

Here are some general software design tips for reducing system noise:

– Do not start transmission on any communication peripheral just before starting the ADC conversion. The toggling of I/Os may create some noise in the supply voltage.

– Do not toggle high-sink I/Os connected to relay coils etc., which cause noise ripples in the power supply.

– Use power saving modes like “Slow” mode and “Wait for Interrupt” mode.

3.1 INSERT “NOP” WHILE CHECKING FOR EOC

Do not execute a lot of instructions while ADC conversion is in progress. Execution of instructions may generate some noise in the power supply. Try to insert the NOP instruction when polling the EOC bit. This reduces the number of “jump” instructions required but it uses up some extra program memory to store the NOP op-codes.

In the software example, we used 4 NOP instructions before checking the EOC. It can be calculated from the execution time (NOP=2 cycles, BTJF = 5 Cycles) that once the ADC is put ON and EOC is set, the execution of the code takes a minimum of 4 cycles to execute the BTJF instruction with 4 NOP instructions.

Code in “C”	**Corresponding assembled code in assembly**
ADC_SET_START; do { asm Nop; asm Nop; asm Nop; }while (!(isADC_EOC_SET));	BSET ADCCSR,#5 NOP ; 2 cyc NOP ; 2 cyc NOP ; 2 cyc BTJF ADCCSR,#7,*-3 ; 5 cyc

Similarly if 1 NOP is used, it will take 9 executions of the loop before the routine exits.

If 2 NOP instructions are used, it will take 7 executions of the loop

If 3 NOP instructions are used it will take 5 executions of the loop

If 4 NOP instructions are used it will take 4 executions of the loop

See: ADC_do_conversion()

3.2 USING SLOW MODE

You can use “Slow mode” to reduce the internal noise generated by the CPU. SLOW mode reduces the CPU frequency f_{CPU}. This reduces the internal noise because the CPU runs slower and executes the code at a lower frequency.

For devices with f_{CPU}(max)=8 MHz and f_{ADC}(max) =2 MHz, you can utilise the SPEED bit available in some ADCs. Refer to section 1.1 on page 4. If you set the SPEED bit, you can use $f_{ADC} = f_{CPU}/2$. So in this configuration you can reduce the f_{CPU} to 4 MHz by setting the SLOW mode and you can set the SPEED bit to make f_{ADC}(max) =2 MHz.

In the software example: the following sequence is used

1. MCCSR register, keep CP[1:0] = 00. $f_{CPU} = f_{OSC2}/2$

2. MCCSR register, Set SMS (Slow mode select) = 1

This configures f_{CPU} = 4 MHz and f_{ADC} = 1 MHz

3. Set the SPEED bit in the ADCCSR register

This configures f_{ADC} = 2 MHz

Use #define ADC_SLOW_MODE_SELECT in adc_tech.h to enable the ADC conversions in SLOW mode.

3.3 USING WAIT MODE

You can also use WAIT mode to reduce the internal noise generated by the CPU. In WAIT mode the CPU is OFF and the peripherals are working. Note that some ADCs do not have interrupt capability, but anyhow you can wake up the CPU from WAIT mode by means of a timer interrupt or an external interrupt.

In the software example, we use the Timer-A output compare interrupt to wake up the CPU from WAIT mode.

The value to be compared is chosen in order to take enough time to allow one ADC conversion to be completed. We can also use a time period which is equivalent to 2 conversions. This would guarantee that at least the second conversion is done while the CPU is in WAIT mode.

Programming Tips

1. If you use the timer interrupt to exit from WAIT mode, take care that the time period should not be defined in such a way that half the conversion is done in WAIT mode and rest when the CPU is running. Otherwise you will not get the benefit of a conversion done in WAIT mode.
2. You need to consider the execution done prior to entering WAIT mode. For example, the ADC conversion is restarted and the WFI instruction is executed. This takes 7 cycles which is approximately 1µs when f_{CPU}=8 MHz.
   So, the time programmed in output compare register must be equal to 1us+ADC conversion time for one ADC conversion.
   For 2 ADC conversions, the time will be 1µs+2*ADC conversion time.
3. Please note that the timer reset value is 0xFFFC. This means that it will take 4 timer cycles to overflow and reset to 0000.
4. On some ST7 devices, the ADC has an End of Conversion interrupt capability, in this case you can use the ADC interrupt directly to wake up from WAIT mode.

Figure 12. Illustration of Wait mode

First column is Maximum value recorded

Second column is minimum value recorded

Average Result

Difference (Max - Min)

```
96n81c2 - HyperTerminal
File Edit View Call Transfer Help
W11b 119 11a 002
A11b 119 119 002 002 000
W11a 119 119 001
A11b 119 119 002 002 000
W11a 119 119 001
A11a 118 118 002 002 000
W11b 11a 11a 001
A11a 119 119 001 001 000
W11b 119 119 002
A11a 119 119 001 001 000
W11
Connected 3:09:11 Auto detect 9600 8-N-1
```

**Rows starting with 'W' are results from WAIT mode execution
**Rows starting with 'A' are results from execution from flash.

In the software example, we take the time to do two ADC conversions while the CPU is in WAIT mode. For ST72F324, this is around 16µs. (= 1µs+ 2*7.5µs). So, the timer compare

value is taken equivalent to 18µs which is 2µs greater than the required time. This is chosen precisely so that the 3rd conversion is not finished before we read the ADC data registers.

With a timer frequency programmed as 1MHz, the time for comparison is taken as 14. (4 values for 0xFFFC to 0x0000)

Comment

The results obtained from executing the ADC conversion in WAIT mode are more accurate. This can be seen from the difference between the maximum value and minimum value recorded for an analog input (see Figure 12). However these results can still be affected by external noise.

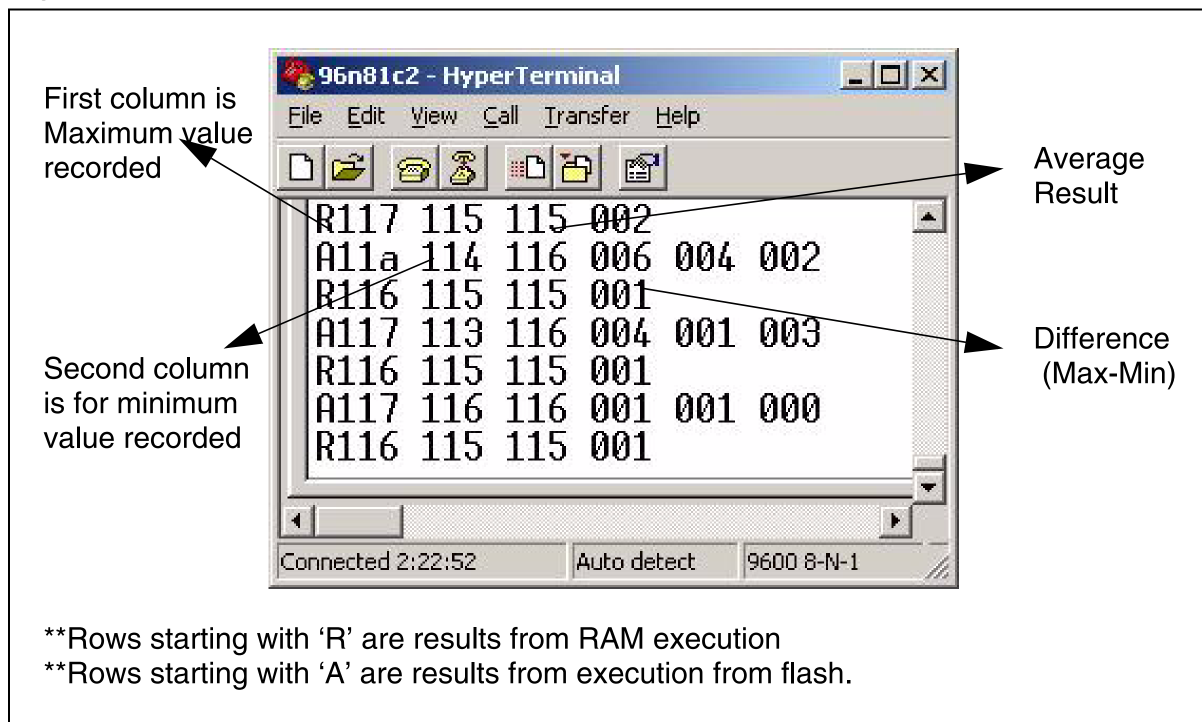
3.4 EXECUTING CODE FROM RAM

The technique of executing the code from RAM can also be used to reduce noise. This is because the CPU does not access the Flash memory to fetch the op-codes but accesses RAM instead.

The disadvantage of this technique is that you need spare RAM to act as program memory. You have to load the ADC conversion function into RAM from the Flash or ROM program memory at some known free RAM locations and then use the in-line assembly code to execute this function.

In the software example provided, we have used the STACK top memory to execute the ADC conversion. The ADC conversion function is loaded in the STACK area at address 100h. The size of the function is 8Bytes. We used these locations because stack-pointer never reaches these locations in the application. Take care not to use lot of RAM, otherwise the STACK area will be corrupted by the function and there could be un-predictable results.

Figure 13. Illustration of execution from RAM



**Rows starting with 'R' are results from RAM execution
**Rows starting with 'A' are results from execution from flash.

Comments

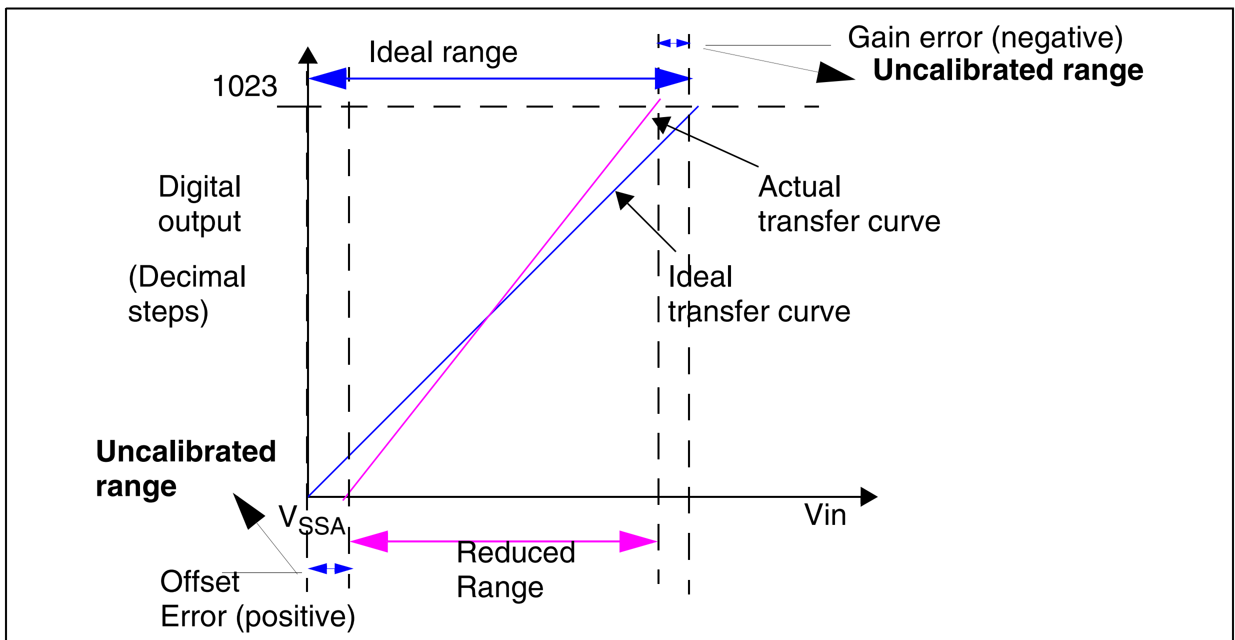
1. The results obtained from executing the ADC conversion from RAM are better. This can be seen in Figure 13 from the difference between the maximum and minimum values. However the results can still be affected by external noise.
2. There can be cases where execution from RAM and execution from flash give the same results.

4 CALIBRATING THE ADC

4.1 CALIBRATION ISSUES

1. If calibration is done on a single channel of the device, the calibration constant is applicable to all the channels.
2. The ADC conversion errors cannot vary from one channel to another unless the source resistance of any of the analog inputs is different, causing a different voltage drop across the source resistance. The R_{AIN} (max) value is provided in the ADC electrical characteristics section of the datasheet.
3. Because of positive offset errors and negative gain errors, the actual range of analog input voltage range will be reduced. The ADC cannot be calibrated outside this range.

Figure 14. 10-bit ADC Calibration Overview



4.2 CALIBRATION METHODS

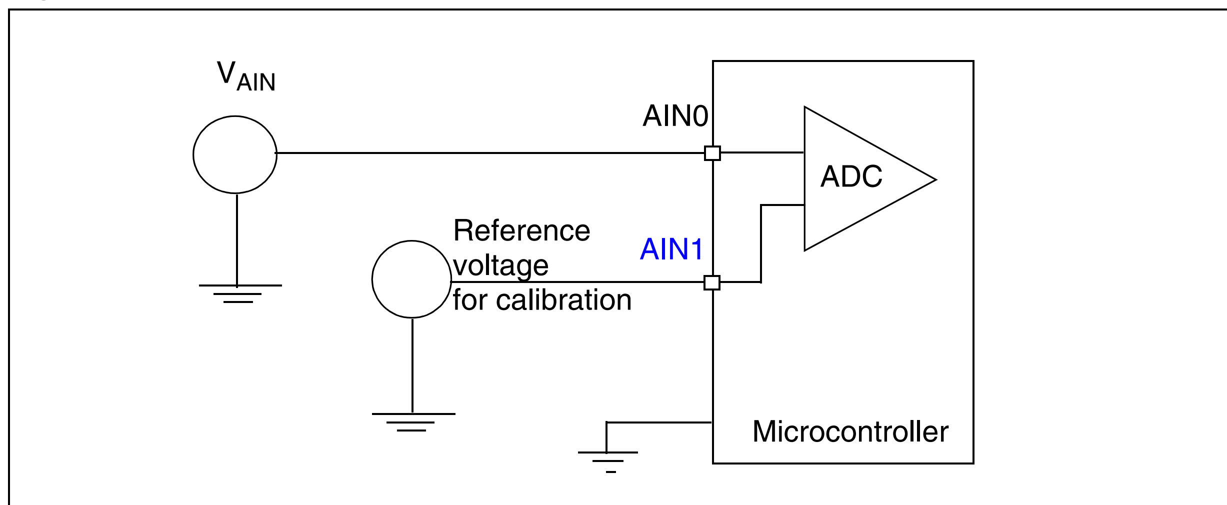
The ADC can be calibrated using a known source so as to get accurate results.

4.2.1 Use accurate voltage reference

This is a simple way to calibrate the ADC. A known reference voltage is connected to a free analog input channel and converted. The digital output received after the conversion can be compared to the already known correct value. The correction factor is then used to correct all other the digital values.

Correction factor = known expected value/Actual value

Figure 15. Hardware setup for calibration



Example

Suppose that the application provides a voltage from a 7805 fixed voltage linear regulator. Because of the tolerance of the linear regulator, the voltage does not stabilise at exactly 5V but remains at 4.900V.

If the full scale reference voltage V_{DDA} is 5V, then the result you would expect from converting a 2.5V input with a 10-bit ADC is:

$$(2.5/5)*1023 = 511.5.$$

So we would expect 512 as converted digital value by ADC.

If the reference voltage is 4.9V the conversion result for 2.5 will be:

$$(2.5/4.9)*1023 = 521.9$$

So we will get 522 as converted result.

This means that with a lower voltage reference (4.9V) than the ideal voltage reference (5V), the ADC will report higher results. So each result should be adjusted by multiplying it with the calibration constant.

So the calibration constant will be (expected value)/actual value = 512/522 = 0.98

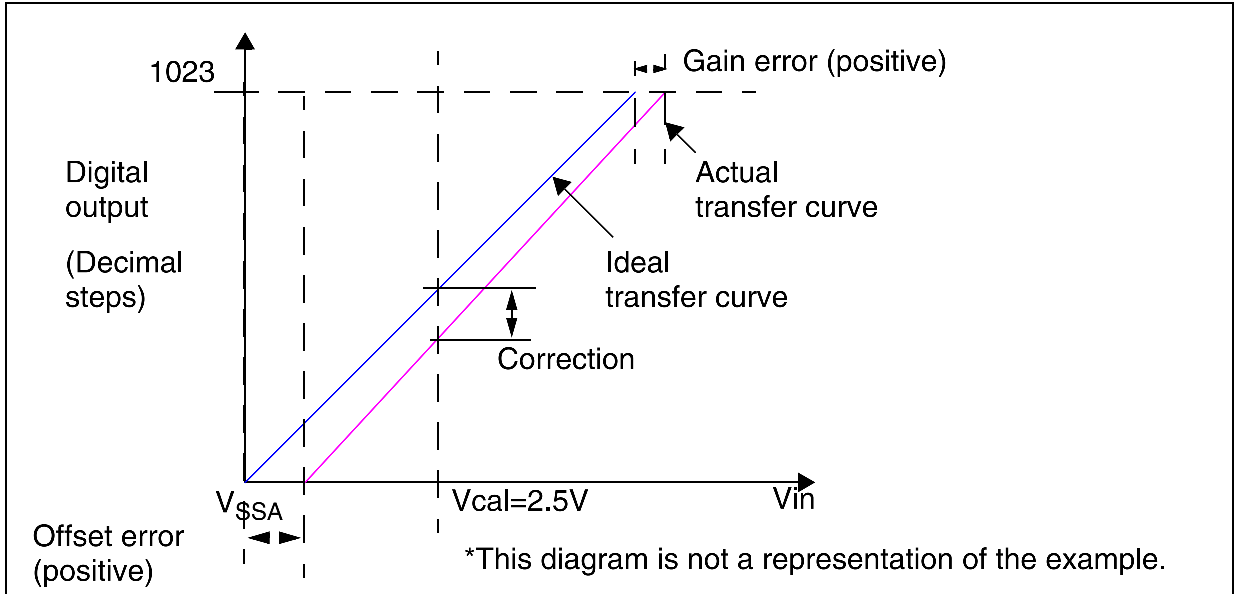
Calibrated ADC result = (calibration constant)*(actual digital result).

Disadvantages

1. The disadvantage of this technique is that the use of very precise voltage references is costly.
2. The error correction made to the reference voltage is applied to all inputs. If there is an error in this input voltage then the same error is applied to all analog inputs.

3. This correction method is an open loop correction method, therefore we cannot guarantee if the actual correction is done.
4. The ‘calibration constant’ is a fraction, calling for floating point arithmetic which requires a lot of RAM. To avoid using the floating point library, you can multiply each result by 100 and divide the final result by 100. This will handle fractions with a precision of up to 2 decimal places. For more precision, you can use higher values like 1000 or 10000.
5. Some errors may be introduced because of calculations (fractions for example).

Figure 16. ADC Calibration



Comments

1. The technique can be useful if the actual transfer curve and ideal transfer curve do not cross each other. This will be the case when both offset and gain errors are positive.
   Also as illustrated in the example, any difference in the reference voltage can be nullified by this technique.
2. You can use Zener diodes and a potential divider formed by resistors to implement a the low cost voltage reference. These discrete components have their own tolerances and drift with temperature etc. so you are advised to verify that the voltage received across the discrete components is same as the expected voltage.
3. If you use a potential divider as a voltage reference, use it only from a precise voltage source, and not from a voltage for which calibration is required i.e V_{DDA} of ADC.

4.2.2 Use of external DAC

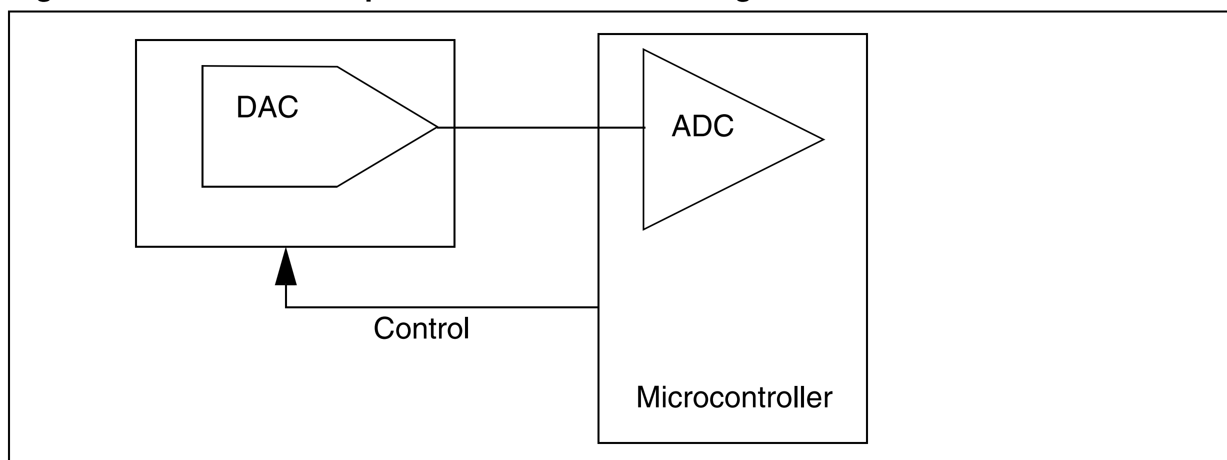
A precise external DAC (Digital to Analog Converter) can be used to provide a software controlled voltage reference. The output received can then be used by the ADC to get the digital value.

The comparison between the expected and actual value will provide the calibration constant. Using the DAC creates a closed loop system and hence this technique is very effective.

Disadvantages

1. The disadvantage of this method is the cost of the extra device which is generally high.
2. The DAC will have its own errors which will effect the calibration, therefore a DAC with very good accuracy is required.

Figure 17. Hardware setup for ADC calibration using DAC



4.2.3 Maintaining a Lookup table

You can maintain a lookup table to correlate the ADC conversion results with the correct result. This requires converting each voltage step with ADC and storing the digital results for each digital code in non volatile (program) memory. This will use lot of program memory.

For example to maintain the lookup table for all 1024 digital codes would require program memory equal to 1024 words. This is equivalent to 2KB memory.

For an 8-bit ADC, the program memory required for the lookup table would be 256 bytes.

Comments:

1. This is the fastest method of getting the correct values for any analog input voltage.
2. It is a very time consuming process to take the readings for all 1023 steps and then maintain the table in the software.

3. The process can be automated using external calibration. A precise analog source can be used for this purpose. The software must make the communication between the source and the device under test.

4. Store the lookup table in EEPROM

The lookup table for different values can be stored in the connected E2PROM. This will make it easy to update the lookup table by calibrating from time to time.

This technique can be used in production. A known and precise analog input voltage source of is used in the test setup and the different voltages are produced. The communication between test setup and microcontroller provides the digital code that can be expected. The actual digital code and expected digital code is then used for making the lookup table.

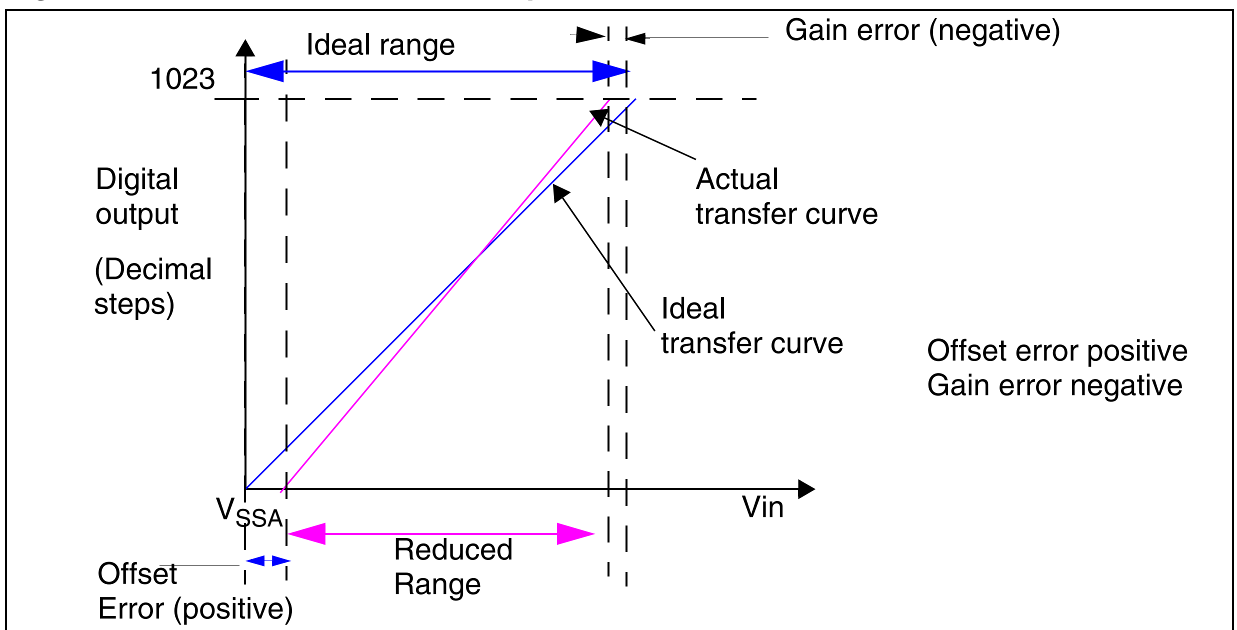
4.2.4 Linear compensation

Linear compensation can be done using the datasheet values for Offset and Gain errors. The ideal transfer curve is a straight line from code 0 to 1023 for a 10-bit ADC.

The actual ADC transfer curve is assumed to follow a straight line from first transition (digital code =1) to last transition (digital code = 1023).

The first transition and last transition error values are known from the datasheet. If the Offset error is positive and the Gain error is negative then the available range is reduced. In this case the actual transfer curve will cross the ideal transfer curve.

Figure 18. Offset and Gain error compensation



To do linear compensation, the offset and gain errors can be spread over the full range. It can be noted from the transfer curve that positive offset errors cause the actual result to be re-

ported as less than the expected result and negative offset error causes the reported result to be more than the expected result.

For example: Offset error = 3 LSBs

Gain error = -1 LSB

So total error = 4 LSBs for 1023 steps.

To avoid using floating point arithmetic, we can take 1 LSB per 255 steps (approximately) to be compensated.

Therefore, add 3 to digital results in the range 1-255.

256-510 add 2

511 - 765 add 1

765 - 1023 subtract 1

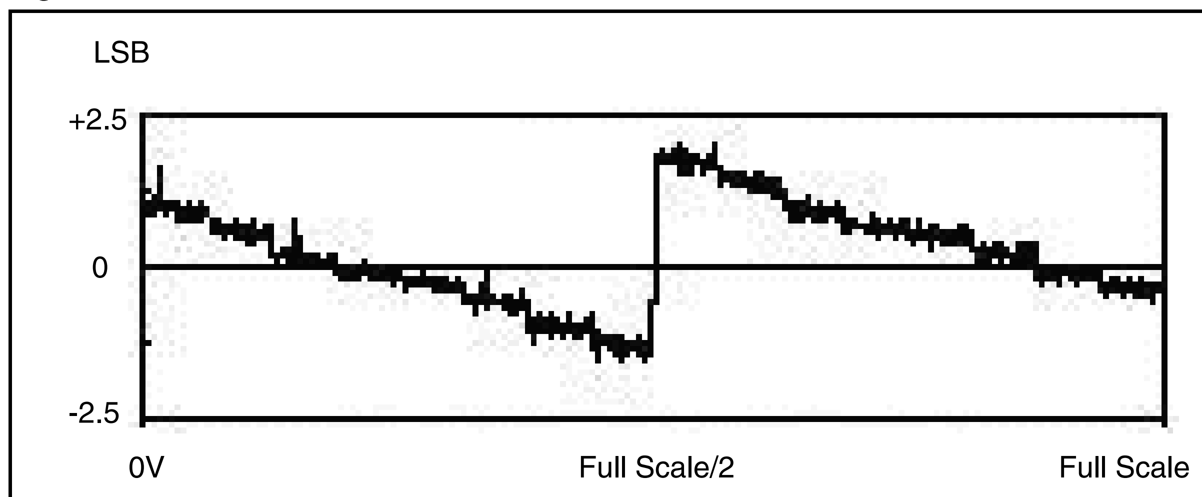
Caution: This technique is provided for theoretical understanding. The error distribution may not be linear. Please refer to the Zone compensation as below.

4.2.5 Zone compensation

As explained in the linear compensation technique, the ADC errors may not be equally distributed over the entire analog input range.

There are analog ranges in which there are positive errors and other ranges with negative errors. Figure 19 shows the approximate error distribution curve. The actual values on the Y axis depend on the device.

Figure 19. Error distribution



Thus, we can divide the ADC converted values into zones which have positive and negative errors. To simplify the calculations we can use following zones for error correction.

00-0x7F and 0x200 to 0x27F: +2

0x80-0xFF and 0x280 to 0x2FF: +1

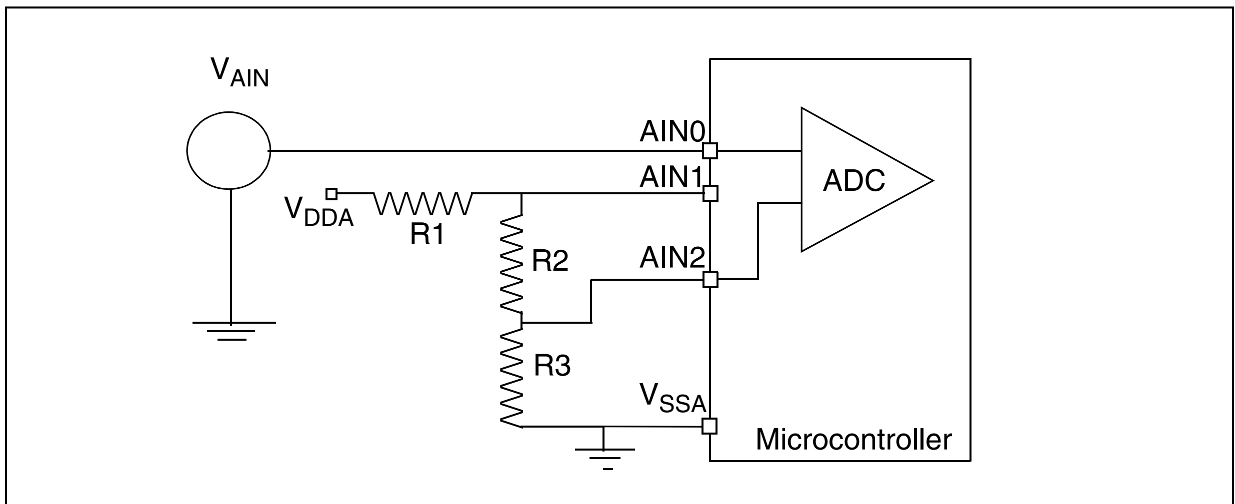
0x100-0x17F and 0x300 to 0x37F: -1

0x180-0x200 and 0x380-0x3FF: 2

4.2.6 Autocalibration for Offset and Gain errors

Approximate Offset and Gain errors can be calculated by software and external hardware and these parameters can be used for calibrating the ADC. Use a spare analog channel for calculating the offset and gain error. Use an external resistor network to get the known voltages near to ground and V_{DDA}. Do the ADC conversion on this spare ADC channel and calculate the Offset and Gain error.

Figure 20. Hardware setup for calculating offset and gain errors



Example:

Get the low voltage on AIN2 equivalent to 10LSB and get the voltage on AIN1 as (1023-10)LSB for a 10-bit ADC. After doing the ADC conversion on AIN2, the digital code received can be compared to 10LSB and offset error can be calculated. This will be an approximate value and assumes a straight line transfer curve from the first transition to the 10LSB. Similarly gain error can be calculated and compensated.

The resistor should be precise with less than 1% tolerance.

Resistance values chosen were R1=R3=100 Ohm, R2=9.8K Ohm.

For V_{DDA} = 5V,

Expected digital output on AIN2 = (100/10K)*1023 = 10 (approx.)

Expected digital output on AIN1 = ((100+9.8K)/10K)* 1023 = 1012.7 = 1013 (approx.)

Offset error (approx.) = Ideal digital value - Actual digital value

Eo approx. = 10-(DIgital code AIN2)

Gain error (approx.) = Actual digital value - Ideal digital value

Eg approx. = (Digital code AIN1)-1013

Please note that the offset error is the difference between first actual transition and first ideal transition. When the actual digital code for same input voltage is received less than the expected (ideal) analog input, this means that a higher input voltage will be required to produce the same digital code. This results in a positive offset error. So we have reversed the polarity and used the ideal digital value - actual digital value, instead of actual analog value- ideal analog value. This calculation is approximate. The results received thus can be used for calibrating the ADC.

Similarly we can calculate the Gain errors.

Mathematic calculations

Assuming the ADC transfer curve is a straight line, we obtain the linear function:

$$Y = ax+b \quad ....[1]$$

where Y is the converter output i.e digital code

a is the slope of the transfer curve

x is the input/ actual reading

b is the offset

For the 1st conversion result of an input voltage equivalent to 10LSB:

$$Y1 = a * x1 + b \quad .... [2]$$

For the 2nd conversion result of an input voltage equivalent to 1013LSB:

$$Y2 = a * x2 + b \quad ..... [3]$$

Subtracting [3] - [2]

$$Y2-Y1 = a (x2-x1) \quad ......[4]$$

$$\text{slope } a = (Y2-Y1)/(x2-x1) \quad ....[5]$$

Putting this value of ‘a’ in the equation [2]

$$b = Y1- ((Y2-Y1)/(x2-x1)) * x1 \quad .....[6]$$

as, $Y = ax + b$

$x = (Y-b)/a$

Therefore after getting the ADC converted values (Y) we can calculate the actual or ideal analog input signal (x) after knowing the slope of transfer curve (a) and offset constant (b).

Practical measurement

To test the calibration technique, we can use the test setup shown in Figure 20. Analog Channel 1 is used for gain error calculation and Channel 2 was used for offset calculation. The test results shown in Table 2. were obtained.

To avoid using floating point variables and the floating point library, all the calibration constants were multiplied by a constant N (= 1000000) and ‘long’ type variables are used.

V_{REF}= 4.9814 V

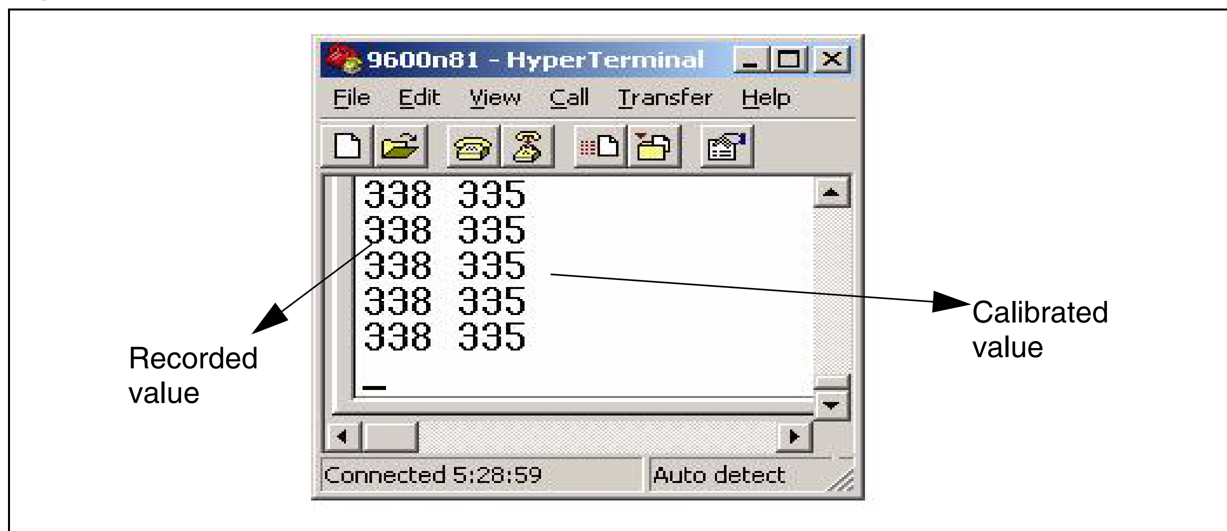
Table 2. Calibration Results

Vin	Ideal results	Recorded value	Calibrated value
0.509 V	104	102	103
1.005 V	206	205	206
1.502 V	308	309	309
2.005 V	412	413	412
2.503 V	514	514	513
3.001 V	616	616	617
3.504 V	719	720	718
4 V	821	824	821
4.5 V	924	927	924
4.97	1020	1023	1018

Comments

1. It can be noted that calibration helps eliminate the errors.
2. In most cases the calibrated value was close to the ideal or expected value.
3. Calibration results are much better in the higher voltage range (>3.5V).
4. For some analog input ranges (approx. 2V to 3V) the recorded values are the same as expected, and the calibration introduced some errors. This is because of non linearity of the ADC. We have assumed the actual transfer curve was a straight line, whereas in practice the ADC transfer curve may exhibit non-linearity.

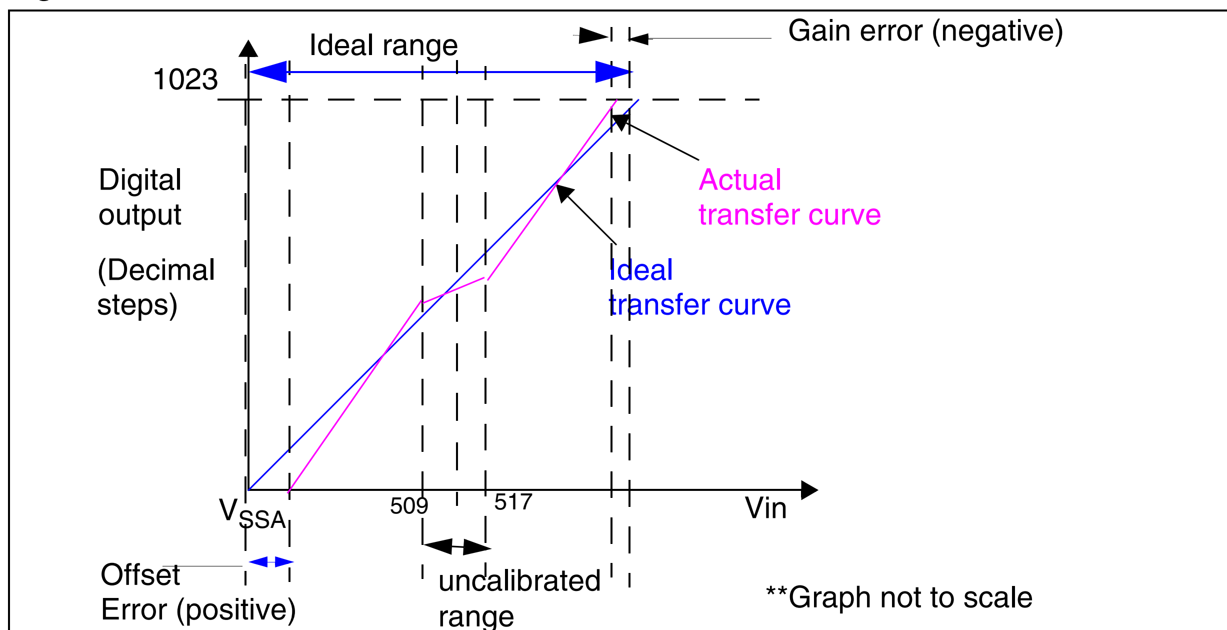
Figure 21. Illustration of calibration



4.2.7 Calibration for Errors using 2 different zones

We can extend the technique mentioned above to do 2-zone calibration. This is required because of the linear distribution of errors from offset error to mid-voltage and then from mid-voltage to gain error as shown in Figure 22.

Figure 22. Calibration for 2 zones



In addition to approximate Offset and Gain errors we need to calculate the error around the center of the analog voltage range. A small voltage range around mid-voltage is to be ignored

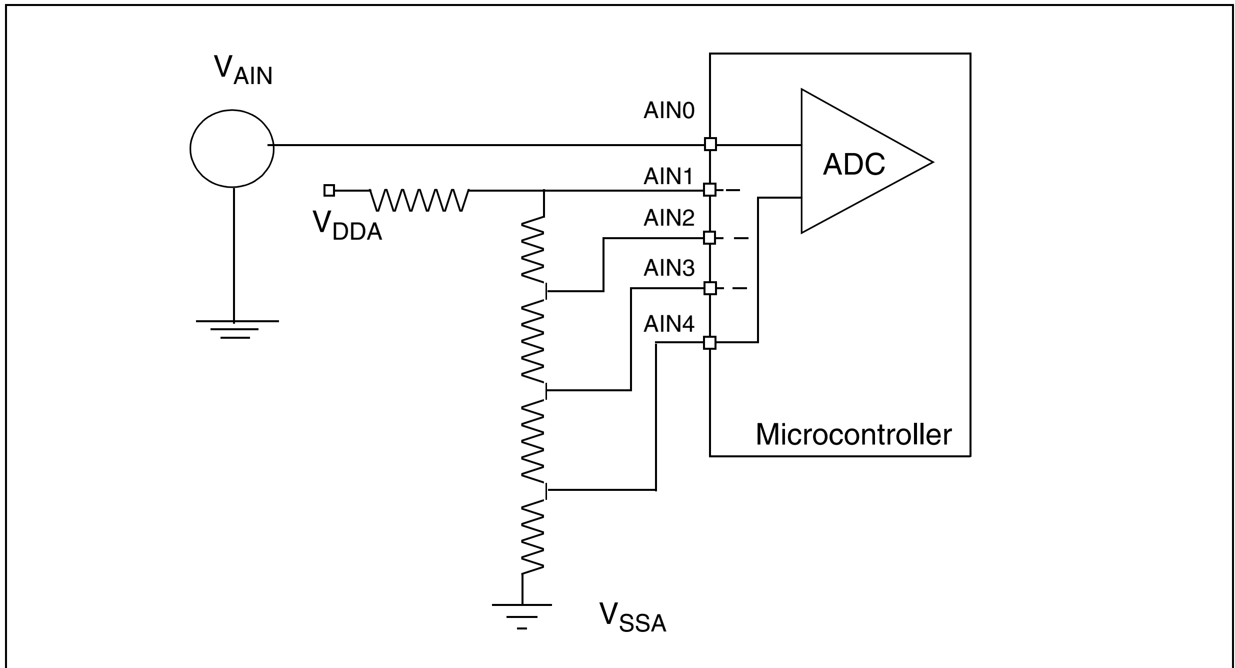
for calibration and this range will not be calibrated. Thus, we need 4 external voltage references for calibration.

For example:- We can choose to calculate the offset error at 10LSB and take the 2nd point of the line at 509LSB.

We can ignore the analog input range from 509LSB to 517LSB.

For second line we take the first point as 517LSB and second as 1013LSB.

Figure 23. Hardware setup for 2-zone calibration



Using the same software technique mentioned in Section 4.2.6 the actual value can be calculated from the converted value.

Comments

1. This technique enables you to able to get close to the ideal result.
2. It requires external resistors and uses four analog channels for calibration and hence, it is costly.
3. Some current will flow from the external resistor network.

5 SOFTWARE

The software available with this application note is provided in C language. The software is provided for guidance and you can use it directly or change it to meet your requirements. For demonstration and easy usability, some parts of the code are repeated. For example, averaging etc.) is tested on ST72F324 and ST72F521 but it can be used with other ST7 devices also. The software covers the following techniques.

Table 3. General techniques for improving ADC accuracy

No.	Software technique	Software available
1	Averaging	Yes
2	Averaging using queue implementation	Yes
3	Histogram technique	Yes
4	Noise filtering technique	Yes

Table 4. Noise reduction techniques

No.	Noise Reduction technique	Software available
1	Using Slow mode	Yes
2	Inserting NOPs	Yes
3	Using WFI mode	Yes
4	Executing code from RAM	Yes

Table 5. Calibration techniques

No.	Calibration technique	Software available
1	Use of accurate voltage reference	No
2	Use of external DAC	No
3	Maintaining lookup table	No
4	Using Linear compensation	No
5	Zone compensation	Yes
6	Autocalibration of offset and gain errors	Yes

5.1 FILE PACKAGE

The package provided with this application note contains the workspace, make-files and the source code.

You can include ADC_tech.c and ADC_tech.h in your workspace to use the ADC software techniques.

5.1.1 ADC_tech.h

This file contains the Parameters, Constants (#define), and some macros, which are used in the ADC_tech.c file

It also contains the prototype declarations for the functions contained in the ADC_tech.c file.

5.1.2 ADC_tech.c

This file contains the software for the ADC techniques described in this application note. It contains the different functions written in C and inline assembly code is used wherever necessary.

5.1.3 Main.c

This file contains the various test routines for demonstrating the ADC techniques. The data received from the different routines is transmitted to the PC through the RS232.

5.2 DEPENDENCIES

The software uses the ST7 Software Library, which is freely available on http://mcu.st.com

This library path used is c:\ST7_lib. You are requested to modify the path in default.env file to specify the path of ST7 software library.

The software was tested using the HIWARE toolchain.

Some parts of the source code directly access the hardware registers which is not normally recommended when using ST7 Library. This is done to reduce the code size and execution time so as to demonstrate the ADC technique under discussion. Necessary software modifications are done to support this direct access to the registers by including peripheral register files.

5.3 GLOBAL VARIABLES

The software uses global variables depending on the ADC technique. You can select a technique by enabling the corresponding declaration in adc_tech.h.

Some global variables are used only for demo purposes and they can be enabled by defining ADC_TEST as 1 in ADC_tech.h

“#define ADC_TEST 1”

All the variables are placed in “default RAM”. Depending on the device, you can place the variables in “short memory” to save code size and execution time.

5.4 INTERRUPTS

Only one interrupt is used for Timer A. This is used for demonstrating “Wait for interrupt”. You are advised to make necessary changes in the software when using this technique for example, in PRM file etc.

5.5 CODE SIZE AND EXECUTION TIME

The following table summarises the approximate code size and execution time. Depending on the compiler and memory placement, these values can change.

The RAM requirements are not provided and you can choose to place the variables as global or local. For techniques which require arrays, the size of array will define the RAM requirements and it is left up to the user.

Table 6. Code size and execution time

No.	Function Name	Code size (Bytes)	Execution time (us)
Averaging technique			
1.	ADC_get_avgdata	63	311
Averaging using queue implementation			
2.	ADC_fill_Q	68	178
3.	ADC_avg_Q	104	31
Noise filtering technique			
4.	ADC_getdata_filter_noise	155	717
Histogram technique			
5.	ADC_getdata_histgrm	669	6550
Execute from RAM			
6.	ADC_do_Conversion	15	10
7.	copy_code_ROM_2_RAM	44	109
8.	ADC_get_avg_execRAM	97	541
Zone compensation			
9.	ADC_zone_compensation	149	24
Wait for Interrupt technique			
10.	TimerA_Init	19	4
11.	ADC_conversion_WFI	14	36**

No.	Function Name	Code size (Bytes)	Execution time (us)
12.	TIMERA_IT_Routine	46	11
13.	ADC_get_avg_WAIT	129	700
**Time for ADC_conversion_WFI is including 2 ADC conversions done in WFI mode			
ADC calibration			
14.	Get_transfer_line_constant	133	484
15.	Get_calibration constant	160	1.2ms
16.	Get_corrected_value	113	778

6 RELATED DOCUMENTS

You can refer to the following application notes for additional useful information:

ADC Application notes

AN1636: Understanding and minimising ADC conversion errors

AN672: Optimizing the ST6 A/D Converter accuracy

AN1548: High resolution single slope conversion with analog comparator of the 52x440

Noise, EMC related documents

AN435: Designing with microcontrollers in noisy environment

AN898: EMC General Information

AN901: EMC Guidelines for microcontroller - based applications

Software techniques

AN1015: Software techniques for improving microcontroller EMC performance

AN985: Executing code in ST7 RAM

“THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNECTION WITH THEIR PRODUCTS.”

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics.

All other names are the property of their respective owners

© 2004 STMicroelectronics - All rights reserved

STMicroelectronics GROUP OF COMPANIES

Australia – Belgium - Brazil - Canada - China – Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com