AN4073
Application note

How to improve ADC accuracy when using STM32F2xx, STM32F40x and STM32F41x microcontrollers

Introduction

The purpose of this application note is to show how to improve the accuracy of A/D conversions for applications using the STM32F20x / STM32F21x (revZ, revY, revX) and STM32F40x/STM32F41x (revZ) microcontrollers.

It also explains the firmware methodology, which can be applied to reduce the ADC error. A reference application is shown with implementation of firmware filtering techniques (using averaging algorithms) to minimize the ADC errors.

The document also gives some general tips on writing firmware for better ADC accuracy.

Please note that the data provided with this application note is for reference only, measured in a lab under typical conditions (unless specified otherwise) and not tested in production.

Table 1 lists the microcontrollers concerned by this application note.

Table 1. Applicable products

Type	Applicable products
Microcontrollers	STM32F20x / STM32F21x (revZ, revY, revX) and STM32F40x/STM32F41x (revZ) microcontrollers

Contents

1 Overview of parameters impacting the ADC accuracy

The accuracy of an analog to digital conversion has an impact on the overall system quality and efficiency. To improve the accuracy, you need to understand the errors associated with the ADC and the parameters affecting them.

The ADC, itself, cannot ensure the accuracy of results. It depends on your overall system design. For this reason, you need to do some careful preparation before starting your development.

Many parameters impact the ADC accuracy, depending on the application. Some of these factors are: PCB layout, voltage source, I/O switching and analog source impedance.

For more details about ADC errors, please refer to AN2834: *How to get the best ADC accuracy in STM32F10xxx devices* and to AN3137: *A/D converter on STM8L devices* application notes.

2 Firmware techniques for improving the conversion accuracy

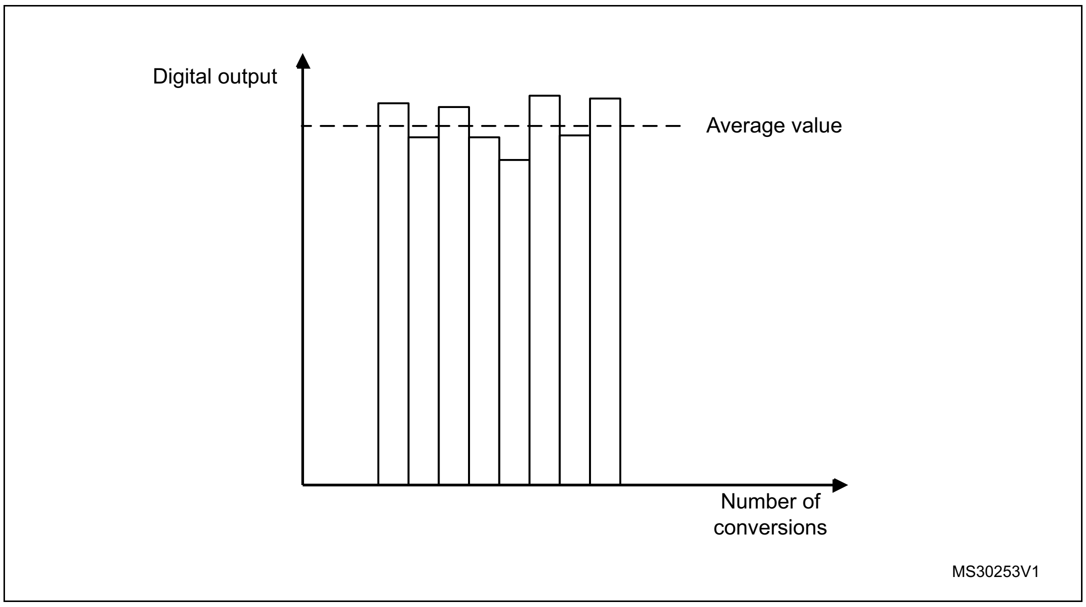
2.1 Averaging

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

2.1.1 Averaging of N ADC samples

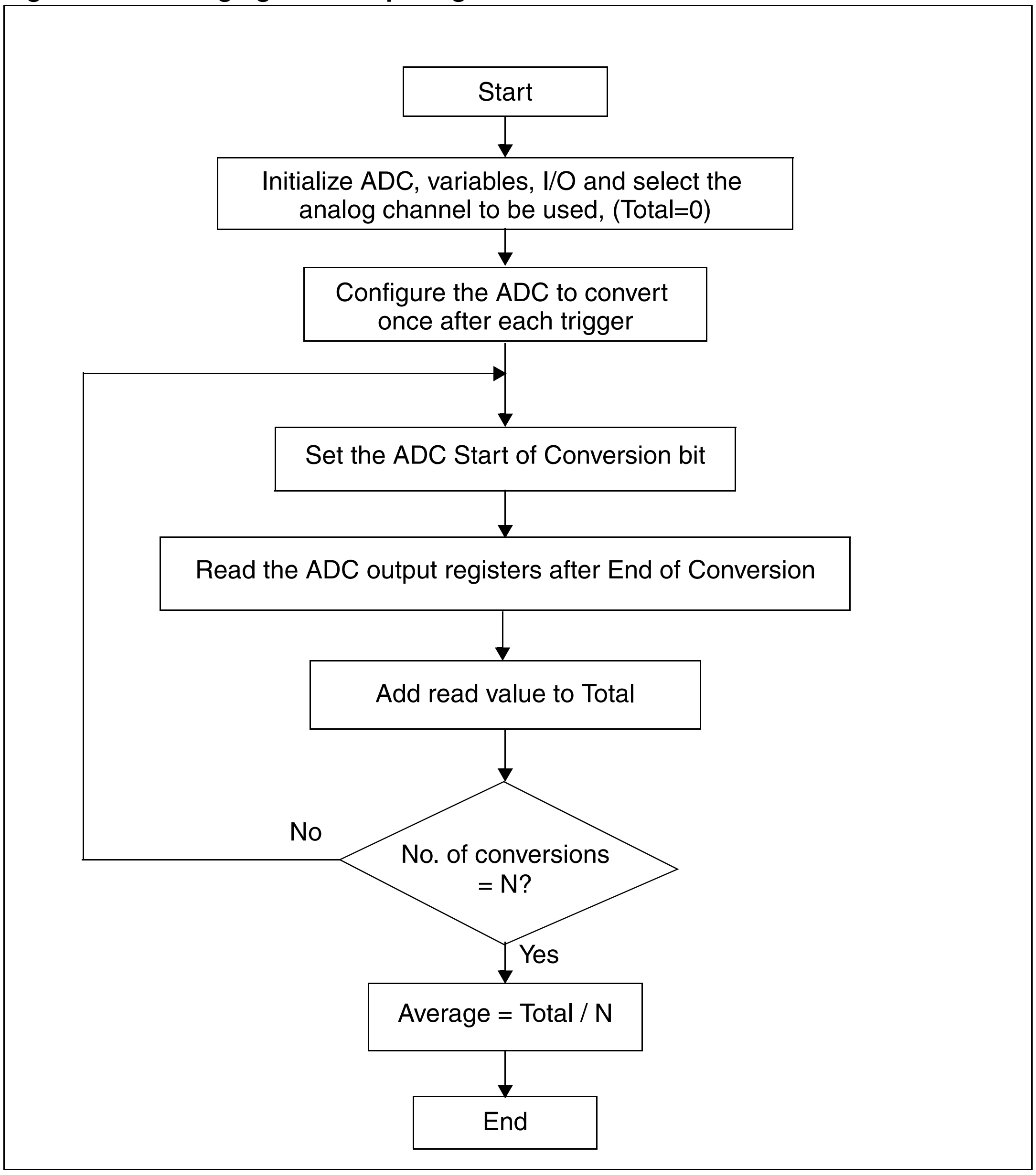
When this method is used, it is better to collect the samples in multiples of 2 (N should be a multiple of 2). This makes it more efficient to compute the average because you can do the division by right-shifting the sum of the converted values. This saves CPU time and code memory needed to execute a division algorithm (in Cortex-Mx, this takes 1 CPU cycle).

Figure 1. Graphical representation of averaging technique



This averaging technique is used to measure the voltage on one analog input pin. A total of N conversions is considered and the average is calculated. This is done in a loop in the firmware.

Figure 2. Averaging of N sample algorithm


Start
Initialize ADC, variables, I/O and select the analog channel to be used, (Total=0)
Configure the ADC to convert once after each trigger
Set the ADC Start of Conversion bit
Read the ADC output registers after End of Conversion
Add read value to Total
No. of conversions = N?
No
Yes
Average = Total / N
End

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 trade-off between the total conversion time and the number of samples used to average, depending on the analog signal variations and the time available for computation.

Note: Refer to Appendix A for more details about the code source used.

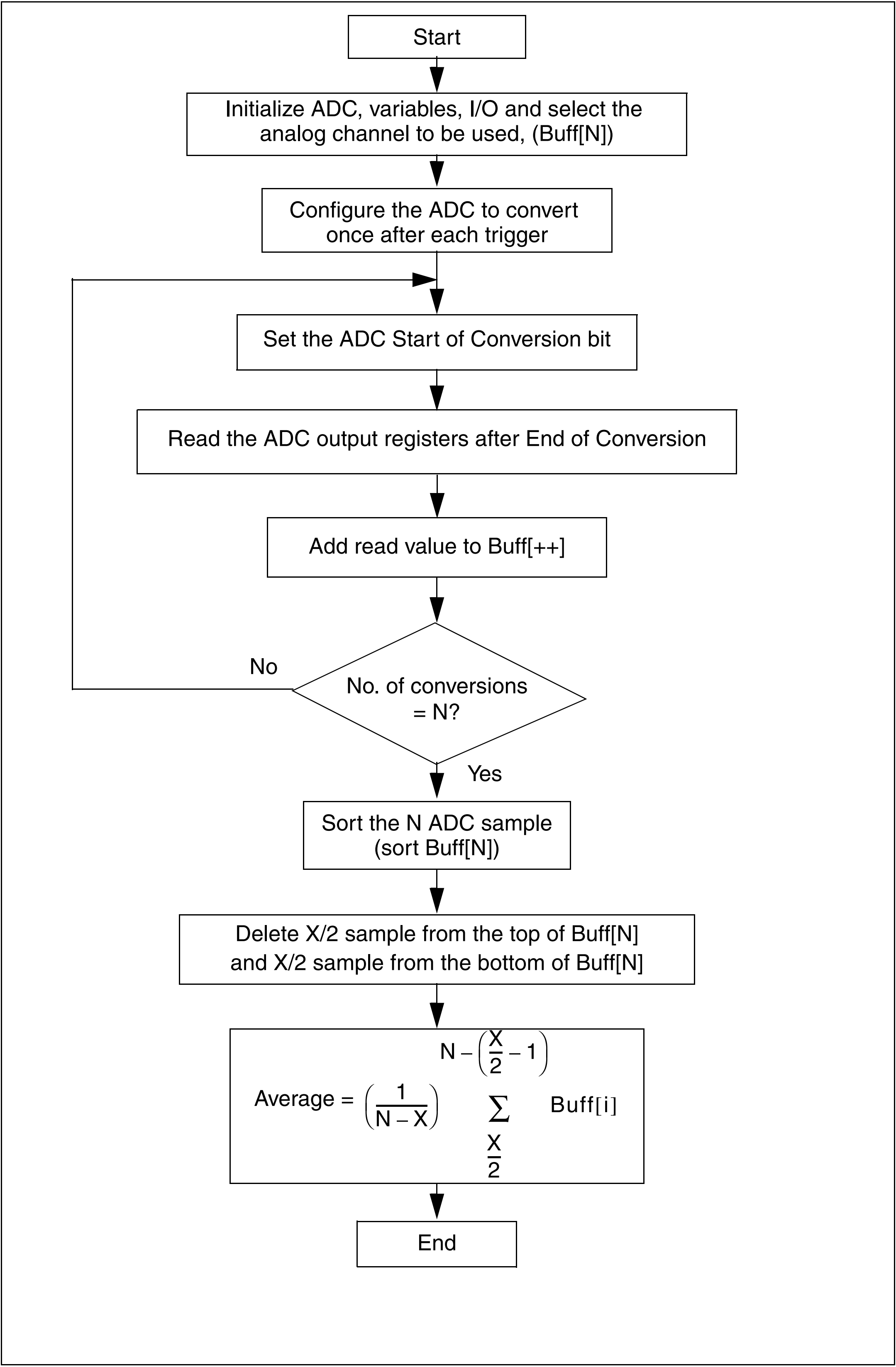
2.1.2 Averaging of N-X ADC samples

This method is based on taking N ADC samples, sorting them from the highest to the lowest value (or the reverse) and deleting the dispersed X samples.

It is recommended to choose N and X as multiples of 2.

This averaging method is more efficient than the previous one, as it deletes the most dispersed values which could impact the average, and it gives a good trade-off between the execution time and the conversion accuracy.

Figure 3. Averaging of N-X ADC sample algorithm


Start
Initialize ADC, variables, I/O and select the analog channel to be used, (Buff[N])
Configure the ADC to convert once after each trigger
Set the ADC Start of Conversion bit
Read the ADC output registers after End of Conversion
Add read value to Buff[++]
No. of conversions = N?
No
Yes
Sort the N ADC sample (sort Buff[N])
Delete X/2 sample from the top of Buff[N] and X/2 sample from the bottom of Buff[N]
$\text{Average} = \left(\frac{1}{N-X}\right) \sum_{\frac{X}{2}}^{N-\left(\frac{X}{2}-1\right)} \text{Buff}[i]$
End

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

Computation time = time taken to read the results, sort the N ADC samples, delete the X most dispersed samples, add the rest of the ADC samples together and calculate the average by dividing the total by N-X.

Note: Refer to Appendix B for more details about the code source used.

2.2 Additional recommendation

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 not be accurate. Both hardware and firmware 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_{REF}) and VSSA analog supply pins are available on the microcontroller package; you can connect a capacitor filter to these power supply pins to filter a high frequency noise.

Here are some general firmware design tips for reducing system noise in order to achieve a better ADC conversion accuracy:

1. Avoid starting transmission on any communication peripheral just before starting the ADC conversion. The toggling of the I/Os may create some noise in the supply voltage.
2. Avoid toggling high-sink I/Os which cause noise ripples in the power supply.
3. Avoid toggling digital outputs on the same I/O port as the A/D input is being converted. This can introduce switching noise into the analog inputs.
4. It is recommended to configure the STM32F2/F4 ART with data cache + Instruction cache enable and to disable the prefetch. This will avoid extra CPU accesses to the Flash memory causing an additional noise which can significantly decrease the ADC accuracy in some applications.

3 Practical measurements

This section gives some ADC accuracy measurements, putting the previously described methods into practice.

3.1 Measurement conditions

3.1.1 Hardware setup

- STM32F407ZGT6 soldered on a test board (with only a minimum number of other hardware components)
- Ambient temperature: 25°C
- 3 power supplies are applied to the MCU: V_{DD}/V_{SS}, V_{DDA}/V_{SSA}, V_{REF+}/V_{REF-}
- Power supply range: $V_{DD}=V_{DDA}=V_{REF+}$ = 3.3 V, f_{ADC} = 36 MHz
- Clock source: external clock (8 MHz) provided by a generator, PLL is enabled, f_{CPU} = 144 MHz
- Three fixed analog input voltages are tested: 0.3 V, 1.65 V and 3V

3.1.2 Firmware setup

- ADC channel 2 is used in single conversion mode
- The sampling time is fixed to 3 ADC clock cycles
- 50000 acquisitions were taken for each fixed analog input voltage
- Five firmware methods were used to examine the ADC converted data:
  - Raw data (without averaging)
  - Averaging of 4 ADC samples
  - Averaging of 6 ADC samples
  - Averaging of 8 ADC samples
  - Averaging of 8 ADC samples and deleting the 4 most dispersed samples

Note: All tests were done with f_{ADC} = 36 MHz, sampling time = 3 ADC cycles and ADC resolution = 12 bits in order to achieve the fastest ADC conversion (2.4 Msps).

3.2 Results

These measurements have been done based on a characterization of the ADC and are confirmed by further investigations (about 100 tests), where many parameters were evaluated to analyze the impact of each one.

The following measurements were considered while evaluating the impact of three different ART configurations on the ADC accuracy:

- ART ON: (data cache + instruction cache + prefetch) ON
- ART OFF: (data cache + instruction cache + prefetch) OFF
- (Data cache + instruction cache) ON + prefetch OFF

3.2.1 ADC measurements when ART is ON

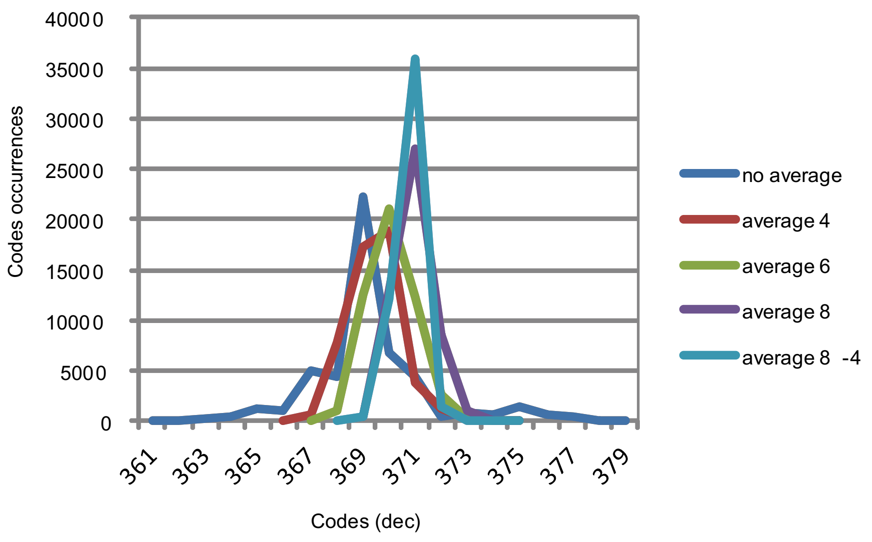
The following table and graph show the ADC code distribution when ART is ON.

Table 2. Distribution of ADC codes when ART is ON (units in LSB)

V_{IN}	Raw data		Averaging by 4		Averaging by 6		Averaging by 8		Averaging by 8 delete 4	
	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB
0.3 V	19	6.37 %	10	0 %	9	0 %	7	0 %	8	0 %
1.65 V	21	7.90 %	13	0.05 %	10	0 %	10	0 %	9	0 %
3 V	21	21.53 % (1)	15	0.38 %	13	0.13 %	12	0.006 %	12	0.04 %

1. Worst case found in a total of more than100 tests

Figure 4. Distribution of ADC codes when ART is ON and V_{IN} = 0.3 V



3.2.2 ADC measurements when ART is OFF

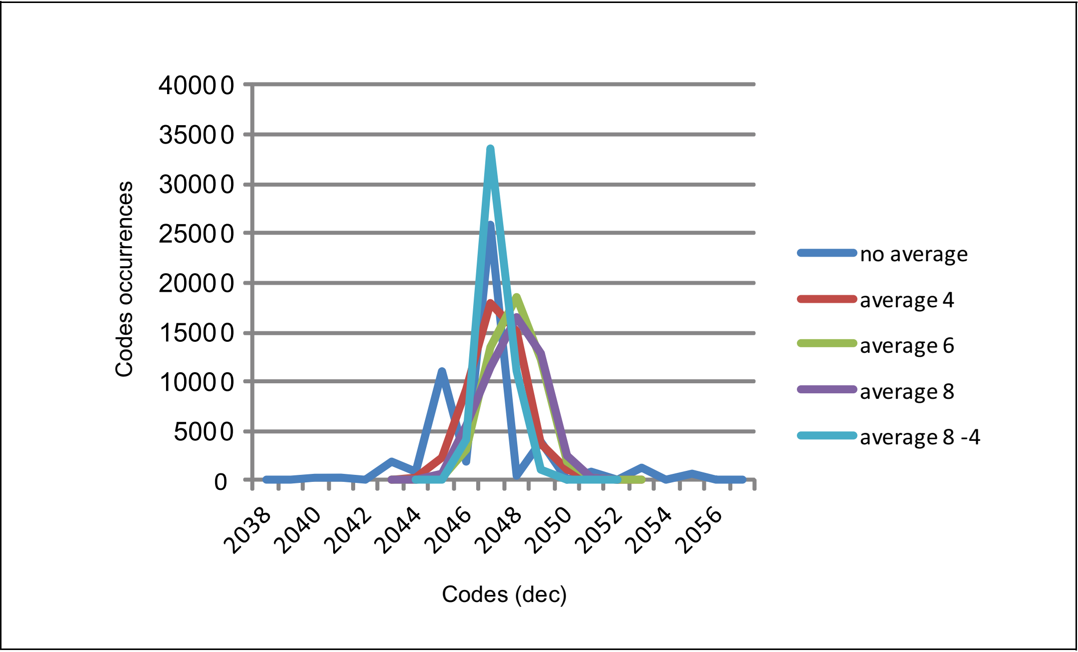
The following table and graph show the ADC code distribution when ART is OFF.

Table 3. Distribution of ADC codes when ART is OFF (units in LSB)

V_{IN}	Raw data		Averaging by 4		Averaging by 6		Averaging by 8		Averaging by 8 delete 4	
	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB
0.3 V	18	4.07 %	11	0.004	9	0 %	7	0 %	8	0 %
1.65 V	20	5.99 %	11	0.01 %	11	0.002 %	10	0 %	9	0 %
3 V	24	16.28 % (1)	18	6.92 %	13	0.044 %	11	0.008 %	12	0.028 %

1. Worst case found in a total of more than100 tests

Figure 5. Distribution of ADC codes when ART is OFF and V_{IN} = 1.65 V



3.2.3 ADC measurements when (Data+Instruction) cache ON + prefetch OFF

The following table and graph show the ADC code distribution when the (Data+Instruction) cache is ON and prefetch is OFF.

Table 4. Distribution of ADC codes when (Data+Instruction) cache ON + prefetch OFF (units in LSB)

V_{IN}	Raw data		Averaging by 4		Averaging by 6		Averaging by 8		Averaging by 8 delete 4	
	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB	Dis-persion (LSB)	Sample over +-5 LSB
0.3 V	16	0.06 %	7	0 %	5	0 %	4	0 %	4	0 %
1.65 V	18	0.064 %	8	0 %	6	0 %	5	0 %	4	0 %
3 V	17	0.068 %	8	0 %	6	0 %	4	0 %	4	0 %

Figure 6. Distribution of ADC codes when (D+I) cache ON + prefetch OFF V_{IN} = 0.3V

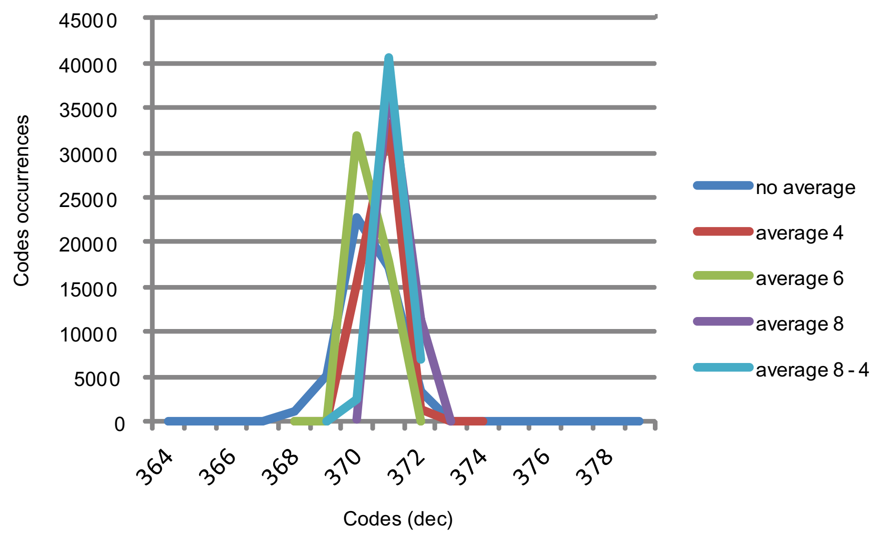


Figure 7. Distribution of ADC codes when (D+I) cache ON + prefetch OFF V_{IN} = 1.65 V

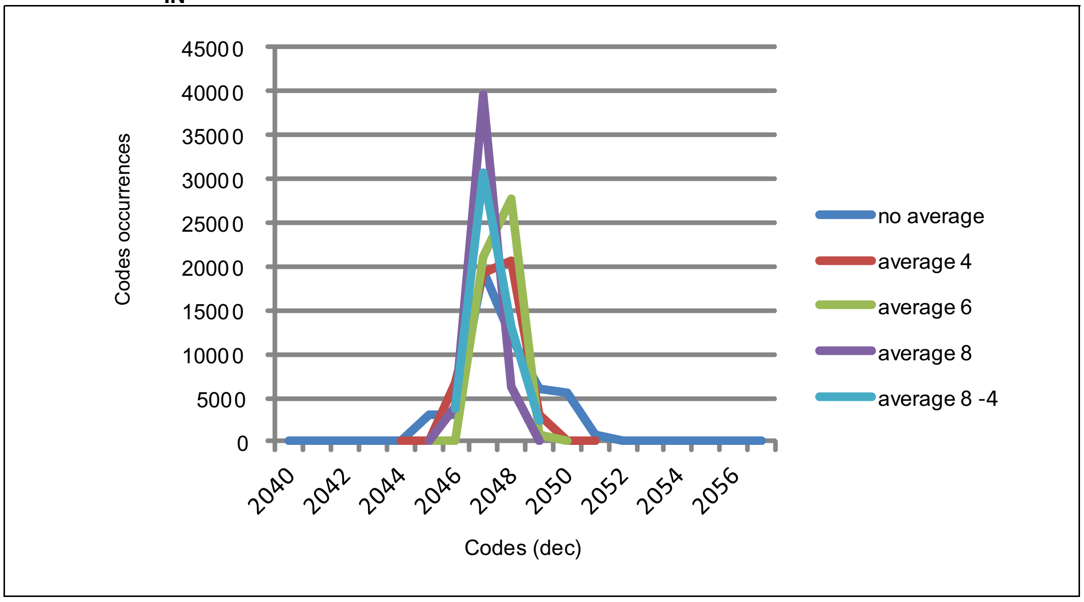
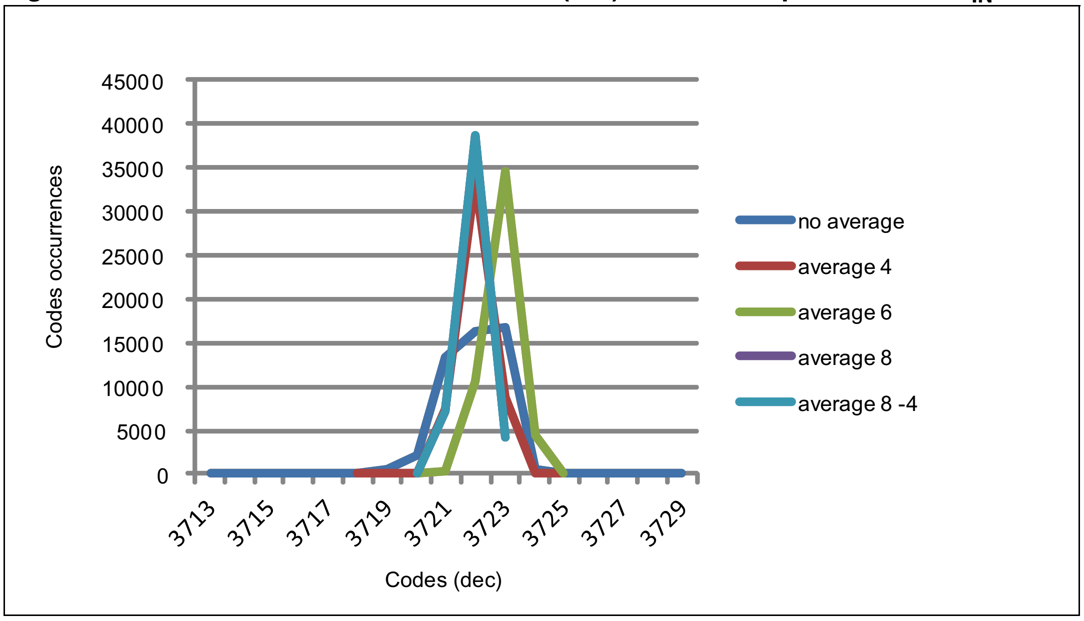


Figure 8. Distribution of ADC codes when (D+I) cache ON + prefetch OFF V_{IN} = 3V



3.3 Timing considerations

The following table gives an idea of the execution time of each averaging algorithm in terms of CPU cycles.

Note: *These timing measurements are considered based on the optimum ART configuration: (Data+Instruction) cache ON, Prefetch OFF, which results in the best ADC noise immunity.*

In other ART configurations, these measurements could result in a difference of a few cycles.

Table 5. Time needed to compute averaging

Computation time	Averaging by 4 (1)	Averaging by 6 (1)	Averaging by 8 (1)	Averaging by 8, delete 4 (2)
CPU cycles(3)	18	26	36	517

1. Computation = time taken to add the N samples and divide them by N.
2. Computation = time taken to sort the 8 ADC samples, delete the 4 dispersed ones, add the other 4 samples and divide them by 4. This result depends on the sorting algorithm (in this case, the sorting algorithm is based on successive permutation, which is not optimized for speed).
3. CPU frequency is 144 MHz, as described in *Section 3.1.1*.

*Note: Please note that the time required to get N samples = N * ($t_{SAMPLING}$ + $t_{CONVERSION}$)*

3.4 Conclusion

In conclusion, to obtain the most accurate conversion results when using the STM32F2/F4 ADC, care must be taken to configure the ART Flash memory accelerator accordingly. This configuration must be (Data+Instruction) ON and prefetch OFF to achieve the best accuracy by limiting the internal noise generated by the Flash.

This application note shows how you can choose the best trade-off between A/D conversion accuracy and the required A/D conversion speed.

For example, the measurements given above demonstrate that, with a 12-bit ADC resolution, +- 5 LSB can be achieved using an averaging by 4 algorithm (0.6 Msps) when the ART Flash memory accelerator is configured with (Data+Instruction) ON and prefetch OFF.

Appendix A Averaging of N ADC samples: source code

```
/**
  * @brief   Get the average of N ADC samples
  * @param Numbre of ADC samples to be averaged
  * @retval The average value
  */
uint16_t ADC_GetSampleAvgN(uint8_t N)
{
  uint32_t avg_sample =0x00;
  uint16_t adc_sample[8]={0,0,0,0,0,0,0,0};
  uint8_t index=0x00;

  /* Get the N ADC samples */
  for (index=0x00; index<N; index++)
  {
      /* ADC start conv */
   ADC_SoftwareStartConv(ADC1);
    /* Wait end of conversion */
    while(ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC) == RESET);
    /* Store ADC samples*/
    adc_sample[index] = ADC_GetConversionValue(ADC1);
  }

  /* Add the N ADC samples */
  for (index=0; index<N; index++)
  {
   avg_sample += adc_sample[index];
  }

  /* Compute the average of N ADC samples */
  avg_sample /= N;

    /* Return average value*/
  return avg_sample;
 }
```

Appendix B Averaging of N-X ADC samples: source code

```
/**
  * @brief   Get the average of N-X ADC samples
  * @param Numbre of ADC samples to be averaged
  * @param Numbre of ADC samples to be averaged
  * @retval The average value
  */
uint16_t ADC_GetSampleAvgNDeleteX(uint8_t N , uint8_t X)
{
  uint32_t avg_sample =0x00;
  uint16_t adc_sample[8]={0,0,0,0,0,0,0,0};
  uint8_t index=0x00;


  for (index=0x00; index<N; index++)
  {
      /* ADC start conv */
   ADC_SoftwareStartConv(ADC1);
    /* Wait end of conversion */
    while(ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC) == RESET);
    /* Store ADC samples */
    adc_sample[index] = ADC_GetConversionValue(ADC1);
  }

  /* Sort the N-X ADC samples */
    Sort_tab(adc_sample,N);

  /* Add the N ADC samples */
    for (index=X/2; index<N-X/2; index++)
  {
   avg_sample += adc_sample[index];
  }

  /* Compute the average of N-X ADC sample */
  avg_sample /= N-X;

    /* Return average value */
  return avg_sample;
}
```

```
/**
  * @brief   Sort the N ADC samples
  * @param ADC samples to be sorted
  * @param Numbre of ADC samples to be sorted
  * @retval None
  */
void Sort_tab(uint16_t tab[], uint8_t lenght)
{
   uint8_t l=0x00, exchange =0x01;
   uint16_t  tmp=0x00;

 /* Sort tab */
    while(exchange==1)
    {
    exchange=0;
    for(l=0; l<lenght-1; l++)
    {
      if( tab[l] > tab[l+1] )
      {
      tmp = tab[l];
      tab[l] = tab[l+1];
      tab[l+1] = tmp;
      exchange=1;
      }
    }
    }
}
```

Revision history

Table 6. Document revision history

Date	Revision	Changes
19-Mar-2012	1	Initial release.
21-Mar-2012	2	Inserted missing text on page 9.
10-Aug-2012	3	Replaced "STM32F20x, STM32F21x, STM32F40x and STM32F41x microcontrollers" by: "STM32F20x / STM32F21x (revZ, revY, revX) and STM32F40x/STM32F41x (revZ) microcontrollers" in the *Introduction*. Added *Appendix A: Averaging of N ADC samples: source code* and *Appendix B: Averaging of N-X ADC samples: source code*, and corresponding notes in *Section 2.1.1* and *Section 2.1.2*.