

Tunku Abdul Rahman University College

BAME 2123

Microcontroller Peripherals

***Analogue Digital Converter***

|  |  |  |
| --- | --- | --- |
| Name: | Foong Ghin Yew |  |
| ID: | 15 WAU 09184 |  |
|  |  |  |
| Programme: | RMB 2 |  |
|  |  |  |
| Tutor: | Dr. Poh Tze Ven |  |

TABLE OF CONTENTS

[LIST OF FIGURE i](#_Toc439016694)

[LIST OF TABLE ii](#_Toc439016695)

[CHAPTER 1: INTRODUCTION 1](#_Toc439016696)

[Objective: 1](#_Toc439016697)

[Brief Background: 1](#_Toc439016698)

[CHAPTER 2: METHODOLOGY 3](#_Toc439016699)

[Information Grabble: 3](#_Toc439016700)

[Software Coding: 5](#_Toc439016701)

[Enabling the peripherals: 5](#_Toc439016702)

[Channel Selection & Setting: 5](#_Toc439016703)

[Conversion Setting: 6](#_Toc439016704)

[Retrieval of Data: 7](#_Toc439016705)

[Hardware Connection & Test: 8](#_Toc439016706)

[Hardware Connection: 8](#_Toc439016707)

[Test to Run: 8](#_Toc439016708)

[CHAPTER 3: RESULT 10](#_Toc439016709)

[Result & Discussion: 10](#_Toc439016710)

[Resolution & Pooling Data Retrieval Method Tests: 10](#_Toc439016711)

[Sequence, Continuous and Interrupt of Regular Group Tests: 12](#_Toc439016712)

[Sequence, Continuous and Interrupt of Injected Group Test: 18](#_Toc439016713)

[DMA Interrupts Data Retrieval in Injected Group Tests: 20](#_Toc439016714)

[Multi ADC Mode Test: 21](#_Toc439016715)

[CHAPTER 4: CONCLUSION 26](#_Toc439016716)

[REFERENCE 28](#_Toc439016717)

[APPENDIXES GENERAL I](#_Toc439016718)

[APPENDIXES MAIN IV](#_Toc439016719)

[APPENDIXES ADC V](#_Toc439016720)

# LIST OF FIGURE

[Figure 1 Sampling of Analogue Signal once 1](#_Toc439016721)

[Figure 2 Sampling the signal multiple times or multiple conversion 2](#_Toc439016722)

[Figure 3 Potential Divider circuit 8](#_Toc439016723)

[Figure 4 Potential Divider circuit to prepare input voltage of 1V, 1.5V and 2V 9](#_Toc439016724)

[Figure 5 Average input voltage of 1.49V read from oscilloscope 10](#_Toc439016725)

[Figure 6 12-bit resolution conversion (1.5V) 11](#_Toc439016726)

[Figure 7 10-bit resolution conversion 11](#_Toc439016727)

[Figure 8 8-bit resolution conversion 12](#_Toc439016728)

[Figure 9 6-bit resolution conversion 12](#_Toc439016729)

[Figure 10 0.99V DC input to channel 0 13](#_Toc439016730)

[Figure 11 1.49V DC input to channel 3 13](#_Toc439016731)

[Figure 12 1.99V DC input to channel 6 14](#_Toc439016732)

[Figure 13 Set CONT: Data of 1st conversion 15](#_Toc439016733)

[Figure 14 Set CONT: Data of 2nd conversion 15](#_Toc439016734)

[Figure 15 Set CONT: Data of 3rd conversion 15](#_Toc439016735)

[Figure 16 Set CONT without SCAN: Data obtained using DMA 16](#_Toc439016736)

[Figure 17 Data obtained when both SCAN and CONT is set 17](#_Toc439016737)

[Figure 18 Data obtained when SCAN is set but CONT is reset 17](#_Toc439016738)

[Figure 19 Injected Group Conversion triggered with JSWSTART 18](#_Toc439016739)

[Figure 20 Injected Group Conversion triggered with SWSTART 20](#_Toc439016740)

[Figure 21 Data read using DMA transfer from DR in ADC to SRAM 20](#_Toc439016741)

[Figure 22 DMA able to work individually with different ADC at the same time 22](#_Toc439016742)

[Figure 23 Value read in CDR during Multi-ADC mode 23](#_Toc439016743)

[Figure 24 Reading CDR in ADC Interrupt 23](#_Toc439016744)

[Figure 25 Reading CDR data from ADC Interrupt 23](#_Toc439016745)

[Figure 26 DMA transferred CDR data to buffer1 24](#_Toc439016746)

[Figure 27 Multi-ADC mode conversion without CONT bits 25](#_Toc439016747)

[Figure 28 Structure of RCC, GPIO, DMA and ADC I](#_Toc439016748)

[Figure 29 Function used to configure the GPIO port to analog pin I](#_Toc439016749)

[Figure 30 Base Address of all the required peripherals II](#_Toc439016750)

[Figure 31 Functions in RCC to unreset pin and enable clock for different peripherals II](#_Toc439016751)

[Figure 32 Declaration of buffer1, buffer2 and buffer3 and function to configure DMA2 for ADC1. III](#_Toc439016752)

[Figure 33 The Interrupt Handler III](#_Toc439016753)

# LIST OF TABLE

[Table 1 GPIO pin and ADC input channel matching table 4](#_Toc439016754)

[Table 2 DMA2 stream channel and ADC channel matching table 4](#_Toc439016755)

[Table 3 Resolution selection result 10](#_Toc439016756)

[Table 4 Table of expected conversion sequence and actual result 12](#_Toc439016757)

[Table 5 Table of results when SCAN and CONT is set high 17](#_Toc439016758)

[Table 6 Table of Injected Conversion result triggered by JSWSTART 19](#_Toc439016759)

[Table 7 Table of Injected conversion result triggered by SWSTART 20](#_Toc439016760)

[Table 8 Expected result and actual result read from SRAM 21](#_Toc439016761)

[Table 9 Data read from CDR in hex and decimal 24](#_Toc439016762)

[Table 10 Table showing two value in 'buffer1' together 24](#_Toc439016763)

# CHAPTER 1: INTRODUCTION

## Objective:

1.1 To study the theory of Analogue Digital Converter (ADC).

1.2 To implement ADC on micro-controller (stm32f429) and investigate the different configuration of the ADC.

## Brief Background:

Analogue Digital Converter (ADC) is a device that use to convert an analogue input signals to digital data. Electronics world required ADC to record an analogue signal into the memory and convert back to the analogue signal when needed. All the electronics memory storage available now can only store digital data and therefore, the only way to store an analogue signal is to convert the signal into digital data and store the converted digital data [3].

Basically, ADC is a voltage comparator that compared the input signal voltage with a set of voltage range in the ADC. ADC compare and fined the voltage range that the input signal voltage drop. When the voltage range that the input voltage level is found, the digital data of that represent that range will be stored into the memory. This process of comparing the voltage is considered as one conversion which means that the signal is converted once as shown in Figure 1 below. When the conversion number is large enough, a long analogue signal can be save into memory as shown in Figure 2.

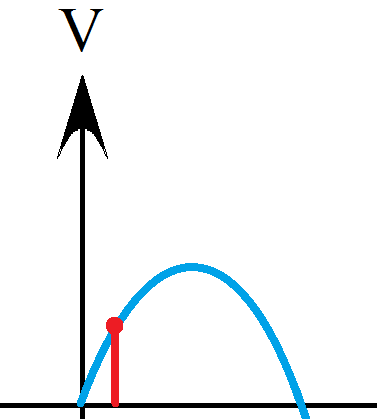


Figure 1 Sampling of Analogue Signal once

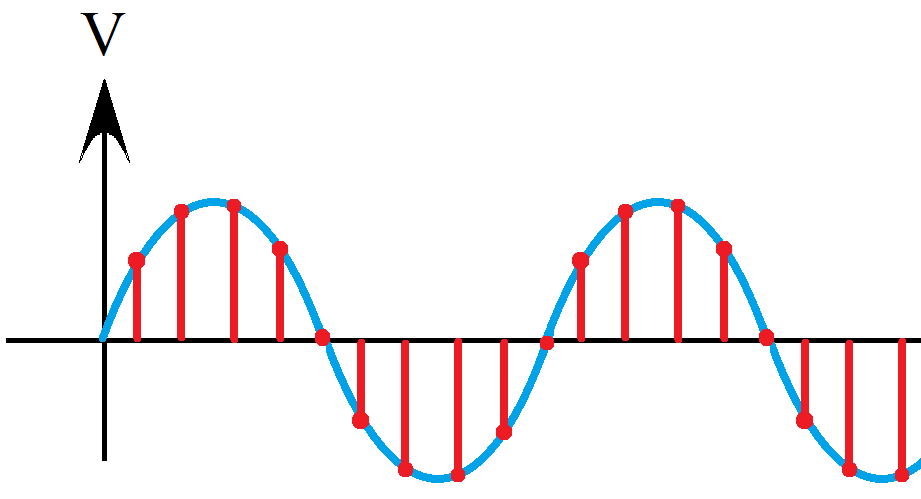


Figure 2 Sampling the signal multiple times or multiple conversion

In Figure 1 and Figure 2, the blue color sine wave is the input signal whereas the red color lines are the converted data or a sample. During the voltage comparison process, the input signal does not compare with the predefine voltage range set directly. The input voltage have to charge up an internal capacitor and compared the capacitor voltage with the predefine voltage range set because the input signal is continuously changing and therefore is not a good candidate to compare.

In ADC, the time taken to charge up the capacitor until the voltage is stable is called the sampling time [1] *(Section 1.16)*. However, the time between two samples is not the sampling time, the time between two samples is so called the conversion time which is the whole process from charging up the internal capacitor to finished convert of voltage to digital data. Sampling time is just a part of the conversion time [4] *(Section 13.5)*. The word “resolution” in ADC defined how accurate the digital data converted from the input signal, meaning that how many voltage level that can be chosen to represent the input signal.

For current trend, ADC is normally placed on a micro-controller as a peripheral instead of a single ADC chip. As micro-controller come together with many peripherals, making the ADC easier to communicate with others peripherals. To implement the ADC on a micro-controller, we need to consider the clock, the input channels, the conversion setting and the data retrieval. In this experiment, ADC with different configuration will be tested using the micro-controller STM32F429ZI.

# CHAPTER 2: METHODOLOGY

This chapter divided into three parts that describe the process on conducting the experiment. First part is the Information Grabble that describe the required information obtaining from the datasheet of the microcontroller. Second part describe the Software Coding used to execute the ADC in stm32f429. The third part is the hardware setting that describe the hardware connection used to do the tests and the procedure to do the test.

## Information Grabble:

There are three ADC peripheral in the micro-controller that have the totally same function as each other in micro-controller STM32f429ZI. Thus, is possible to have three conversion of analogue signal occur at the same time. However, the datasheet stated that the same channel should NOT go under conversion by different ADC at the same time. Another thing to take note here is that the input voltage must not exceed 3V as it may burn the ADC chip in the STM32F429ZI board. The following list listed the things required on working the ADC.

1. Setting RCC to enable clock for required peripherals
2. Setting the GPIO pin as ADC input
3. Setting the ADC characteristic
   * 1. Selection of Channel
     2. Resolution
     3. Sampling time
     4. Continuous or non-continuous
4. Setting the DMA for ADC data retrieval

Everything in digital need clock to run, therefore the first thing required is to **enable the clock** for the peripherals to be use. In this experiment, ADC is not the only peripheral needed but also GPIO and DMA. GPIO is needed to receive the input analogue signal to the micro-controller that is going to perform the conversion by ADC. Whereas, DMA is not a “must” in implementing an ADC but this peripheral was used in this experiment for the purpose of data retrieval and data transfer.

GPIO pin that is going to use as the input of the ADC channel must be set as analogue pin. The available ADC channel input matching with the GPIO ports. Information about GPIO input pin that is available for ADC channel are tabulated in Table 1 below. Note that A0 in the table mean GPIO port **A** pin **0** [2].

Table 1 GPIO pin and ADC input channel matching table

|  |  |  |  |
| --- | --- | --- | --- |
| **ADC Channels** | **ADC 1** | **ADC 2** | **ADC 3** |
| ADC\_IN0 | A0 | A0 | A0 |
| ADC\_IN1 | A1 | A1 | A1 |
| ADC\_IN2 | A2 | A2 | A2 |
| ADC\_IN3 | A3 | A3 | A3 |
| ADC\_IN4 | A4 | A4 | F6 |
| ADC\_IN5 | A5 | A5 | F7 |
| ADC\_IN6 | A6 | A6 | F8 |
| ADC\_IN7 | A7 | A7 | F9 |
| ADC\_IN8 | B0 | B0 | F10 |
| ADC\_IN9 | B1 | B1 | F3 |
| ADC\_IN10 | C0 | C0 | C0 |
| ADC\_IN11 | C1 | C1 | C1 |
| ADC\_IN12 | C2 | B2 | C2 |
| ADC\_IN13 | C3 | B3 | C3 |
| ADC\_IN14 | C4 | C4 | C4 |
| ADC\_IN15 | C5 | C5 | C5 |

There were two available Direct Memory Access (DMA) channel for each ADC that was listed in different stream or channel in the DMA2. The ADC channel and DMA stream channel matching should be configured as shown in Table 2 below. Blank spaces in the table is reserved for the use for other peripherals. Other channels and streams that is not listed here means that those channels and streams cannot be applied for ADC usage.

Table 2 DMA2 stream channel and ADC channel matching table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Stream** | **0** | **1** | **2** | **3** | **4** |
| **Channel 0** | ADC1 |  |  |  | ADC1 |
| **Channel 1** |  |  | ADC2 | ADC2 |  |
| **Channel 2** | ADC3 | ADC3 |  |  |  |

## Software Coding:

For the microcontroller STM32F429ZI, there are three type of registers including Status Register (SR), Control Register (CR) and Data Register (DR) that use flag to indicate the current status, control and setting performance of the peripherals and storing the data respectively. This part describe the way to configure these three type of registers. Important part of the code can be reviewed in the Appendixes. Full code can be reference to <https://github.com/EmeraldWolfram/ADC_CoIDE>.

### Enabling the peripherals:

In STM32F429ZI, there are two control bits that have to be configured to enable a peripherals. First, un-reset all the pins of the particular peripherals and second is to enable the clock for the peripherals. Both control bits can be found and configured in the registers RSTR and ENR of the RCC section. For the case of ADC, the input clock is APB2. Therefore, setting a low to the ADC control bit in APB2RSTR control register can un-reset the pin and setting a high to the ADC control bit in APB2ENR control register can enable the clock for the ADC. Refers to the datasheet to determine which control register in the RCC section that store the RSTR and ENR of that peripheral, then un-reset the pin and enable the clock. As a special case to ADC peripheral, ADC required to be awaken before it can use by setting a high to ADON in control register CR1 of the ADC section.

### Channel Selection & Setting:

There are total of 16 available GPIO external channel and 3 available internal channel for the ADC. From Table 1 in previous section, select the channel wanted to use and configure the specific GPIO pins to analogue and no pull-up or pull-down. For internal channel, configuration for GPIO is not required but ADC section still required configuration. In this experiment, channel used are ADC\_IN0, ADC\_IN3, ADC\_IN6 and the internal channel VBAT.

The ADC of STM32F429ZI can be divided into two group of channels to undergo conversion, the regular group and the injected group. The regular group can have maximum of 16 member of channel in it and conversion will run from the 1st channel until the 16th. Whereas, the injected group can only have maximum of 4 member of channel. The different between this two groups is that Injected Group consist of four separated data registers to store the data converted from the 4 member in the injected group. However, the regular group will have to share a single data register. Therefore, the code must make sure that the data converted can be retrieved in time before the next converted data replace the current data in the regular group’s data register.

The two groups of channels used the registers SQR and JSQR in the ADC section to queue the member of channels for the regular group and injected group respectively. In both registers, there are L bits that define the number of channels that will undergo conversion. Next, place the channel number (0 to 18) into the queue starting from SQR3 to SQR1 for regular group. Whereas, JSQR is a single 32 bit register and was enough to store the queue that can have a maximum of 4 channels queued.

### Conversion Setting:

The **main section** that describe the test experimented in this experiment will be in this section which setup different configuration setting for the ADC. Setting the SWSTART bit in the ADC’s control register CR2 trigger the conversion of regular group to start and JSWSTART for injected group. The resolution of the result can be set at the two RES bits in the control register CR1 that represent the converted data in 6-bit, 8-bit, 10-bit and 12bit.

Next, setting the CONT bit in the control register CR2 to high enable the ADC to continuously perform conversion of input analogue signal channel to channel one by one without setting a new trigger to the conversion. However, this only enable the regular group to continuously perform A-to-D conversion. Doing the same thing to Injected Group by setting JAUTO bit in CR1 to high.

Three ADC (ADC1, ADC2 and ADC3) can work together by setting multi-mode in the control register CCR in the Common ADC section. SWSTART in the ADC1 will be the trigger to start the conversions during multi-mode. ADC2 and ADC3 will follow the trigger of the ADC1. Please note that every control register or control bit that have the alphabet ’J’ in front refers to the injected group and the one without the ‘J’ refers to the regular group.

### Retrieval of Data:

There are three way to obtain the data converted by the ADC. First way is the easiest pooling method that keep on checking the status register in a loop and read the data when the status flag indicate the data is ready.

The second way is using the ADC interrupts method. ADC can generate an interrupt when the conversion is finished by enabling the EOC and JEOC interrupts for regular and injected group respectively. When the interrupt flag raised, the program will jump to the Interrupt Service Routine (ISR), then code written in the ISR retrieve the data and clear the flag.

The third way is using the DMA interrupts method. DMA automatically read the data from the source address assigned and transfer to the SRAM that was assigned. According to Table 2, the DMA for the particular stream and channel is configured to peripheral-to-memory mode such that the source address is the data register DR of the ADC. The source address does not increment as the address of the DR register does not change but the destination memory should increment to store the data of the different conversion.

## Hardware Connection & Test:

### Hardware Connection:

After all the software code was written, the next step to test would be setting up the hardware. The hardware were set up as follow to provide a DC value as the input value to the micro-controller as shown in Figure 3. Input voltage is varies by changing the resistor R2. The calculation of the voltage is easy using this equation, *Input Voltage = {3 × R2 ÷ (R1 + R2)} V.*

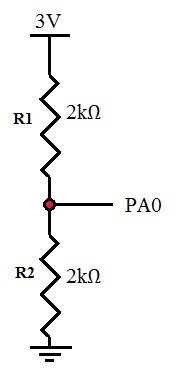


Figure 3 Potential Divider circuit

After setting up all the required software coding and hardware wire connection, debugging feature provided by software CoIDE is used to obtain the result by reading the registers throughout the execution of the code written.

### Test to Run:

Before starting the test, note that all the GPIO port that will be used must be configured to analogue, clock enabled and pin un-reset. Starting from testing different resolution with a 1.5V input voltage by comparing the calculated data with the converted data. The calculation of data can be calculated using the equation, *Converted Data = 2n × (Voltage read from oscilloscope)V ÷ 3V.* The ‘n’ in the equation represent the number of bit of the converted data. The converted data is then obtained using the pooling method as stated earlier. At the same time single conversion mode will be tested as only one conversion were done each time.

Next, sequence of conversion of channels and the continuous conversion mode will be tested together. First, enable the GPIO pin for the ADC input channel 0, channel 3 and channel 6. Set the resolution to 8 bit and queue them in the SQR registers in the sequence of channel 0 then channel 3 and finally channel 6. Connect input voltage of 1.0V to channel 0, 1.5V to channel 3 and 2.0V to channel 6 as shown in Figure 4. Then start the conversion and compare the 1st converted data, 2nd converted data and 3rd converted data with the calculated converted data. This test will then use ADC interrupts to obtain the result. However, due to some technical problem which will be explain in the next chapter, the test then use the DMA to retrieve data.

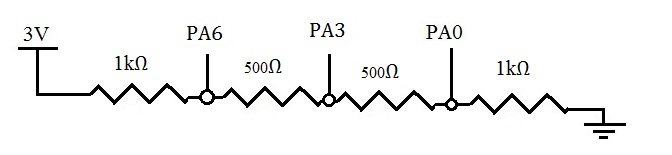


Figure 4 Potential Divider circuit to prepare input voltage of 1V, 1.5V and 2V

Sequence of conversion for Injected Group will then be tested by changing the sequence from the previous test in regular group to the injected group. JAUTO bit will then be set high to see if there is changes. Note that this tests used ADC interrupts to retrieve the data instead of pooling as a test to the implementation of interrupts and data registers allocated for the Injected Group. DMA is not used because the this test wanted to show the advantage of Injected Group over Regular Group.

Final experiment will be conducted to investigate the multi-mode of the ADC by enabling all ADC1, ADC2 and ADC3. ADC1 will have channel 6 queued in the regular group, ADC2 channel 3 in the regular group and ADC3 have the channel 0 in the regular group. All the three ADC will first run individually instead of enabling the Multi Mode. All ADC data is transferred to SRAM using DMA. After that, setting DMA mode 1 set to Triple ADC regular simultaneous mode start the conversion using ADC3, ADC2 and ADC1.

### Extension test:

The usage of the JOFFSETx registers was tested using the ADC interrupt data retrieval method. The test on JOFFSET1 register was conducted in two parts where first part check within the normal working boundary. Whereas, part two check the data when the offset value was larger than the input signal value. Channel 0 that was connected to a 1.5V DC input signal was added into the Injected Group to check the differences.

# CHAPTER 3: RESULT

This chapter express and discuss the results of tests obtained from the previous chapter. Test tested on the ADCs in the micro-controller include resolution setting test, pooling data retrieval method, Continuous Mode on regular group, Continuous Mode on injected groups, ADC interrupts data retrieval method, single ADC DMA data retrieval method and Multi ADC mode.

## Result & Discussion:

### Resolution & Pooling Data Retrieval Method Tests:

The voltage read from the oscilloscope is averaged to 1.49 V as shown in Figure 5 below. By using the average voltage 1.49V read from the oscilloscope, the expected conversion data were calculated and tabulated together with the data converted by the ADC. The result of converted data from the ADC were shown in Figure 6, 7, 8 and 9.

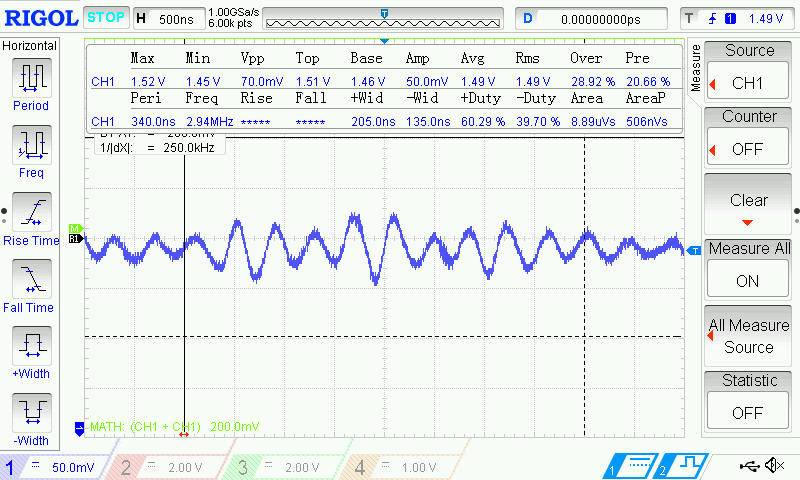


Figure 5 Average input voltage of 1.49V read from oscilloscope

Table 3 Resolution selection result

|  |  |  |
| --- | --- | --- |
| **Resolution, RES[1:0]** | **Calculated Data, decimal** | **Converted Data, decimal** |
| 00 (n = 6 bits) | 32 | 31 |
| 01 (n = 8 bits) | 127 | 127 |
| 10 (n = 10 bits) | 509 | 509 |
| 11 (n = 12 bits) | 2034 | 2037 |

From Table 3 above, the result shows that the calculated data is totally the same for the 10 bit resolution and 8 bit resolution case. Whereas, the result for the 12 bit resolution and 6 bit resolution is nearly the same as the converted data. This happens as the DC input voltage to the ADC have a minor fluctuation due to noise as shown the Figure 6 and Figure 9 below. If the input voltage is stable without noise in it, the data converted would be much more precise. The data retrieved for the 12-bit resolution have a different of 3 digit between the calculated and actual data showing that a higher resolution can measure a minor noise that is not seen while sampling in 10-bit or lower resolution.

Pooling method is used to retrieve the data from the DR by checking the status in a loop. This method waste a lot of processing power the processor continuously checking the status flag. However, pooling is a simple to implement as Interrupt and Interrupt Handler does not need to be configured.

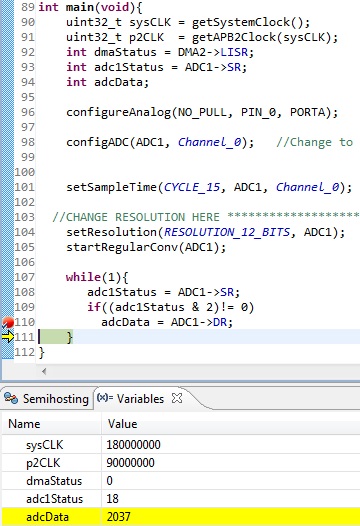
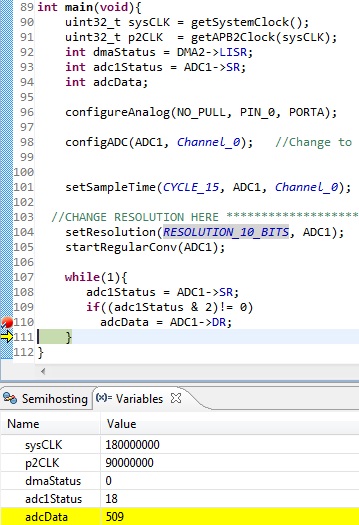


Figure 6 12-bit resolution conversion (1.5V)

Figure 7 10-bit resolution conversion

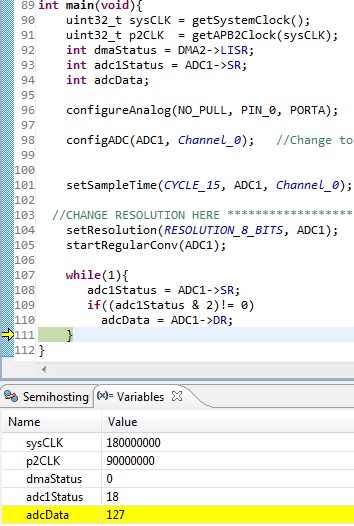
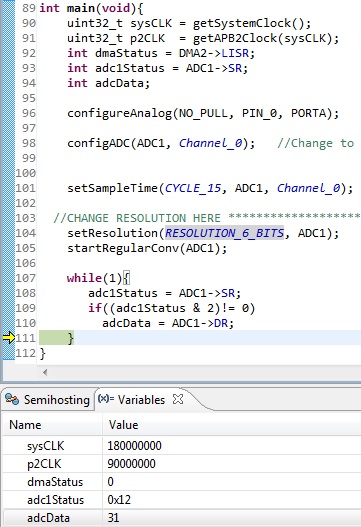


Figure 8 8-bit resolution conversion

Figure 9 6-bit resolution conversion

### Sequence, Continuous and Interrupt of Regular Group Tests:

The regular group of ADC1 is queued as **channel 0 → channel 3 → channel 6** where the channels input is connected according to Figure 4 earlier. The voltage drop on each point were measured using an oscilloscope. Figure 10, 11 and 12 shows the voltage level read from the oscilloscope. Expected data were calculated in sequence of 85, 127 and 170 for the voltage in 8 bit resolution of 0.99V, 1.49V and 1.99V respectively. After queued this channel sequence in registers SQR, the CONT bit in CR2 is set high to select continuous mode.

Table 4 Table of expected conversion sequence and actual result

|  |  |  |
| --- | --- | --- |
| **Conversion Sequence** | **Expected Data, decimal** | **Converted Data, decimal** |
| 1st (Channel 0) | 85 | 85 |
| 2nd (Channel 3) | 127 | 85 |
| 3rd (Channel 6) | 170 | 84 |

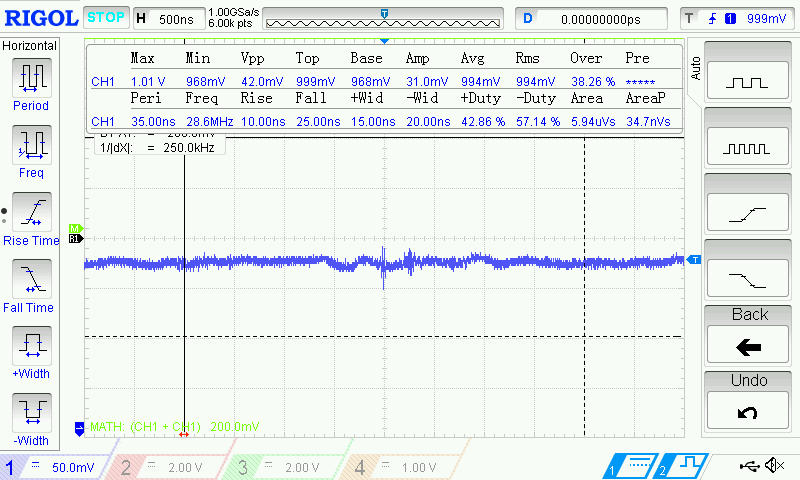


Figure 10 0.99V DC input to channel 0

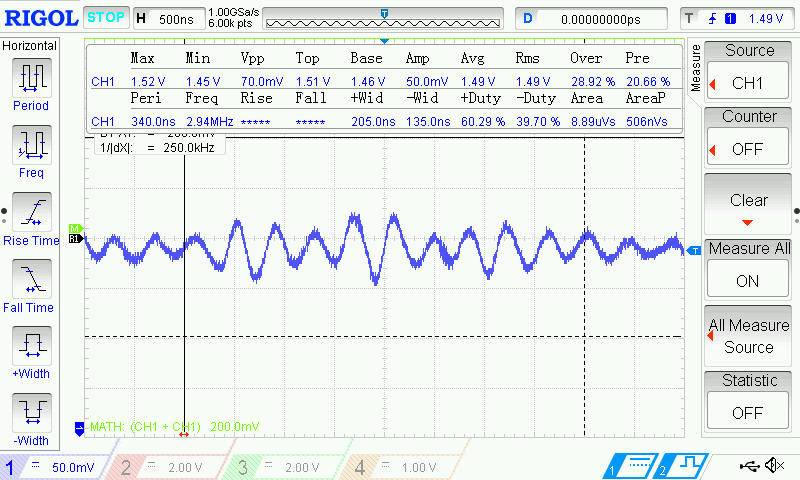


Figure 11 1.49V DC input to channel 3

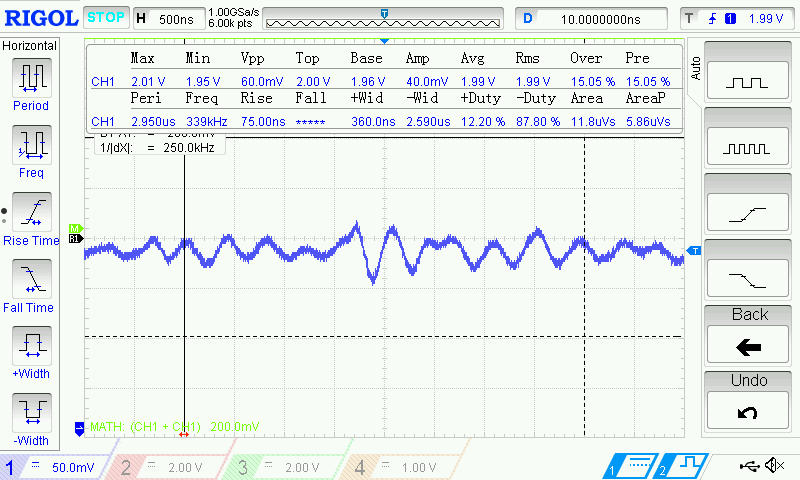


Figure 12 1.99V DC input to channel 6

The converted data is not as expected and maintain around the value 85 showing that the ADC is continuously converting a 0.99V input voltage which is the channel 0. This result suspect that setting the CONT bit in the ADC will continuously convert the 1st channel in the regular group only. It will notconvert the 2nd and 3rd queued channel with setting CONT bit high only as shown in Figure 13, 14 and 15 below.

The original test expected to use the ‘count’ in the code to count the times of entering the ADC Interrupt Service Routine (ISR) that is use to handle the interrupts generated from the ADC. Count is zero meaning that it was the first time entering the ISR and so on for count larger than zero. However, the converted data is not as expected, another suspicion is that the ADC converted multiple time in a single debugger step and luckily every time the step it read when the ADC just finished conversion for the channel 0.

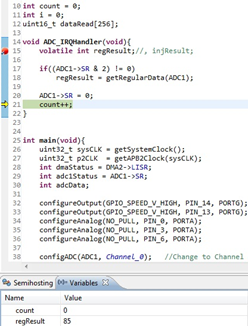
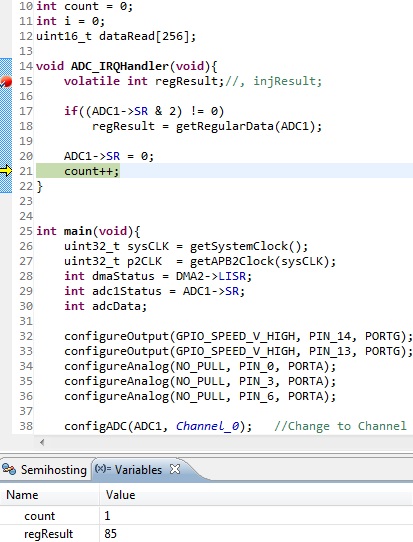


Figure 13 Set CONT: Data of 1st conversion

Figure 14 Set CONT: Data of 2nd conversion

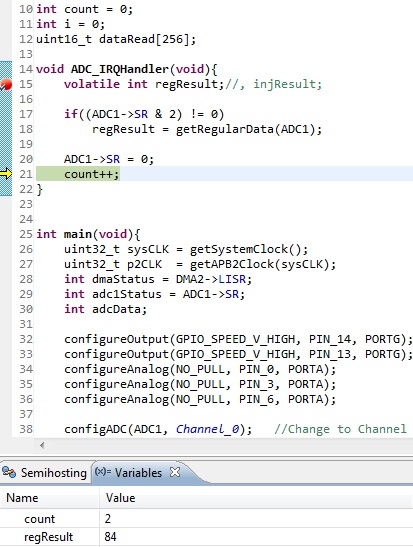


Figure 15 Set CONT: Data of 3rd conversion

Due to the first inspection that the ADC is constantly converting channel 0, the SCAN bit in the CR1 register of the ADC section is set to high, the converted data now changed each time entering the Interrupt Handler but the data obtained does not follow the sequence as expected also. This lead to the second inspection that the STM32F429 microcontroller perform the ADC conversion faster than the debugger itself. Therefore, within a single step in the debugger, the ADC in the hardware might already converted for many times.

As to obtain the data in sequence, DMA is implemented to check the sequence. At first, the DMA for only CONT bit set to high but SCAN bit reset is tested as it is known that the data will be constantly checking for channel 0. The source address (PAR) is set to the Data Register of the ADC1 and the destination address (M0AR) is set to the address of an array of integer called ‘buffer1’. DMA configuration will be further discussed in next section as the main objective in this section is to test the SCAN bit and CONT bit. After getting the DMA working, the result for only the CONT bit set high is obtained as shown in Figure 16 below. From Figure 16, inspection 1 and 2 was proven as the reading are all around 84 and 85 and the first entry to the Interrupt Handler in the debugger step already generated a list of converted data in the buffer1 array.

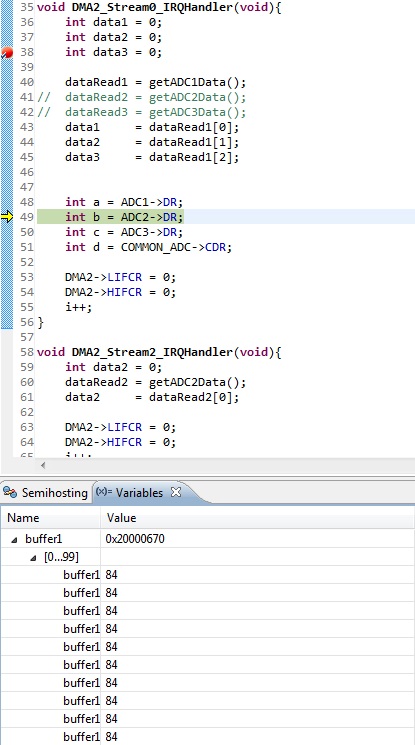


Figure 16 Set CONT without SCAN: Data obtained using DMA

After confirming that the problem of the incorrect sequence of data obtained, the SCAN bit and CONT bit is now set high together to test the result. With DMA transfer, the system obtained the data following the sequence like the expected data as shown in Table 5 and Figure 17. This shows that the ADC will perform conversion for every channels in the regular group by setting the SCAN bit high. After the third entry to the Interrupt Handler or all the channels in the regular queue is converted, the system again start to convert the analogue signal from the channel 0. However, when the CONT bit is reset, the system never enter the Interrupt Handler again after finishing the three conversion as shown in Figure 18.

Table 5 Table of results when SCAN and CONT is set high

|  |  |  |
| --- | --- | --- |
| **Conversion Sequence** | **Expected Sequence, decimal** | **Converted Data, decimal** |
| 1st (Channel 0) | 84 (0.98V) | 84 |
| 2nd (Channel 3) | 126 (1.48V) | 125 |
| 3rd (Channel 6) | 166 (1.95V) | 165 |

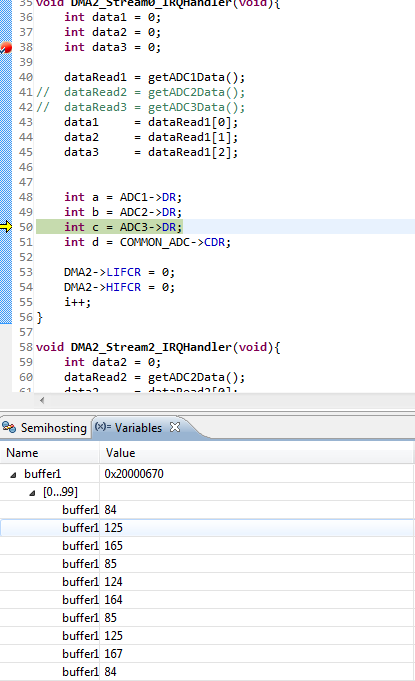


Figure 17 Data obtained when both SCAN and CONT is set

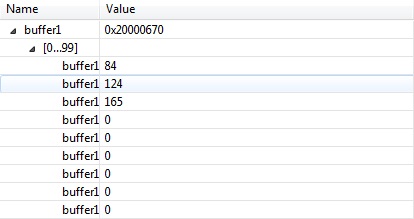


Figure 18 Data obtained when SCAN is set but CONT is reset

### Sequence, Continuous and Interrupt of Injected Group Test:

This test used the same circuit connection in Figure 4. Therefore, the calculated data were same as the previous test on regular group as the reading is the same from the oscilloscope. The only different is the sequence of channels changed to **channel 6 → channel 3 → channel 0** and now queued in the Injected Group. The test begin with setting JAUTO and SCAN bit high but CONT bit low. The injected conversion then started by setting JSWSTART bit high.

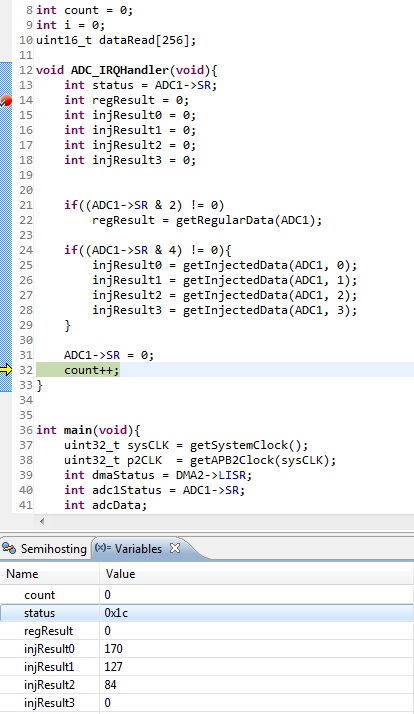


Figure 19 Injected Group Conversion triggered with JSWSTART

On the 1st attempt of JEOC Interrupts, the system entered the ADC ISR with the complete conversion of all the three channels as shown in Figure 19 above. The result was tabulated in Table 6 below. This shows that during SCAN mode, all the channels queued in the Injected Group were all converted and stored in its respective JDR (Injected Data Register). Therefore, all the data were ready to be read when entered the Interrupt Handler according to sequence as the first converted data in the queue will be stored in the first JDR. However, it is not possible to do this for the regular group because there is only one DR (Data Register) for the Regular Group. One advantage of Injected Group here is that the sequence of the conversion can be read according to its respective Data Register instead of guessing what reading currently show.

Advantage of the Injected Group over the Regular Group as each member in the Injected Group have its own data register. The system never enter the ADC ISR anymore after the first entry. This mean that the system does not start the A-to-D conversion again as expected because the CONT bit is set low. From Figure 19 above, the third register ‘injResult3’ that copy data in JDR4 read a 0 because there are only 3 channels queued in the Injected Group which used JDR1, JDR2 and JDR3. This result prove that the new sequence starting from channel 6 again will not store the converted value in JDR4 but start from JDR1 again meaning that the register is really assigned to the respective queued instead of following the sequence.

Table 6 Table of Injected Conversion result triggered by JSWSTART

|  |  |  |
| --- | --- | --- |
| **Conversion Channel** | **Expected Data, decimal** | **Converted Data, decimal** |
| JDR1 (Channel 6) | 170 | 170 |
| JDR2 (Channel 3) | 127 | 127 |
| JDR3 (Channel 0) | 85 | 84 |
| JDR4 | 0 | 0 |
| DR (Channel 6 in regular) | 0 | 0 |

Next, the test used the SWSTART bit (trigger bit for Regular Group) to start the conversion instead of the JSWSTART bit to start it. The system work almost the same as triggering the conversion using the JWSTART to the Injected Group. The only different is that a regular data was read also as shown in Figure 20 below. In the perspective of Injected Group only, using the trigger of SWSTART or JSWSTART is the same. Therefore, the JAUTO is concluded to cast the Injected Group to behave like a Regular Group that could be triggered by SWSTART but with an advantages of more data register.

Another interesting result was found in Figure 20 as both the 1st channel queued in Regular Group and the 1st Channel’s in Injected Group is channel 6 but the result obtained is different with 170 and 169 respectively. This shows that the ADC actually sample the channel at different time for the Regular Group and the Injected Group. From here, a usage of having two group in the ADC is found where two data can be compared using the injected group and regular group so that user could know there is wrong configuration when the reading in the Injected Group is much different from the Regular Group.

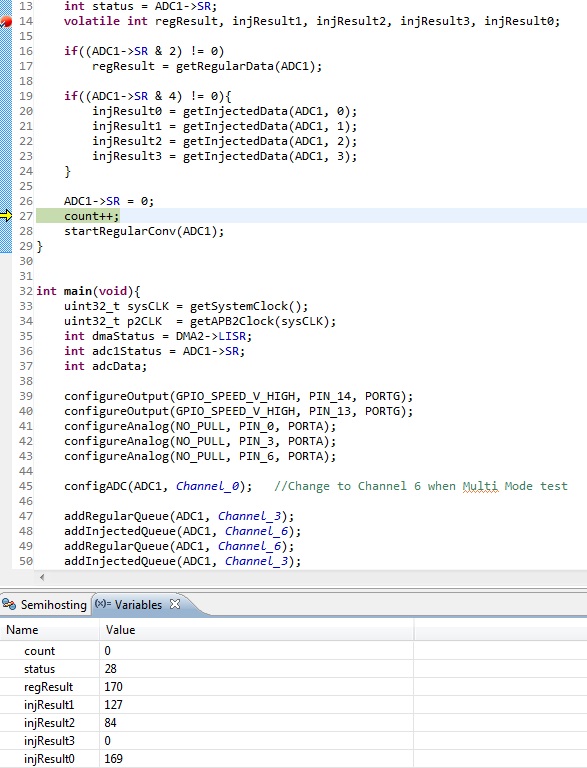


Figure 20 Injected Group Conversion triggered with SWSTART

Table 7 Table of Injected conversion result triggered by SWSTART

|  |  |  |
| --- | --- | --- |
| **Conversion Sequence** | **Calculated Data, decimal** | **Converted Data, decimal** |
| Channel 6 (Injected) | 170 | **169** |
| Channel 3 (Injected) | 127 | 127 |
| Channel 0 (Injected) | 85 | 84 |
| Channel 6 (Regular) | 170 | **170** |

### DMA Interrupts Data Retrieval in Injected Group Tests:

As promised earlier, the DMA will be discuss further in this section. The ADC configuration for DMA test is just like the previous test on Injected Group where the JAUTO bit and SCAN bit was set high. For the DMA, first enable the clock at control register AHB1ENR and un-reset the pin through the control register AHB1RSTR in the RCC section. Then, the DMA is configured in **Direct Mode** with **DMA control flow** and **peripheral-to-memory** mode. The source address PAR is set to the address of the JDR1 in ADC1. The destination memory M0AR is set to the address of an integer array buffer and both the PAR and M0AR were set to be incrementing in **one word size** and **half-word size** respectively.

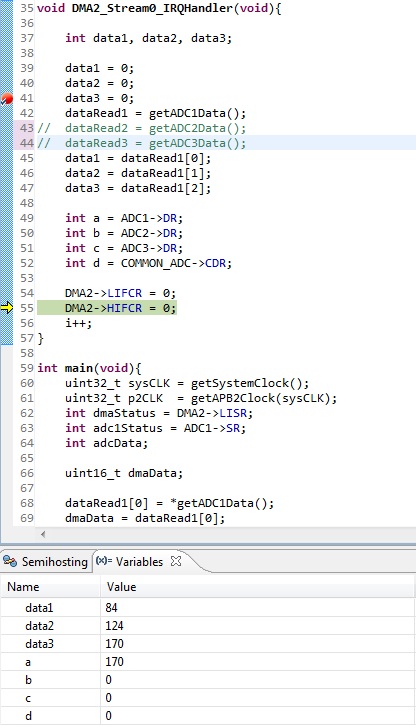


Figure 21 Data read using DMA transfer from DR in ADC to SRAM

In the ISR, all the data was clear to zero before reading value from the buffer declared to make sure the data reading is not because of luck. When the conversion started, ‘data1’ that point to the 1st address of the buffer read the value 84 which is the conversion of channel 0. ‘data2’ and ‘data3’ that point to the 2nd and 3rd address of the buffer were showing the 127 and 170 that convert for channel 3 and channel 6 respectively. This shows that the DMA can actually obtain the data from the ADC conversion without using the ADC Interrupt Service Routine. This provide a good memory structure when the previous or old data is needed together with the new data as all the data was saved in different memory spaces as the data can be automatically assigned to the buffer 1 by 1. Another advantage of DMA seen in previous section when the experimenter is trying to use a debugger that is slower to read a sequence, DMA is able to solve this problem.

Table 8 Expected result and actual result read from SRAM

|  |  |  |
| --- | --- | --- |
| **Conversion Sequence** | **Calculated Data, decimal** | **Converted Data, decimal** |
| Data1 (buffer[0]) | 84 | 84 |
| Data2 (buffer[1]) | 127 | 124 |
| Data3 (buffer[2]) | 170 | 170 |

### Multi ADC Mode Test:

There are two test run for using multiple ADC at the same time. The first test configured all the ADC with the specification of channel 6, channel 3 and channel 0 in the regular group of the ADC1, ADC2 and ADC3 respectively. After that, DMA2 Stream 0 channel 0, Stream 2 channel 1 and Stream 1 channel 2 are configured to transfer data from ADC1, ADC2 and ADC3 respectively. All the three respective DMA2 interrupt are enabled with DMA control flow. The SWSTART bit in the three ADC were then set high to trigger the ADC conversion.

|  |  |  |
| --- | --- | --- |
| ADC 1 | Channel 6 (1.96V) | 162 - 172 |
| ADC 2 | Channel 3 (1.48V) | 121 - 131 |
| ADC 3 | Channel 0 (0.98V) | 88 - 98 |

Figure 22 shows that the three data from the three ADC were retrieved from the three different buffer in the SRAM. The data is obtained from three different DMA Stream ISR but Figure 22 retrieve all three data in the Stream 0 ISR just to reduce the number of figure required to be displayed in this report.

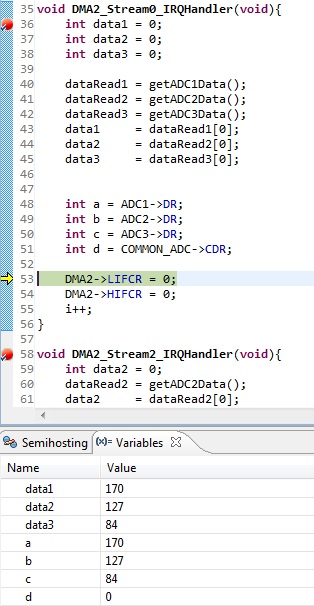


Figure 22 DMA able to work individually with different ADC at the same time

Whatsoever, this test just to prove that all three ADC can work individually at the same time as the data is able to be retrieved from ADC2 and ADC3 and transfer using DMA2. The data can be retrieve through its own DMA Stream as the debugger entered the three configured DMA2 stream ISR during the experiment.

In Figure 22, the integer ‘a’, ‘b’ and ‘c’ read the Regular Data Register of the respective ADC and was exactly the same as the data read from the SRAM allocated to the DMA destination. This prove that the data is truly transferred from their respective destination. Integer ‘d’ that have the value of Common Data Register (CDR) is 0 because the Multi-ADC mode is not configured. All the individual DMA transfer is then disabled to test for the multi-mode ADC DMA transfer.

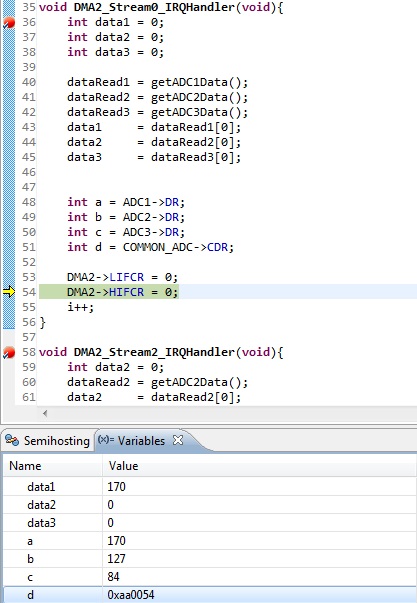


Figure 23 Value read in CDR during Multi-ADC mode

When the Multi-ADC mode is configured to Triple Regular Simultaneous Mode in the Common Control Register (CCR) under the COMMON ADC section, the result were as shown in Figure 23 where the integer ‘d’ now read the value 0x00AA0054. Note that the CONT bit in all the three ADC is set to high. By splitting the hex number into 2 half-word, it was 0xAA and 0x54 which equal to 170 and 84 in decimal. This number were the values in the Data Register of ADC1 and ADC3 respectively as it was the same as integer ‘a’ and ‘c’. The value of the CDR is then obtained as shown in Figure 24 and 25. All the decimal representation of the hex code in Figure 23, 24 and 25 have been converted to decimal and tabulated in Table 9.

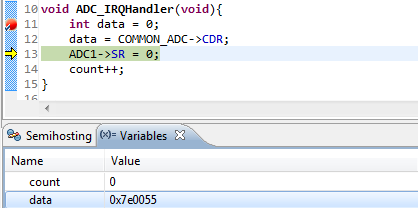


Figure 24 Reading CDR in ADC Interrupt

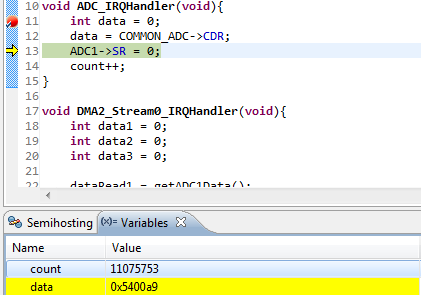


Figure 25 Reading CDR data from ADC Interrupt

Table 9 Data read from CDR in hex and decimal

|  |  |  |
| --- | --- | --- |
| **ADC Conversion** | **Data read in CDR, hex** | **Data converted, decimal** |
| ADC1 | 0xAA | 170 |
| ADC2 | 0x7F | 127 |
| ADC3 | 0x54 | 84 |

When reading the global variable ‘buffer1’ that was declared as 16-bit integer array, the value are as shown in Figure 26 below. By reading two values in Figure 26 in hex and mix it together, the sequence read as Table 10 below. The right column of Table 10 below show the value appearing in hex code and it was found that the value is actually not in the sequence of how the datasheet declare. The first transfer is the channel 0 and channel 6 which is ADC3 & ADC1, meaning that the ADC actually start from ADC3 & ADC1 instead of ADC1 & ADC2. When the debugger is stop and restarted, the sequence does not change. Therefore, it was proven that the datasheet actually didn’t give the correct sequence here as the sequence come with ADC3 & ADC1, ADC1 & ADC2 and finally ADC2 & ADC3.

Table 10 Table showing two value in 'buffer1' together

|  |  |
| --- | --- |
| **Two value read from Figure 26** | **After converted into Hex code** |
| 84 & 170 | 0054 00AA |
| 170 & 127 | 00AA 007F |
| 127 & 84 | 007F 0054 |

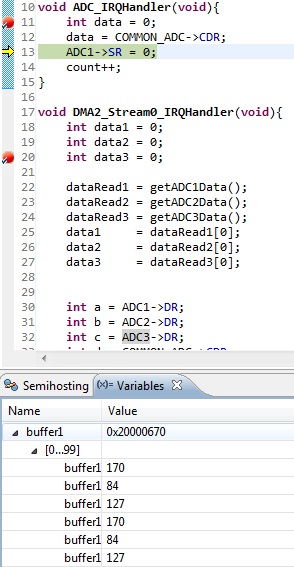


Figure 26 DMA transferred CDR data to buffer1

When all the CONT bit in the three ADC is reset to low, the system conversion halt after converted the ADC1 & ADC3 and did not convert the data in ADC2 or channel 3. The result is shown in Figure 27 below where the conversion only have 169 (ADC1) and 84 (ADC3). The value afterward are all zero showing that it never perform A-to-D conversion again. However, what interesting here is that the system actually convert the data of ADC1 and ADC3 and stopped. At first, the experimenter expect to read the ADC1 only without the value of ADC3. However, the result shows that actually the ADC3 and ADC1 perform the conversion together and that’s why there were two reading. The system know that there are two value to be stored in the CDR register and therefore perform the first two value in the sequence which is ADC1 and ADC3. After the system performed the conversions, the system did not see the CONT bit and therefore halt without converting ADC3 & ADC2.

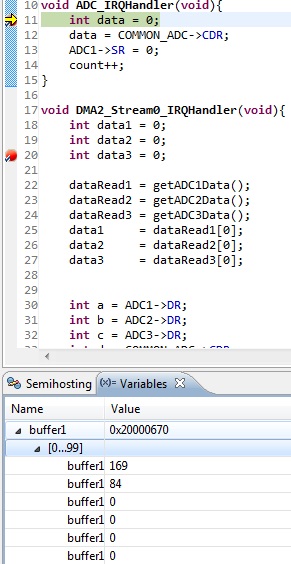


Figure 27 Multi-ADC mode conversion without CONT bits

### Offset Voltage Test:

First part of the offset voltage test take in a 1.5V DC voltage from channel 0. When the interrupt of End Of Injected Conversion (JEOC flag raised), a value of 127 was obtained as shown in Figure 28 below. When the offset value was set using the setAllOffset function with a value of 27, data read 100 instead of 127 which equal to the value of original reading subtracting the offset value as shown in Figure 29.

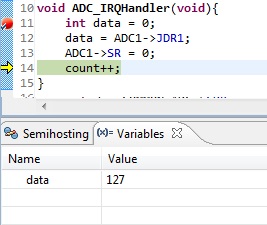


Figure 28 Original value before offset 27

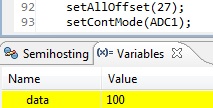


Figure 29 After calling function setAllOffset(27), data read 100 = 127 - 27

The second part of the the offset test offset the injected group by 128 which mean taking the original value and minus 128 (0x80 in hex). The offseted value read 65535 (0xFFF in hex) which indicate a negative 1 as shown in Figure 30 below. Therefore, it was concluded that the JDR register can actually store a signed value. As the resolution was set to 8 bit, the maximum value was actually 255. In the previous experiment, the calculation matching the result are all in unsigned-integer format. Therefore, this experiment also shows that the JDR1 register can not differentiate a negative value and the maximum value when the resolution was 12 bits because the maximum value and negative 1 value both equal to 65535 or 0xFFF. For the resolution below 12-bits, the MSB or bit eleven can be used to indicate the sign as it does not affect the result of the lower resolution. Therefore, the range of an 8-bit resolution can be proven to be -256 to 255 instead of 0 to 255.

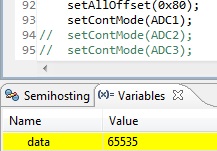


Figure 30 Datae read -1 when offset 128 because 127-128 = -1

# CHAPTER 4: CONCLUSION

Generally, the peripherals that need to be used must have the clock enabled and pin un-reset. Without the clock and un-reset the pin, all the register will always read zero. GPIO ports can be use as inputs to the ADC other than the three internal pin VBAT, VTEMP and VREF. The GPIO pin used must match channel in the ADC and must be configured as analog pin. After setting the GPIO ports, add the channel selected into the SQR or JSQR registers of the ADC and start the conversion. ADC can be divided into two group, Regular Group that can queue up to 16 conversion but only one data register assigned to it and Injected Group that only can queue up to 4 conversion but data register was reserved for each conversion.

There are 3 main control bits that alter the conversion which is the SCAN bit, JAUTO bit and the CONT bit in the Control Register of the ADC. The CONT bit make a choice whether to continuously perform conversion after the first trigger of the SWSTART bit or to wait for another trigger each time a conversion is done. Without the SCAN bit, the ADC will not perform conversion for channel other than the 1st channel in the queue. During SCAN mode, all the channels queued in the Injected Group will be converted before generating an interrupts because there are sufficient Data Register to store the data for each conversion. Whereas, the program will have to generate interrupt for multiple times to retrieve all the channels’ converted data for the Regular Group. When the JAUTO bit is set, Injected Group can be triggered by triggering the Regular Group where setting the SWSTART bit. This make the Injected Group behave like the Regular Group.

There are three method to retrieve data converted by the ADC. Method 1 used the pooling method which used a loop to check the status flag and read the data from Data Register when the status is ready. This method waste a lot of processing power but easy to be implemented. Method 2 is the method of using Interrupts of the ADC. This method enable the program to retrieve the data only when the status flagged. Processing power is saved but there might have chances that the data converted is lost due to insufficient of time to retrieve the data. DMA is the third method that transfer the data from a source address to a destination address. By setting the destination address to increment, the old data will be keep for much longer time giving the processor more time read the converted data.

Three ADC can be used at the same time without using the Multi-ADC mode but three different DMA interrupts will be needed if DMA data retrieval method is used. When Multi-ADC mode is used, the CONT bit in all the three ADC have to be set high or it will halt there and wait for another trigger. The data converted in the Multi-ADC mode will be stored in the CDR register of the COMMON ADC in the sequence of ADC3 & ADC1, ADC1 & ADC2 and finally ADC2 & ADC3.

# REFERENCE

[1] Valiton, R 2013, ‘Sample and Hold Time’, *Understanding ADC parameters*, vol 1. no. 1, pp. 16 – 18.

[2] TILZOR 2014*, Library 06- AD Converter on STM32F4*, viewed 10th December 2015,

<<http://stm32f4-discovery.com/2014/04/library-06-ad-converter-on-stm32f4xx/>>

[3] Kester, W 2005, ‘Fundamentals of Sampled Data Systems’, *Analog-Digital Conversion*, vol. 2, no. 2, pp. 2.21 – 2.27.

[4] ST 2015, ‘Analog-to-digital converter’, *Reference Manual 0090*, vol. 2015, no. 1, viewed 28th October 2015,

<http://www.st.com/stonline/stappl/resourceSelector/app?page=fullResourceSelector&doctype=datasheet&LineID=1806>

# APPENDIXES GENERAL



Figure 31 Structure of RCC, GPIO, DMA and ADC

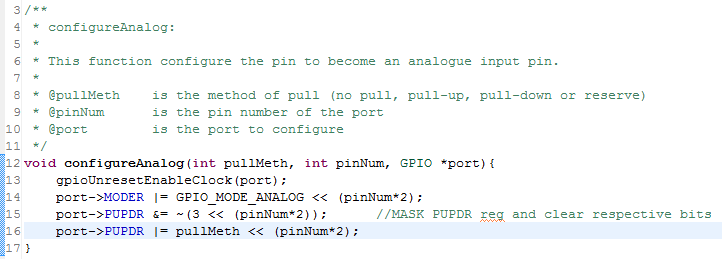


Figure 32 Function used to configure the GPIO port to analog pin

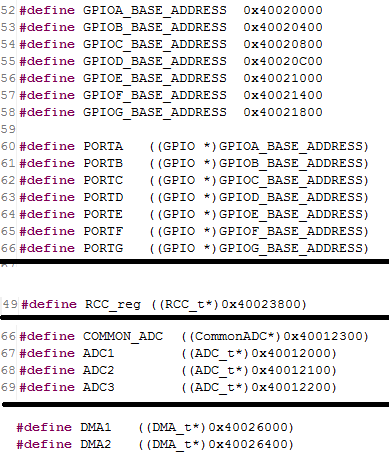


Figure 33 Base Address of all the required peripherals

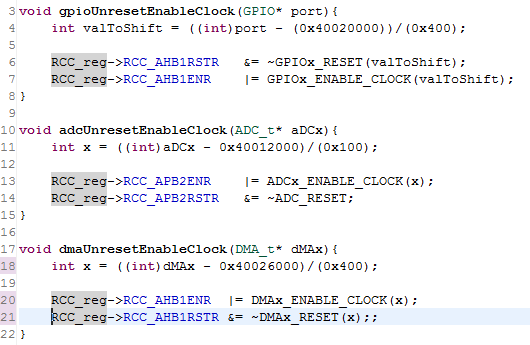


Figure 34 Functions in RCC to unreset pin and enable clock for different peripherals

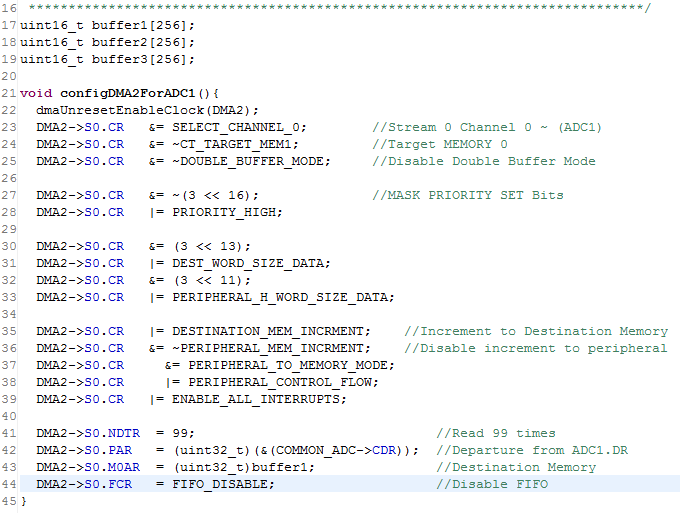


Figure 35 Declaration of buffer1, buffer2 and buffer3 and function to configure DMA2 for ADC1.

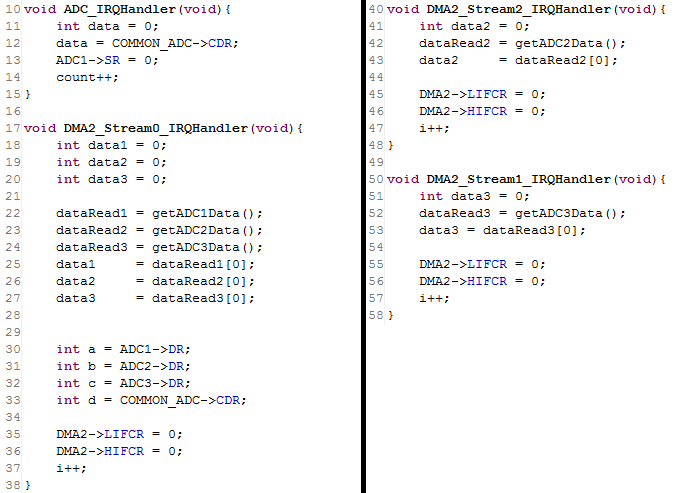


Figure 36 The Interrupt Handler

# APPENDIXES MAIN

**int** **main**(**void**){

uint32\_t sysCLK = getSystemClock();

uint32\_t p2CLK = getAPB2Clock(sysCLK);

configureOutput(GPIO\_SPEED\_V\_HIGH, PIN\_14, PORTG);

configureOutput(GPIO\_SPEED\_V\_HIGH, PIN\_13, PORTG);

configureAnalog(NO\_PULL, PIN\_0, PORTA);

configureAnalog(NO\_PULL, PIN\_3, PORTA);

configureAnalog(NO\_PULL, PIN\_6, PORTA);

configDMA2ForADC1();

configDMA2ForADC2();

configDMA2ForADC3();

configADC(ADC1, *Channel\_6*); //Change to Channel 6 when Multi Mode test

configADC(ADC2, *Channel\_3*);

configADC(ADC3, *Channel\_0*);

// addRegularQueue(ADC1, Channel\_3);

// addRegularQueue(ADC1, Channel\_6);

setSampleTime(*CYCLE\_15*, ADC1, *Channel\_6*);

setSampleTime(*CYCLE\_15*, ADC2, *Channel\_3*);

setSampleTime(*CYCLE\_15*, ADC3, *Channel\_0*);

//CHANGE RESOLUTION HERE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

setResolution(*RESOLUTION\_8\_BITS*, ADC1);

setResolution(*RESOLUTION\_8\_BITS*, ADC2);

setResolution(*RESOLUTION\_8\_BITS*, ADC3);

// adcEnableSignleDMA(ADC1);

// adcEnableSignleDMA(ADC2);

// adcEnableSignleDMA(ADC3);

adcEnableMultiADC();

setContMode(ADC1);

setContMode(ADC2);

setContMode(ADC3);

enableDMA();

startRegularConv(ADC1);

startRegularConv(ADC2);

startRegularConv(ADC3);

HAL\_NVIC\_EnableIRQ(*ADC\_IRQn*);

HAL\_NVIC\_EnableIRQ(*DMA2\_Stream0\_IRQn*);

HAL\_NVIC\_EnableIRQ(*DMA2\_Stream1\_IRQn*);

HAL\_NVIC\_EnableIRQ(*DMA2\_Stream2\_IRQn*);

**while**(1){

writeOne(PIN\_13, PORTG);

writeZero(PIN\_14, PORTG);

\_delay(100000);

writeZero(PIN\_13, PORTG);

writeOne(PIN\_14, PORTG);

\_delay(100000);

}

}

Full code of main

# APPENDIXES ADC

**#include** "Adc.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Author : Steven Foong Ghin Yew (Emerald)

\* Course : RMB2

\* Date : 20 November 2015

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This file consist of 11 functions that simplify the development

\* of software using ADC. The function are listed below:

\*

\* 1. configADC

\* 2. setResolution

\* 3. setSampleTime

\* 4. getRegularData

\* 5. getInjectedData

\* 6. startRegularConv

\* 7. startInjectedConv

\* 8. addRegularQueue

\* 9. addInjectedQueue

\* 10. enableVbat

\* 11. enableTempSensor

\* 12. enableRegularWD

\* 13. enableInjectedWD

\* 14. setContMode

\* 15. setDisconMode

\* 16. adcEnableDMA

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Control that can be made but not written as function here are:

\*

\* @aDCx can be ADC1, ADC2 or ADC3

\*

\* 1. Alignment of data (Right or Left) default set to right in configureADC

\* For left align, use the following code:

\* aDCx->CR2 |= LEFT\_ALIGN;

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1. configADC:

\*

\* This function enable the ADCx by awaken the ADCx and enable the CLOCK

\* - Enabled EOC and JEOC to generate Interrupt

\* - Channel 0 set as 1st and only conversion in regular queue

\* - Channel 18 (Vbat) set as 1st and only conversion in injected queue

\* - Data obtained is right aligned

\* - Only 1 conversion will be done

\*

\* NOTE: It call adcUnresetEnableClock in Rcc.h to enable the CLOCK

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** **configADC**(ADC\_t\* aDCx, Channel regularChannel){

adcUnresetEnableClock(aDCx);

aDCx->CR2 |= AWAKEN\_ADC; //Wake up the ADC

/\*\*\*\*\*Enable EOC interrupts\*\*\*\*\*/

aDCx->CR1 |= EOC\_INTERRUPT\_ENB;

aDCx->CR1 |= JEOC\_INTERRUPT\_ENB;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

COMMON\_ADC->CCR |= ENABLE\_VBAT; //Enable Vbat for Injected Group

aDCx->JSQR &= ~(3 << 20); //1 Conversion in Injected Group

aDCx->JSQR |= *Channel\_18*; //Queue Channel\_18 (Vbat) to Injected Group

aDCx->SQR1 &= ~(15 << 20); //1 Conversion in Regular Group

aDCx->SQR3 |= regularChannel; //Queue Channel\_0 (PA0) to Injected Group

aDCx->CR2 &= ~CONTINUOUS\_CONVERSION; //Single Conversion and END

aDCx->CR2 &= ~LEFT\_ALIGN; //Right Align Data

}

/\*\*

\* 2. setResolution:

\*

\* This function configure the resolution of the converted analogue signal.

\* Define how precise the result wanted.

\*

\* NOTE: The resolution set for every channels configured for ADCx

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @res is the resolution of the result, (6, 8, 10 or 12)bits

\*/

**void** **setResolution**(Resolution res, ADC\_t\* aDCx){

aDCx->CR1 &= ~(3 << 24);

aDCx->CR1 |= res << 24;

}

/\*\*

\* 3. setSampleTime:

\*

\* This function select the sample time for conversion of each channel.

\* Define the time to charge up the capacitor and stabilise the voltage

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @sampTime is the resolution of the result, (6, 8, 10 or 12)bits

\*/

**void** **setSampleTime**(SampleTime sampTime, ADC\_t\* aDCx, Channel channel){

**if**(channel < 10){

aDCx->SMPR2 &= ~(7 << channel);

aDCx->SMPR2 |= (sampTime << channel\*3);

}

**else** {

aDCx->SMPR1 &= ~(7 << channel);

aDCx->SMPR1 |= (sampTime << (channel - 10)\*3);

}

}

/\*\*

\* 4. getRegularData:

\*

\* This function read the data of regular group and return it to the caller

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*/

**int** **getRegularData**(ADC\_t\* aDCx){

**int** data = (aDCx->DR);

**return** data;

}

/\*\*

\* 5. getInjectedData:

\*

\* This function read the data of injected group and return it to the caller

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @queue is the current position in the sequence of queue in the list

\*/

**int** **getInjectedData**(ADC\_t\* aDCx, **int** queue){

**switch**(queue){

**case** 0:

**return** aDCx->JDR1;

**break**;

**case** 1:

**return** aDCx->JDR2;

**break**;

**case** 2:

**return** aDCx->JDR3;

**break**;

**case** 3:

**return** aDCx->JDR4;

**break**;

**default**:

**return** aDCx->JDR1;

**break**;

}

}

/\*\*

\* 6. startRegularConv:

\*

\* This function start the conversion of the regular group

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*/

**void** **startRegularConv**(ADC\_t\* aDCx){

aDCx->CR2 |= START\_REGULAR\_CONVERSION;

}

/\*\*

\* 7. startInjectedConv:

\*

\* This function start the conversion of the Injected group

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*/

**void** **startInjectedConv**(ADC\_t\* aDCx){

aDCx->CR2 |= START\_INJECT\_CONVERSION;

}

/\*\*

\* 8. addRegularQueue:

\*

\* This function add the input channel into the current regular

\* conversion group queue. Note that there is a default channel\_0

\* set as the 1st ADC conversion. The 1st channel in the queue can

\* be modified manually. Please refer to datasheet if wanted to

\* change the 1st conversion channel in the queue.

\*

\* If the queue is FULL with 16 channel queue,

\* the function will not do anything

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @channel is the channel to add to the current queue

\*/

**void** **addRegularQueue**(ADC\_t\* aDCx, Channel channel){

**int** numOfConversion = (aDCx->SQR1) >> 20;

**if**(numOfConversion < 6){

numOfConversion++;

aDCx->SQR3 &= ~(31 << (numOfConversion \* 5));

aDCx->SQR3 |= channel << (numOfConversion \* 5);

}

**else** **if**(numOfConversion < 12){

numOfConversion++;

aDCx->SQR2 &= ~(31 << ((numOfConversion - 6) \* 5));

aDCx->SQR2 |= channel << ((numOfConversion - 6) \* 5);

}

**else** **if**(numOfConversion < 15){

numOfConversion++;

aDCx->SQR1 &= ~(31 << ((numOfConversion - 12) \* 5));

aDCx->SQR1 |= channel << ((numOfConversion - 12) \* 5);

}

aDCx->SQR1 &= ~(15 << 20);

aDCx->SQR1 |= numOfConversion << 20;

}

/\*\*

\* 9. addInjectedQueue:

\*

\* This function add the input channel into the current injected

\* conversion group queue. Note that there is a default channel\_0

\* set as the 1st ADC conversion. The 1st channel in the queue can

\* be modified manually. Please refer to datasheet if wanted to

\* change the 1st conversion channel in the queue.

\*

\* If the queue is FULL with 4 channel queue,

\* the function will not do anything

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @channel is the channel to add to the current queue

\*/

**void** **addInjectedQueue**(ADC\_t\* aDCx, Channel channel){

**int** numOfConversion = (aDCx->JSQR) >> 20;

**if**(numOfConversion < 4){

numOfConversion++;

aDCx->JSQR &= ~(31 << (numOfConversion \* 5));

aDCx->JSQR |= channel << (numOfConversion \* 5);

}

aDCx->JSQR &= ~(15 << 20);

aDCx->JSQR |= numOfConversion << 20;

}

/\*\*

\* 10. enableVbat & 11. enableTempSensor :

\*

\* This two functions enable the internal sensors Vbat (battery voltage)

\* & TempSensor (temperature degree celcius)

\*

\* Both sensors are connected to channel 18 in stm32f42x & stm32f43x

\* Both sensors are connected to channel 16 in stm32f40x & stm32f41x

\*/

**void** **enableVbat**(**void**){

COMMON\_ADC->CCR |= ENABLE\_VBAT;

COMMON\_ADC->CCR &= ~ENABLE\_TEMP\_AND\_VREFINT;

}

**void** **enableTempSensor**(){

COMMON\_ADC->CCR |= ENABLE\_TEMP\_AND\_VREFINT;

COMMON\_ADC->CCR &= ~ENABLE\_VBAT;

}

/\*\*

\* 12. enableRegularWD & 13. enableInjectedWD :

\*

\* This two functions enable the Watchdog feature to guard the input signal

\* referencing to the Threshold set in aDCx->HTR and aDCx->LTR.

\*

\* AWD flag will be raised in aDCx->SR when the input signal exceed the limit

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @useIRQ is the selection of interrupt generation

\* YES = generate interrupt when AWD rise

\* NO = do not generate interrupt when AWD rise

\*/

**void** **enableRegularWD**(ADC\_t\* aDCx, Question useIRQ){

aDCx->CR1 |= ENABLE\_REGULAR\_WATCHDOG;

**if**(useIRQ == *YES*)

aDCx->CR1 |= AWD\_INTERRUPT\_ENB;

**else**

aDCx->CR1 &= ~AWD\_INTERRUPT\_ENB;

}

**void** **enableInjectedWD**(ADC\_t\* aDCx, Question useIRQ){

aDCx->CR1 |= ENABLE\_INJECTED\_WATCHDOG;

**if**(useIRQ == *YES*)

aDCx->CR1 |= AWD\_INTERRUPT\_ENB;

**else**

aDCx->CR1 &= ~AWD\_INTERRUPT\_ENB;

}

/\*\*

\* 14. setContMode

\*

\* This function enable both the Regular Group and Injected Group to

\* perform ADC continuosly. However, it remove the higher priority of

\* Injected Group to Regular Group, making both Group sharing the same

\* priority. To DISABLE the auto conversion on Injected Group, enter

\* aDCx->CR1 &= ~SET\_JAUTO;

\* into your function.

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*/

**void** **setContMode**(ADC\_t\* aDCx){

aDCx->CR2 |= CONTINUOUS\_CONVERSION;

aDCx->CR1 |= SET\_JAUTO;

}

/\*\*

\* 15. setDisconMode

\*

\* This function enable the selected group to chop the queue into

\* smaller group of channels queue. Eg. Regular Group queued

\* 1. Channel\_0

\* 2. Channel\_1

\* 3. Channel\_6

\* 4. Channel\_7

\* 5. Channel\_12

\*

\* with grp = REGULAR\_GRP and numOfChnDiscon = 3

\* the conversion will perform when triggered

\* 1. Channel\_0

\* 2. Channel\_1

\* 3. Channel\_6

\* Then, it wait for second trigger and convert

\* 1. Channel\_7

\* 2. Channel\_12

\* Then restart from Channel\_0 upon the 3rd time trigger

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\* @grp is the selection of Group (REGULAR\_GRP or INJECTED\_GRP)

\* @numOfChnDiscon is the number of channel to be chop into a small group

\*/

**void** **setDisconMode**(ADC\_t\* aDCx, **int** grp, **int** numOfChnDiscon){

**if**(grp == REGULAR\_GRP){

aDCx->CR1 |= ENABLE\_REGULAR\_DISCON;

aDCx->CR1 |= numOfChnDiscon << 13;

}

**else**{

aDCx->CR1 |= ENABLE\_INJECTED\_DISCON;

aDCx->CR1 |= numOfChnDiscon << 13;

aDCx->CR1 &= ~SET\_JAUTO; //JAUTO cannot be set together with DISCON, check datasheet

}

}

/\*\*

\* 16. adcEnableSignleDMA & 17. adcEnableMultiADC

\*

\* These function enable the use of DMA to transfer data from DR register to Memory Located

\*

\* @aDCx is the selection of ADC (ADC1, ADC2 or ADC3)

\*/

**void** **adcEnableSignleDMA**(ADC\_t\* aDCx){

aDCx->CR2 |= 3 << 8; //ENABLE DMA and DDS

}

**void** **adcEnableMultiADC**(){

COMMON\_ADC->CCR |= ~(7 << 13); //MASKED DMA config bits

COMMON\_ADC->CCR |= 4 << 13; //Triple ADC Mode

COMMON\_ADC->CCR |= 0x15; //ADC1, ADC2, ADC3 working together on Regular Simultaneous Mode.

}

Full Code Of ADC