APPLICATION NOTE
Using DMA for ADC Continuous Conversion Mode

Rodrigo Drummond Lima

August 3, 2021

1 Introduction

This application note was developed as a project for the Embedded Systems Programming course at the Federal University of Minas Gerais (UFMG) in August 2021, with professor Ricardo de Oliveira Duarte.

This application note describes how to use the Direct Memory Access (DMA) to obtain data from the Analog to Digital Converter (ADC) working with continuous conversion. This operation mode is very useful in applications such as waveform visualization and signal processing.

2 The ADC in STM32F103RB

The ADC in the STM32F103RB is a successive approximation type and has a 12-bit resolution. It means the data acquired is stored into a 16-bit register, with the 4 leftmost bits being 0 or signal-extended, depending on channel converted. In this context, the 12-bit resolution leads to 4096 possible values, so, if the ADC has voltage references of 0V an 3.3V, one bit corresponds to 0.8mV.

The ADC also features a continuous conversion mode, in which the ADC starts another conversion as soon as it finishes the previous one. This mode can be configured through the STM32CubeMX by enabling it in the ADC configuration tab. The ADC also features a DMA request, allowing it to use peripheral to memory direction to write the data in the 16-bit register to a buffer in memory using the Direct Memory Access controller.

3 The DMA Controller

Since we will use the continuous conversion mode of the ADC, we gather a big stream of data in a short amount of time. In this way, direct memory access is a way to store this data quickly without utilising too much CPU resources, keeping it free for other operations. The STM32F103RB has only one DMA controller with seven channels. The priorities for each channel are decided by its number, the lower number has higher priority by default, but these attributes can be modified by the user.

The DMA can be configured to store data in circular mode, in which it stores data in a FIFO buffer. This mode in particular is very useful for the ADC continuous conversion mode, since it starts one conversion after another and keeps updating the data in the memory buffer.

DMA is also useful for transferring data to different parts of the memory or peripherals. For instance, you can use DMA to pass a buffer to the UART Transmission Data Register and transmit this buffer via TX pin.

4 Application example

In this example, we want to use DMA channel 1 to store data converted by the ADC in a circular buffer, process it, use DMA channel 2 to transfer the processed data to UART_TX data register and read the data transferred in a serial monitor. In the test, it will be used a 10kΩ potentiometer with its middle pin connected to the ADC input and the others to 3.3V and GND.

4.1 Setting up the STM32CubeMX

First, we need to attach the ADC1 to a GPIO pin. In this example, we used IN0, which corresponds to A0 in the CN8 block. In configuration, you need to enable continuous conversion mode, as shows figure 1.

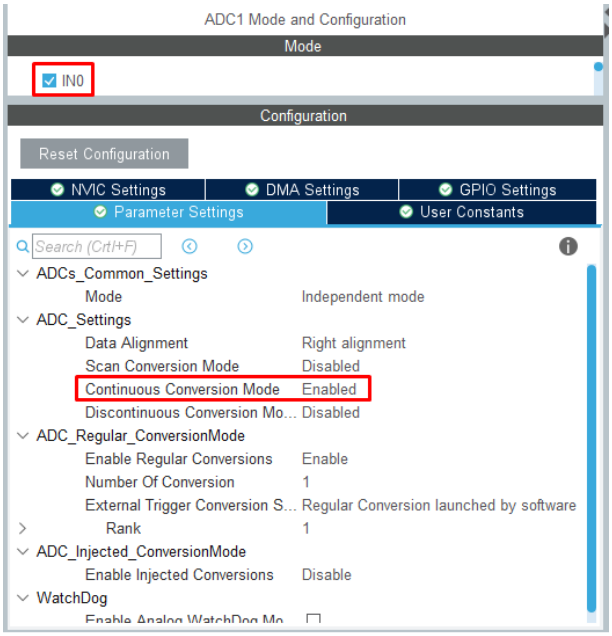

Figure 1: ADC configuration in STM32CubeMX.

Next, configure the USART3 to asynchronous mode and set the desired Baud Rate. In this example, we used 9600 Bits/s. The TX pin is attached to PA2 an RX is attached to PA3. See figure 2.

Figure 2: USART3 configuration in STM32CubeMX.

Finally, we can setup the tho DMA channels. For both, we want to use circular mode and Half-Word data width, as we are going to use a FIFO buffer. Figure 3 shows the configuration for DMA channel 1, attached to ADC1 and figure 4 shows the configuration for channel 2, attached to USART3_TX.

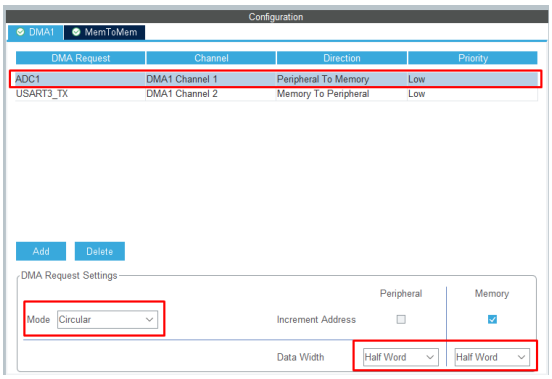

Figure 3: DMA configuration for ADC1 in STM32CubeMX.

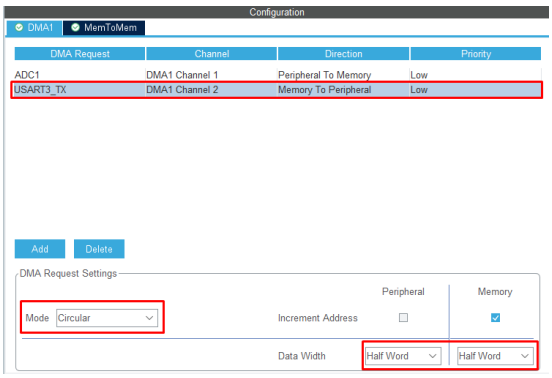

Figure 4: DMA configuration for USART3_TX in STM32CubeMX.

4.2 Software implementation

First, a global 16 bit buffer is created with the ADC_BUF_SIZE defined. The STM32CubeMX automatically creates the handlers for both DMAs and the ADC1 and USART3 themselves. Inside the main code, the HAL function HAL_ADC_Start_DMA is used to start the DMA attached to the ADC1. Two callbacks for the ADC are used in the application. The first one is the HAL_ADC_ConvHalfCpltCallback, which is called when the buffer is half filled. Inside this callback, the first half of the buffer will be processed to be sent to UART while the other half will be filled by DMA at the same time. When the buffer is filled, it will be called the second callback function, HAL_ADC_ConvCpltCallback, which will process the other half of the buffer.

This strategy regarding filling half of the buffer to process it while the other half is being filled is called ping-pong buffer and it is very useful and efficient because CPU can use half of the buffer while the DMA fills the other half. Figure 5 shows a block diagram that illustrates it.

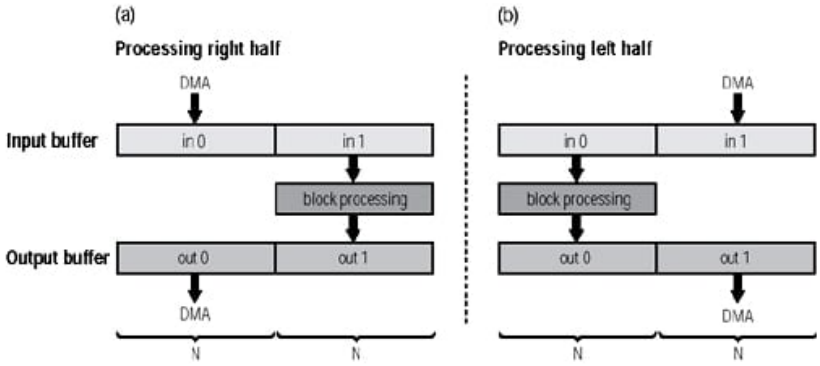

Figure 5: Block diagram of a ping-pong buffer.

The processing made in each callback consists in converting the 12 bit data output of the ADC to a 8 bit one, since we want to send it via UART, which is 8-bit long. In that sense, we simply do a bitwise 4-bit shift to the right and store this value to another buffer of the same size, but with 8-bit integer type. In that way, we expect to read the 256 as the highest value and 0 for the lowest.

Inside the while loop, we set the DMA transmiter bit to send data to the UART via DMA. Next, start the DMA attached to the UART with interrupt mode and indicate the data to be transferred, which is the buffer created in the ADC callbacks. A callback to indicate that the transfer was complete was also created so the USART3 DMA is disabled inside it.

4.3 Results

The board with all connections can be seen in figure 6.

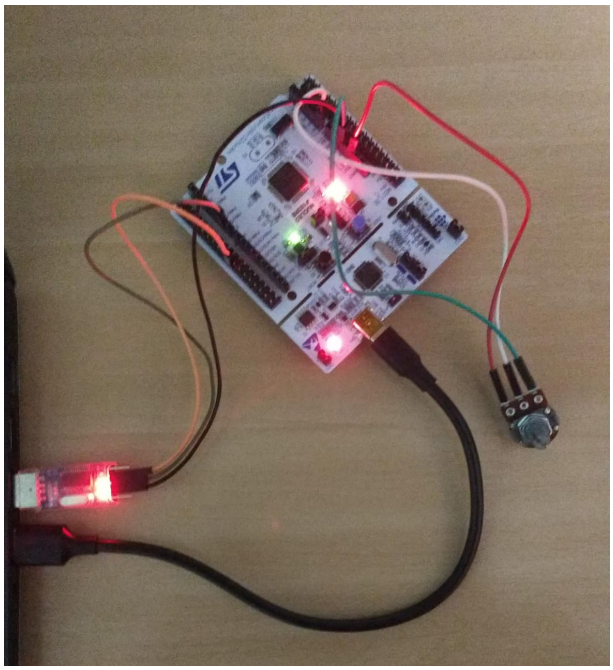

Figure 6: Pin connections to the STM32F103RB Board for the example application.

First, testing with the debug tool, we can check the adc_buf buffer in different positions of the potentiometer. figures 7, 8 and 9 show the values inside the buffer for minimum, intermediate and maximum resistances of the potentiometer.

Expression	Type	Value
adc_buf	uint16_t [216]	0x20000200 <adc_buf>
[0...99]	uint16_t [100]	0x20000200 <adc_buf>
adc_buf[0]	uint16_t	0
adc_buf[1]	uint16_t	0
adc_buf[2]	uint16_t	0
adc_buf[3]	uint16_t	0
adc_buf[4]	uint16_t	0
adc_buf[5]	uint16_t	0
adc_buf[6]	uint16_t	0
adc_buf[7]	uint16_t	0
adc_buf[8]	uint16_t	0

Figure 7: ADC buffer values for minimum voltage.

Expression	Type	Value
adc_buf	uint16_t [216]	0x20000200 <adc_buf>
[0...99]	uint16_t [100]	0x20000200 <adc_buf>
adc_buf[0]	uint16_t	3081
adc_buf[1]	uint16_t	3079
adc_buf[2]	uint16_t	3077
adc_buf[3]	uint16_t	3081
adc_buf[4]	uint16_t	3077
adc_buf[5]	uint16_t	3077
adc_buf[6]	uint16_t	3077
adc_buf[7]	uint16_t	3076
adc_buf[8]	uint16_t	3079

Figure 8: ADC buffer values for intermediate voltage.

Expression	Type	Value
adc_buf	uint16_t [216]	0x20000200 <adc_buf>
[0...99]	uint16_t [100]	0x20000200 <adc_buf>
adc_buf[0]	uint16_t	4027
adc_buf[1]	uint16_t	4029
adc_buf[2]	uint16_t	4029
adc_buf[3]	uint16_t	4027
adc_buf[4]	uint16_t	4025
adc_buf[5]	uint16_t	4026
adc_buf[6]	uint16_t	4028
adc_buf[7]	uint16_t	4027
adc_buf[8]	uint16_t	4029

Figure 9: ADC buffer values for maximum voltage.

For the 8-bit buffer generated in the ping-pong buffer and sent to the UART, figures 10, 11 and 12 were extracted.

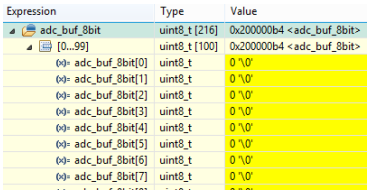

Figure 10: UART buffer values for minimum voltage.

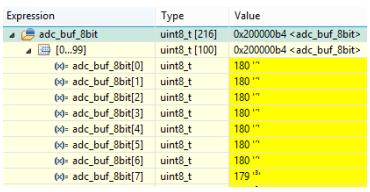

Figure 11: UART buffer values for intermediate voltage.

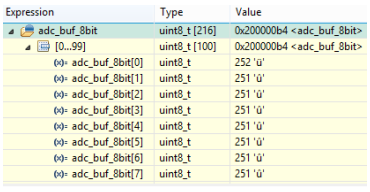

Figure 12: UART buffer values for maximum voltage.

Finally, we used a serial monitor to print the data received via UART port. The program used was RealTerm. Figures 13, 14 and 15 show the results for minimum, intermediate and maximum voltages.

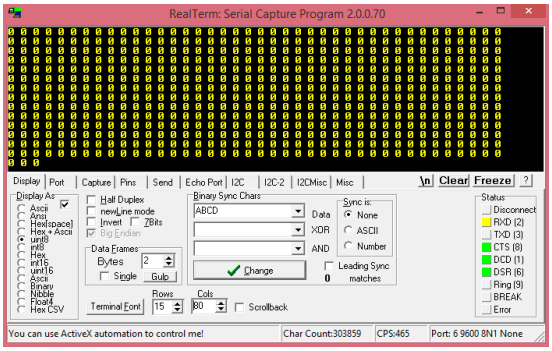

Figure 13: Data received shown in serial monitor for minimum voltage.

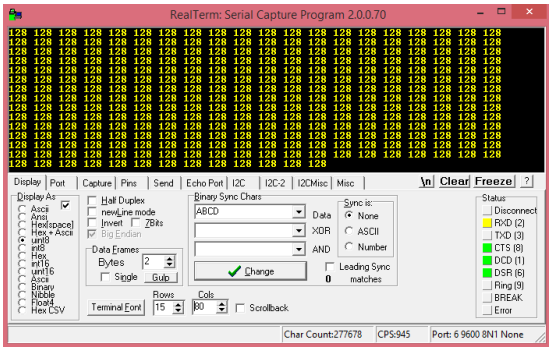

Figure 14: Data received shown in serial monitor for intermediate voltage.

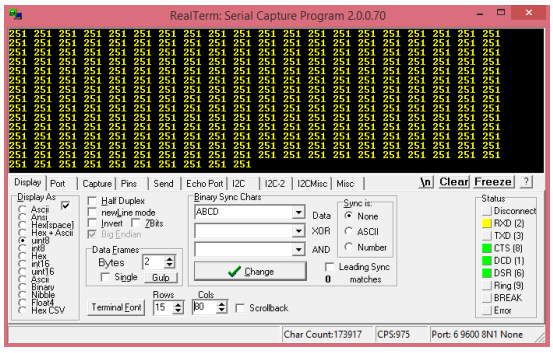

Figure 15: Data received shown in serial monitor for maximum voltage.

References

[1] **RM0008 - Reference Manual** *STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and. STM32F107xx advanced Arm. ®. -based 32-bit MCUs*

[2] **UM1850 - User Manual** *Description of STM32F1 HAL and Low-layer drivers*