NXP Semiconductors

Using DMA to Emulate ADC Flexible Scan Mode with SDK 2.x

By: Technical Information Center

Table of Contents

1 Introduction 2
2 DMA to emulate ADC Flexible Scan 3
2.1 System overview and flow diagram for Flexscan. 3
3 SDK implementation 6
3.1 ADC configuration 6
3.2 LPMTR configuration 7
3.3 DMA configuration 7
4 ADC Flex Scan mode with DMA 10
Appendix A: Requirements 11
Reference 12

NXP SECURE CONNECTIONS FOR A SMARTER WORLD

1 Introduction

This document describes how to combine ADC, DMA and a timer to implement a ADC Flexible storing data with SDK 2.2 and MCUXpresso IDE. In this configuration ADC measures will be stored into an internal memory buffer with DMA, which imply, no CPU. With this configuration MCU uses less resources, only using ADC and DMA (2 channels) and a timer to trigger conversions. This way, the MCU does not need to read the ADC result register, because the memory management will be done by DMA automatically.

To implement this application we will use SDK libraries, which includes ADC and DMA libraries. The timer used to trigger ADC conversion will be the LPTMR. MCUXpresso IDE will be used as development environment, please check appendix A to look where you can download SDK libraries and IDE. The project will be created from scratch, regardless of this, MCUXpresso IDE and SDK libraries allows us to create a project with some headers and libraries in the project, if you have problems in creating a new project in MCUXpresso please check Reference 4.

In this document we will use the FRDM-K64F which is an ultra-low-cost development platform, but this implementation could be easily migrated to any kinetis family that support second-generation eDMA module (enhanced Direct Memory Access).

This document is an update of the community doc: https://community.nxp.com/docs/DOC-104395 which uses old SDK libraries.

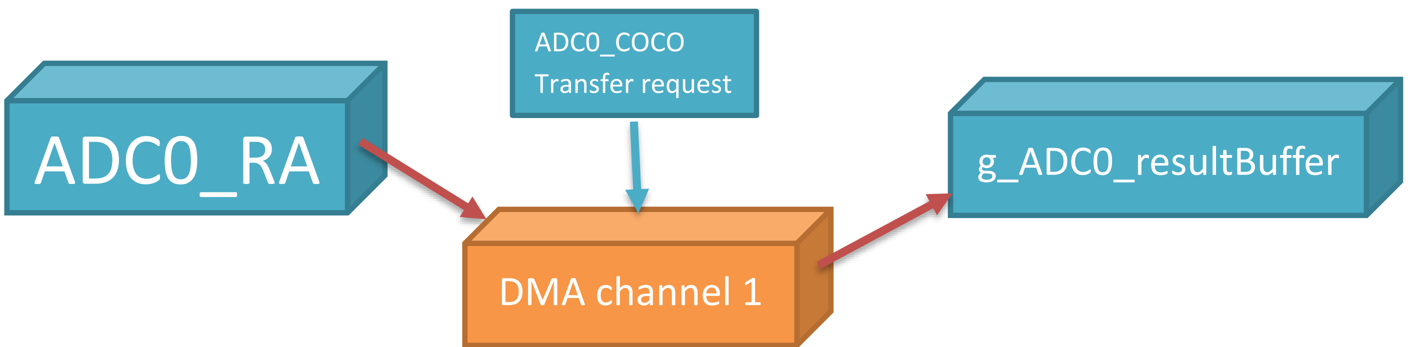
2 DMA to emulate ADC Flexible Scan

Flexible scan implementation needs 2 types of data movement: ADC measures (from peripheral to internal memory) and ADC mux change (from internal memory to peripheral). Second one is needed to change ADC input channel and have the flexible implementation, first one save ADC measures in our internal memory.

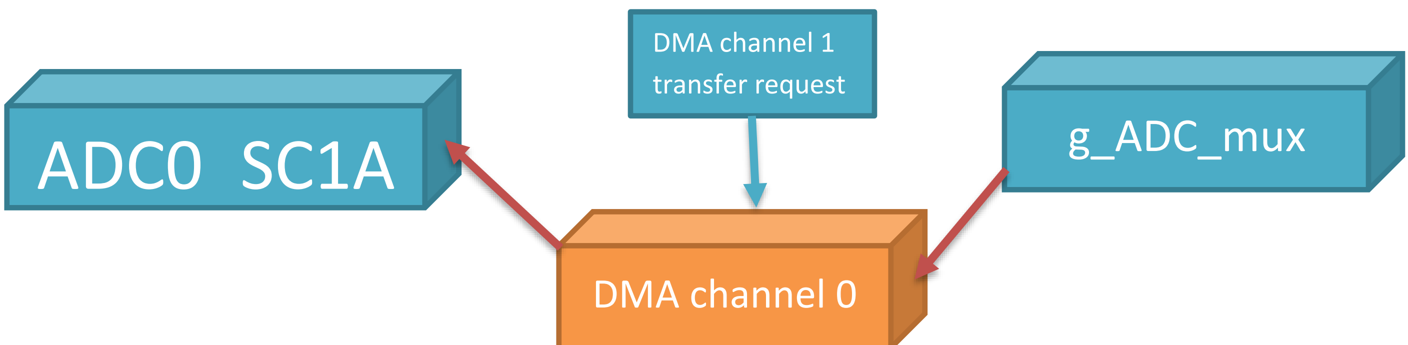
DMA peripheral (for this MCU) includes up to 16 channels, for this application we will only use 2 channels, one for ADC measures and one for ADC muxing channel. It is important to remark that we will use the linking feature in these channels to implement the ADC changing channel automatically, later in this document it will be explain this feature.

2.1 System overview and flow diagram for Flexscan.

As already mentioned, this implementation will use one DMA channel to transfer data from ADC measures (ADC0_RA register in this case) to a memory buffer (named here as g_ADC0_resultBuffer), so follow figure shows this movement:



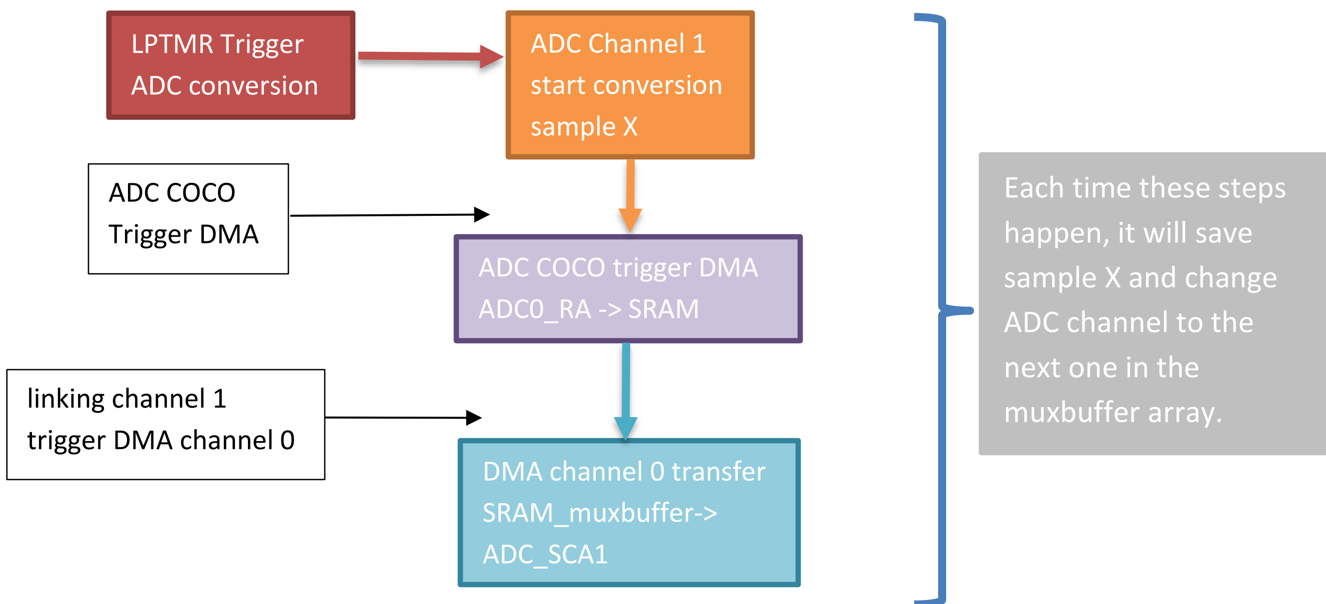
As you can see, ADC0_COCO (conversion complete) will be the one that request a transfer from ADC0_RA to resultBuffer. Once this transfer has completed, we need to change ADC channel, so when DMA channel 1 transfer finishes, it will indicate us to trigger the other data movement, in this case from internal buffer g_ADC_mux (that save our adc channels) to the ADC0_SC1A register that sets the ADC channel, so the following diagram shows this.



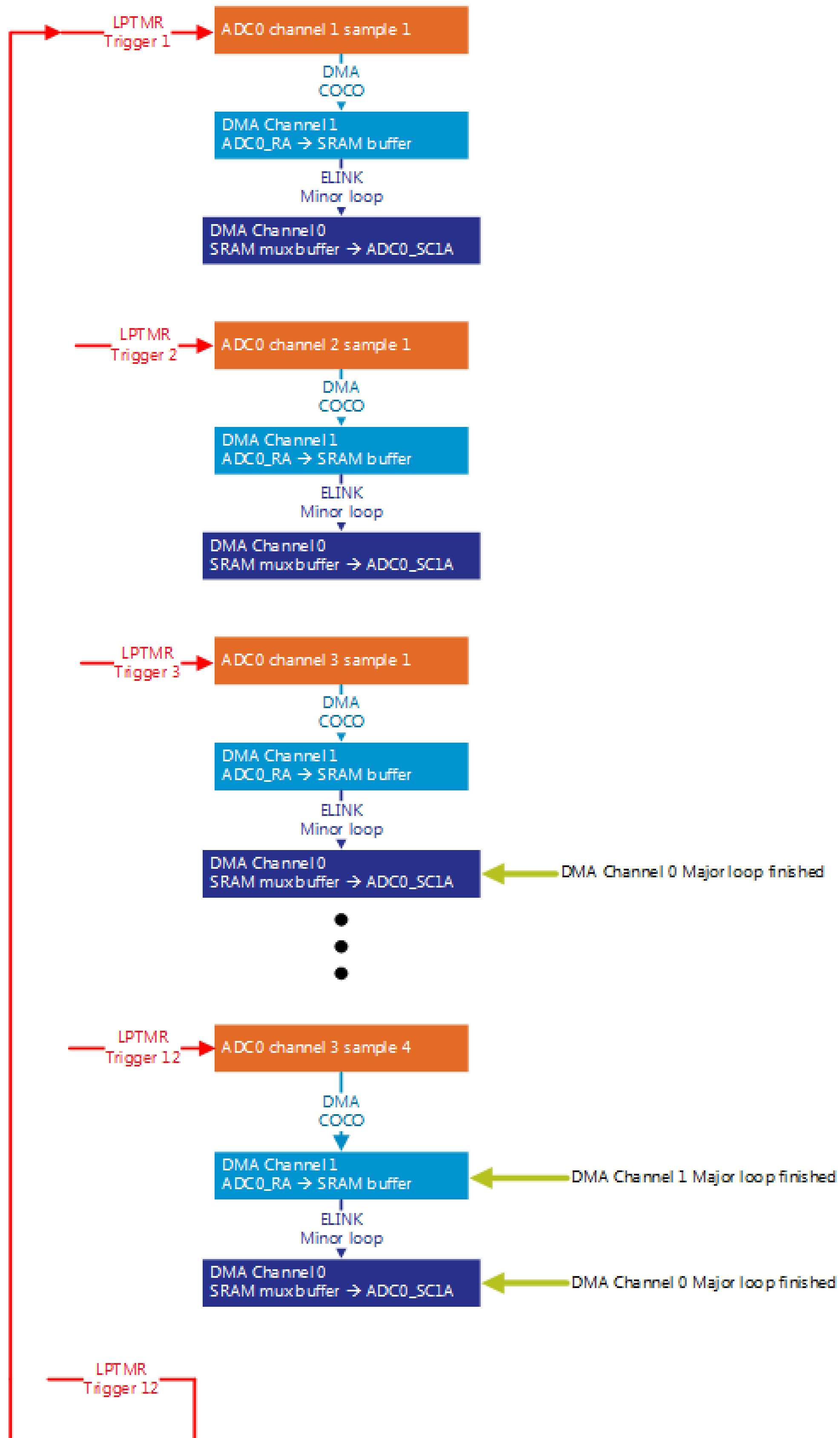
To do this, this application used a DMA feature that link DMA channels, channel-to-channel linking. So, when one DMA channel finished, the linked DMA channel will be triggered, in this case when DMA channel 1 has completed to move data from ADC measure register to internal memory then DMA channel 0 need to be triggered to change the ADC channel, so next conversion ADC would have selected other ADC channel.

Once that the ADC channels is changed (and the HW trigger happens), the COCO flag from ADC will trigger other DMA request in channel 0 and the process will start again. This flow process can be repeated as many channels as we have and as many samples as we want, for this example code we will measure 3 channels and have 4 samples, so the result buffer size is 3 × 4 = 12 (the real buffer size is 16, to demonstrate that only 12 data field parts are written).

The ADC works in hardware trigger mode, with the LPTMR timer working as the trigger source. This mean that even though DMA channel 0 changed the ADC input channel, conversion will not start until HW trigger (LPTMR) start the conversion, this mean that the flow of the program can be as follow.



In the example code provided, when DMA channel 0 has changed 3 times the ADC channel, DMA channel 1 major loop will happen, and when the sample X is the last sample (sample 4) of the last channel in muxbuffer, then Major loop in DMA channel 0 happens and the application ends (or it could start again). This mean that the application here can be shown as:


LPTMR Trigger 1
ADC0 channel 1 sample 1
DMA COCO
DMA Channel 1
ADC0_RA → SRAM buffer
ELINK Minor loop
DMA Channel 0
SRAM mux buffer → ADC0_SC1A
LPTMR Trigger 2
ADC0 channel 2 sample 1
DMA COCO
DMA Channel 1
ADC0_RA → SRAM buffer
ELINK Minor loop
DMA Channel 0
SRAM mux buffer → ADC0_SC1A
LPTMR Trigger 3
ADC0 channel 3 sample 1
DMA COCO
DMA Channel 1
ADC0_RA → SRAM buffer
ELINK Minor loop
DMA Channel 0
SRAM mux buffer → ADC0_SC1A
DMA Channel 0 Major loop finished
LPTMR Trigger 12
ADC0 channel 3 sample 4
DMA COCO
DMA Channel 1
ADC0_RA → SRAM buffer
DMA Channel 1 Major loop finished
ELINK Minor loop
DMA Channel 0
SRAM mux buffer → ADC0_SC1A
DMA Channel 0 Major loop finished
LPTMR Trigger 12

3 SDK implementation

Implementation of this code can be dived in three parts; LPTMR, ADC and DMA configuration.

3.1 ADC configuration

```
/*****************************************
 * ADC Config
 *****************************************/
EnableIRQ(ADC0_IRQn);

adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;
/*
 * adc16ConfigStruct.referenceVoltageSource = kADC16_ReferenceVoltageSourceVref;
 * adc16ConfigStruct.clockSource = kADC16_ClockSourceAsynchronousClock;
 * adc16ConfigStruct.enableAsynchronousClock = true;
 * adc16ConfigStruct.clockDivider = kADC16_ClockDivider8;
 * adc16ConfigStruct.resolution = kADC16_ResolutionSE12Bit;
 * adc16ConfigStruct.longSampleMode = kADC16_LongSampleDisabled;
 * adc16ConfigStruct.enableHighSpeed = false;
 * adc16ConfigStruct.enableLowPower = false;
 * adc16ConfigStruct.enableContinuousConversion = false;
 */
ADC16_GetDefaultConfig(&adc16ConfigStruct);
ADC16_Init( ADC0, &adc16ConfigStruct);

ADC16_DoAutoCalibration( ADC0 );

ADC16_EnableHardwareTrigger( ADC0, true);
ADC16_EnableDMA( ADC0, true);

adc16ChannelConfigStruct.channelNumber = g_ADC_mux[2];
adc16ChannelConfigStruct.enableInterruptOnConversionCompleted = true; /* Enable the interrupt. */
adc16ChannelConfigStruct.enableDifferentialConversion = false;
ADC16_SetChannelConfig( ADC0, 0, &adc16ChannelConfigStruct);
```

ADC configuration is a normal ADC config, with 12 single ended, Asynchronous clock source and Vref as reference voltage. Hardware trigger and DMA support are enable to trigger conversions with LPTMR and to be able to trigger a DMA request when COCO interrupt happen. In ADC channel configuration, it is enable Conversion completed interrupt and it is loaded the value of g_ADC_mux to channel number, this will be the first channel that LPTMR will trigger.

3.2 LPMTR configuration

```
/******************************************
 * LPTMR for ADC HW trigger Config
 ******************************************/
lptmr_config_t lptmrConfig;
/*
 * lptmrConfig.timerMode = kLPTMR_TimerModeTimeCounter;
 * lptmrConfig.pinSelect = kLPTMR_PinSelectInput_0;
 * lptmrConfig.pinPolarity = kLPTMR_PinPolarityActiveHigh;
 * lptmrConfig.enableFreeRunning = false;
 * lptmrConfig.bypassPrescaler = true;
 * lptmrConfig.prescalerClockSource = kLPTMR_PrescalerClock_1;
 * lptmrConfig.value = kLPTMR_Prescale_Glitch_0;
 */
LPTMR_GetDefaultConfig(&lptmrConfig);
lptmrConfig.bypassPrescaler = false;
lptmrConfig.prescalerClockSource = kLPTMR_PrescalerClock_1;

/* Initialize the LPTMR */
LPTMR_Init(LPTMR0, &lptmrConfig);

/* Set the LPTimer period */
LPTMR_SetTimerPeriod( LPTMR0,USEC_TO_COUNT(200000, CLOCK_GetFreq(kCLOCK_LpoClk)));

/* Configure SIM for ADC hw trigger source selection */
SIM->SOPT7 |= 0x8EU;
```

LPMTR is a common configuration with LPTMR as time counter. Please notice that prescaler could be used and select a prescaler clock. To set LPTMR period it is used the macro USEC_TO_COUNT, which just take the value in microseconds and it calculate the number of ticks to count, using the source clock. Also at the end, LTPMR is configured to be used as HW trigger for ADC conversions in the SIM register.

3.3 DMA configuration

```
/* Configure DMAMUX */
DMAMUX_Init(DMAMUX0);

DMAMUX_SetSource(DMAMUX0, DMAChannel_0, 60); /* Channel 0 Source 60: DMA always enabled */
DMAMUX_EnableChannel(DMAMUX0, DMAChannel_0);

DMAMUX_SetSource(DMAMUX0, DMAChannel_1, 40); /* Channel 1 Source 40: ADC COCO trigger */
DMAMUX_EnableChannel(DMAMUX0, DMAChannel_1);
```

For DMAMUX configuration it is enable Channel 0 and 1 and it is selected the source trigger, DMA for channel 0 (it will be triggered with linking feature) and ADC COCO flag for channel 1 (trigger when the ADC conversion complete).

```
EDMA_CreateHandle(&g_EDMA_Handle_1, DMA0, DMAChannel_1);
EDMA_SetCallback(&g_EDMA_Handle_1, EDMA_Callback_1, NULL);

EDMA_PrepareTransfer(&transferConfig_ch1,   /* Prepare TCD for CH1 */
                     (uint32_t*) (ADC0->R), /* Source Address (ADC0_RA) */
                     sizeof(uint16_t),      /* Source width (2 bytes) */
                     g_ADC0_resultBuffer,   /* Destination Address (Internal buffer)*/
                     sizeof(g_ADC0_resultBuffer[0]), /* Destination width (2 bytes) */
                     sizeof(uint16_t),      /* Bytes to transfer each minor loop (2 bytes) */
                     B_SIZE * 2,            /* Total of bytes to transfer (12*2 bytes) */
                     kEDMA_PeripheralToMemory); /* From ADC to Memory */
/* Push TCD for CH1 into hardware TCD Register */
EDMA_SubmitTransfer(&g_EDMA_Handle_1, &transferConfig_ch1);

/* Link channel 1 to channel 0
 * when channel 1 transfer ADC data,
 * then channel 0 will change ADC channel*/
EDMA_SetChannelLink(DMA0, DMAChannel_1, kEDMA_MinorLink, DMAChannel_0);
EDMA_SetChannelLink(DMA0, DMAChannel_1, kEDMA_MajorLink, DMAChannel_0);

EDMA_CreateHandle(&g_EDMA_Handle_0, DMA0, DMAChannel_0);
EDMA_SetCallback(&g_EDMA_Handle_0, EDMA_Callback_0, NULL);

EDMA_PrepareTransfer(&transferConfig_ch0,   /* Prepare TCD for CH0 */
                     &g_ADC_mux[0],         /* Source Address (ADC channels array) */
                     sizeof(g_ADC_mux[0]),  /* Source width (1 bytes) */
                     (uint32_t*)(ADC0->SC1),/* Destination Address (ADC_SC1A_ADCH)*/
                     sizeof(uint8_t),       /* Destination width (1 bytes) */
                     sizeof(uint8_t),       /* Bytes to transfer each minor loop (1 bytes) */
                     CHANNELS,              /* Total of bytes to transfer (3*1 bytes) */
                     kEDMA_MemoryToPeripheral);/* From ADC channels array to ADCH register */
/* Push TCD for CH0 into hardware TCD Register */
EDMA_SubmitTransfer(&g_EDMA_Handle_0, &transferConfig_ch0);
```

For DMA configuration, it is needed to create 2 tcd configuration, there are set in transferConfig_chx. First one defined is for DMA channel 1 (data from ADC measure to internal memory), and the second is for DMA channel 0 (data from internal memory to ADC_SC1 register to change ADC channel). Please noticed that these definitions are in bytes to transfer, so some of them are sizeof(uint16_t). Here is also the setup to link channel 1 to channel 0.

```
/* If transfer will continue it will need to adjust last source and destination */
DMA0->TCD[0].SLAST = -1 * CHANNELS;
DMA0->TCD[1].DLAST_SGA = -2 * B_SIZE;

EDMA_StartTransfer(&g_EDMA_Handle_1);

/* Start the LPTimer */
LPTMR_StartTimer( LPTMR0);
```

Finally, it is added the adjustment for TCD, so when DMA major loop finished in both, it will point to the start of the source and destination. Noticed that this is added just for internal memory address, this is because for the case of a peripheral (ADC in this case), pointer to address doesn’t change.

Then DMA channel 1 and LPTMR are started.

When DMA channels were initialized, their callbacks were also defined;

```
void ADC0_IRQHandler(void)
{
    g_Adc16ConversionDoneFlag = true;
}

void EDMA_Callback_0(edma_handle_t *handle, void *param, bool transferDone, uint32_t tcds)
{
    if (transferDone)
    {
        g_Transfer_Done_ch0 = true;
    }
}

void EDMA_Callback_1(edma_handle_t *handle, void *param, bool transferDone, uint32_t tcds)
{
    if (transferDone)
    {
        /* If next line is uncommented, it will start again after 12 transfers finished */
        //EDMA_StartTransfer(&g_EDMA_Handle_1);
        g_Transfer_Done_ch1 = true;
    }
}
```

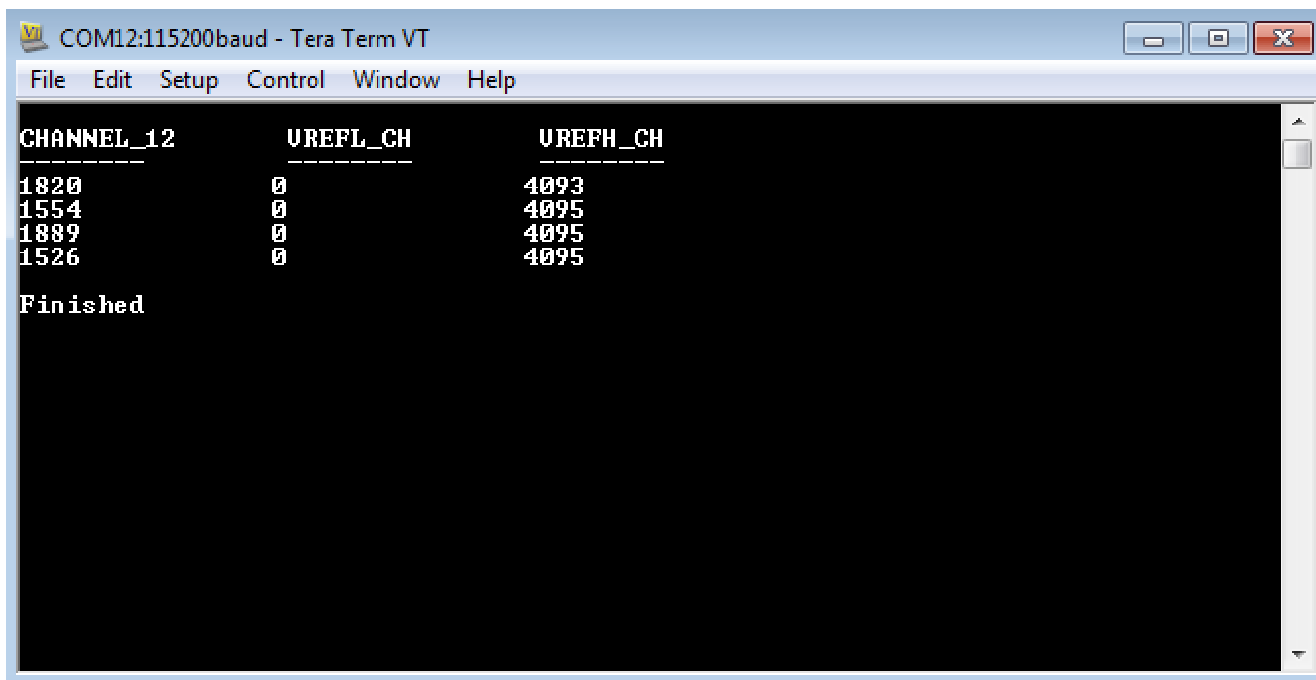
This flags are just used for this implementation and as reference, they can be removed if needed. After Major loop has finished, SDK implementation disable DMA transfers automatically, so noticed that the callback_1 for DMA channel 1 has a line commented, that if uncommented, this application will act as a ADC that load measurements in an internal Ring buffer Indefinitely.

4 ADC Flex Scan mode with DMA

With a basic Print implementation of ADC results we can show the functionality of this example project.

```
uint8_t i = 0;
PRINTF("\r\nCHANNEL_12\t VREFL_CH\t VREFH_CH\r\n--------\t --------\t --------\t\r\n");
while (1)
{
    if (g_Transfer_Done_ch0)
    {
        PRINTF("%d \t\t", g_ADC0_resultBuffer[i++]);
        PRINTF("%d \t\t", g_ADC0_resultBuffer[i++]);
        PRINTF("%d \t\t", g_ADC0_resultBuffer[i++]);
        PRINTF("\r\n");
        g_Transfer_Done_ch0 = false;
        if (g_Transfer_Done_ch1)
        {
            i = 0;
            PRINTF("\r\nFinished\r\n");
            g_Transfer_Done_ch1 = false;
        }
    }
}
```

There are obtained the following results,



Appendix A: Requirements

1. Download page for MCUXpresso IDE: http://www.nxp.com/mcuxpresso
2. Download page for SDK 2.x drivers: https://mcuxpresso.nxp.com/ ,

Go to *Build an SDK,* select your device and click on *Specify Additional Configuration Settings*, verify that you have selected KDS as toolchain and then *Go to SDK Builder.* Click in *Download Now,* accept Term and conditions and download SDK packet. Please check: https://community.nxp.com/docs/DOC-333304

Reference

1. https://mcuoneclipse.com/2017/03/28/mcuxpresso-ide-unified-eclipse-ide-for-nxps-arm-cortex-m-microcontrollers/
2. https://community.nxp.com/docs/DOC-104395
3. www.nxp.com/docs/en/application-note/AN4590.pdf
4. Creating New Projects using installed SDK Part Support, MCUXpresso_IDE_User_Guide.pdf