SMART ARM-based Microcontrollers

AT10843: CPU Usage Demonstration Using DMAC of SAM L22

APPLICATION NOTE

Description

The Direct Memory Access Controller (DMAC) in Atmel® | SMART SAM L22 can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. It enables high data transfer rates with minimum CPU intervention, and frees up CPU time. It contains both a Direct Memory Access engine and a Cyclic Redundancy Check (CRC) engine. With access to all peripherals, the DMAC can handle automatic transfer of data between communication modules.

This application note demonstrates the CPU usage when an application is executed with and without DMA. The analog data from light sensor is sampled with ADC and data is sent to USART. In this application note, the CPU usage is calculated with and without DMA for the data transfer.

Features

This application note covers the following peripheral features:

- DMA data transfer between
  - Peripheral (ADC) to peripheral (USART)
  - Peripheral (ADC) to memory (SRAM)
  - Memory (SRAM) to memory (SRAM)
  - Memory (SRAM) to peripheral (USART)
- Transfer trigger sources
  - Software
  - Peripherals
- Multi-buffer transfer modes by linking multiple descriptors
- Enabling three independent channels with automatic descriptor for each channel
- Fixed priority scheme within each priority level
- 1K beats AHB data transfer in single block transfer
- Multiple addressing modes
  - Static
  - Programmable increment scheme

- Transaction complete interrupt generation
- DMA Event output
- Event system for direct peripheral-to-peripheral communication signaling
- Event triggered ADC conversion for accurate timing
- DMA transfer of conversion result
- CPU usage calculation using System Timer (SysTick)

Table of Contents

1. Abbreviations

ADC	Analog to Digital Converter
ASF	Atmel Software Framework
Atmel Studio	Integrated Development Environment (IDE) for Atmel applications
CDC	Communication Device Class
DMAC	Direct Memory Access Controller
DRE	Data Register Empty
EDBG	Embedded Debugger
EVSYS	Event System
IDE	Integrated Development Environment
Ksps	Kilo samples per second
MCU	Micro Controller Unit
RAM	Random Access Memory
SERCOM	Serial Communication Interface
SysTick	System Timer Tick
USART	Universal Synchronous and Asynchronous Receiver and Transmitter
USB	Universal Serial Bus

2. Pre-requisites

The solutions discussed in this document requires basic familiarity with the following tools.

- Atmel Studio 7 or later
- SAM L22 Xplained Pro
- ASF 3.27 or later

This application note covers the overview of the following peripherals.

- DMAC
- SERCOM – USART
- EVSYS
- ADC
- SysTick

Refer to the product datasheet for better understanding and working of each of the peripherals.

3. Setup

The application is developed for SAM L22 Xplained Pro board using Atmel Studio 7 or later. This chapter covers hardware and software setup required for testing this application.

3.1. Hardware Setup

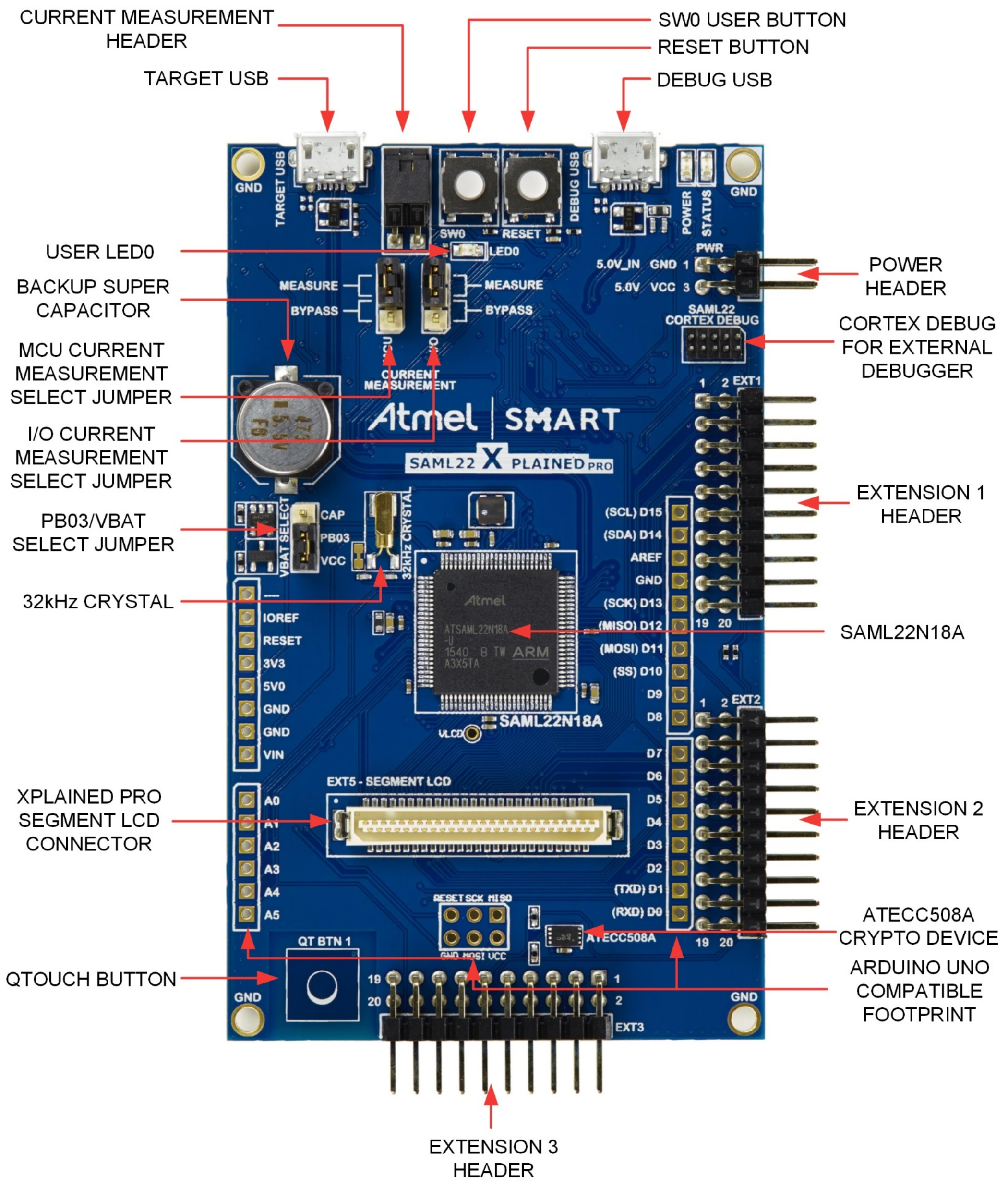
3.1.1. SAM L22 Xplained Pro

The Atmel SAM L22 Xplained Pro evaluation kit is a hardware platform to evaluate the Atmel ATSAML22N18A.

The kit offers a set of features that enables the ATSAML22N18A user to get started with the SAM L peripherals instantly and to understand the steps to integrate the device in a custom design.

This is an evaluation kit that allows connecting multiple external components via a wing connector. A wing is a self contained board that can be connected to the Xplained Pro using a wing connector. The SAM L22 Xplained Pro has three such wing connectors and EXT1 is used in this application.

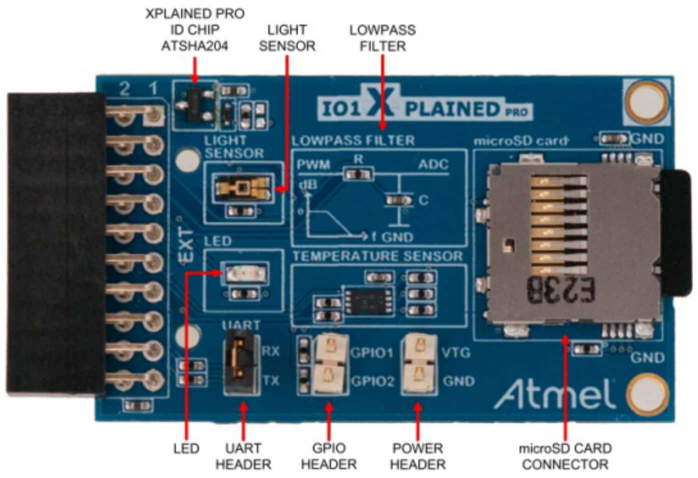
Figure 3-1. SAM L22 Xplained Pro Evaluation Kit Overview



3.1.2. IO1 Xplained Pro Extension Board

Atmel IO1 Xplained Pro extension board is a generic extension board for the Xplained Pro platform. It connects to any Xplained Pro standard extension header on any Xplained Pro MCU board. The extension board utilizes all functions available on the standard Xplained Pro extension header.

Figure 3-2. IO1 Xplained Pro Extension Board



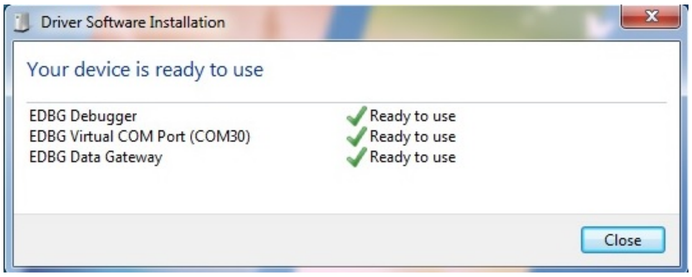
Atmel IO1 Xplained Pro has been designed to be connected to the Xplained Pro header marked EXT1. However, it is compatible with all Xplained Pro EXT headers available on an Xplained Pro board. The pin-out of the respective Xplained Pro evaluation kit is needed to find out which Xplained Pro EXT headers can be used. In this application note, EXT1 has been used.

IO1 Xplained Pro features a 'TEMT6000' light sensor from Vishay Intertechnology, Inc. Pin3 of extension board is utilized for this purpose. The sensor data can be gathered from an ADC pin on Xplained Pro MCU board. In SAM L22 Xplained Pro kit, it is connected to EXT1 header. This application utilizes Light sensor on IO1 Xplained pro board as an analog input to the ADC.

3.2. Software Setup

There are two USB ports on the SAM L22 Xplained Pro board - **DEBUG USB and TARGET USB**. For debug using Embedded debugger **EDBG, DEBUG USB** port has to be connected. When the SAM L22 Xplained Pro kit is connected to the PC, the Windows® Task bar will pop-up a message shown as follows:

Figure 3-3. SAM L22 Xplained Pro Driver Installation

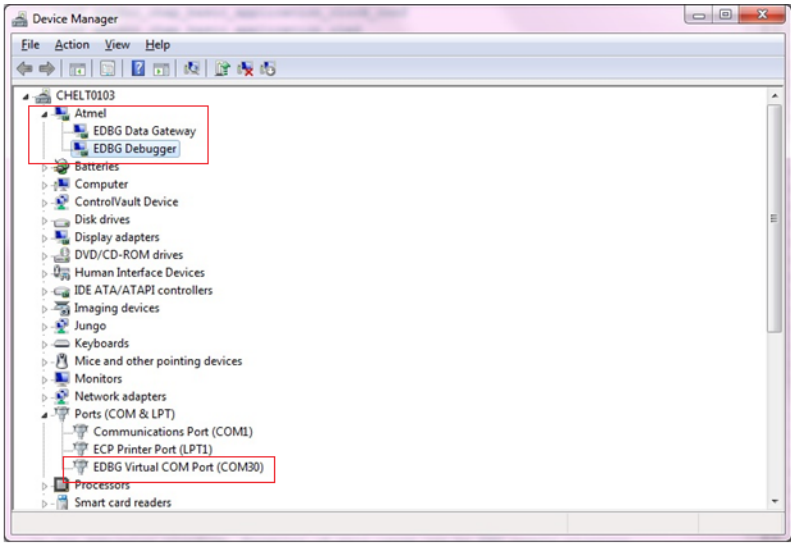

Driver Software Installation

Your device is ready to use

EDBG Debugger	✓ Ready to use
EDBG Virtual COM Port (COM30)	✓ Ready to use
EDBG Data Gateway	✓ Ready to use

If the driver installation is successful, EDBG will be listed in the Device Manager:

Figure 3-4. Successful EDBG Driver Installation


Device Manager

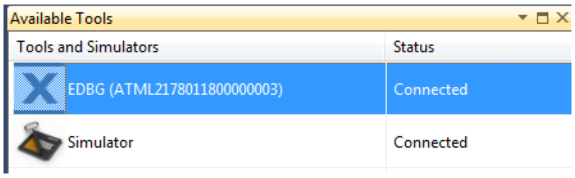
File Action View Help

- CHELT0103
  - Atmel
    - EDBG Data Gateway
    - EDBG Debugger
  - Batteries
  - Computer
  - ControlVault Device
  - Disk drives
  - Display adapters
  - DVD/CD-ROM drives
  - Human Interface Devices
  - IDE ATA/ATAPI controllers
  - Imaging devices
  - Jungo
  - Keyboards
  - Mice and other pointing devices
  - Monitors
  - Network adapters
  - Ports (COM & LPT)
    - Communications Port (COM1)
    - ECP Printer Port (LPT1)
    - EDBG Virtual COM Port (COM30)
  - Processors
  - Smart card readers

To ensure that the EDBG tool is detected in Atmel Studio:

Open Atmel Studio, Go to **View > Available Atmel Tools**. The **EDBG** should get listed in the tools and the tool status should display as **Connected**. This tool status indicates that the tool is communicating properly with Atmel Studio.

Figure 3-5. EDBG under Available Atmel Tools


Available Tools

Tools and Simulators	Status
EDBG (ATML2178011800000003)	Connected
Simulator	Connected

If the tool is not displayed in **Available Tools**, disconnect the tool and reconnect again.

Right click on the tool in the **Available Tools** list, click on **Upgrade**. This will verify if the firmware in the tool is up to date. Click on **Upgrade** to upgrade the firmware to latest version.

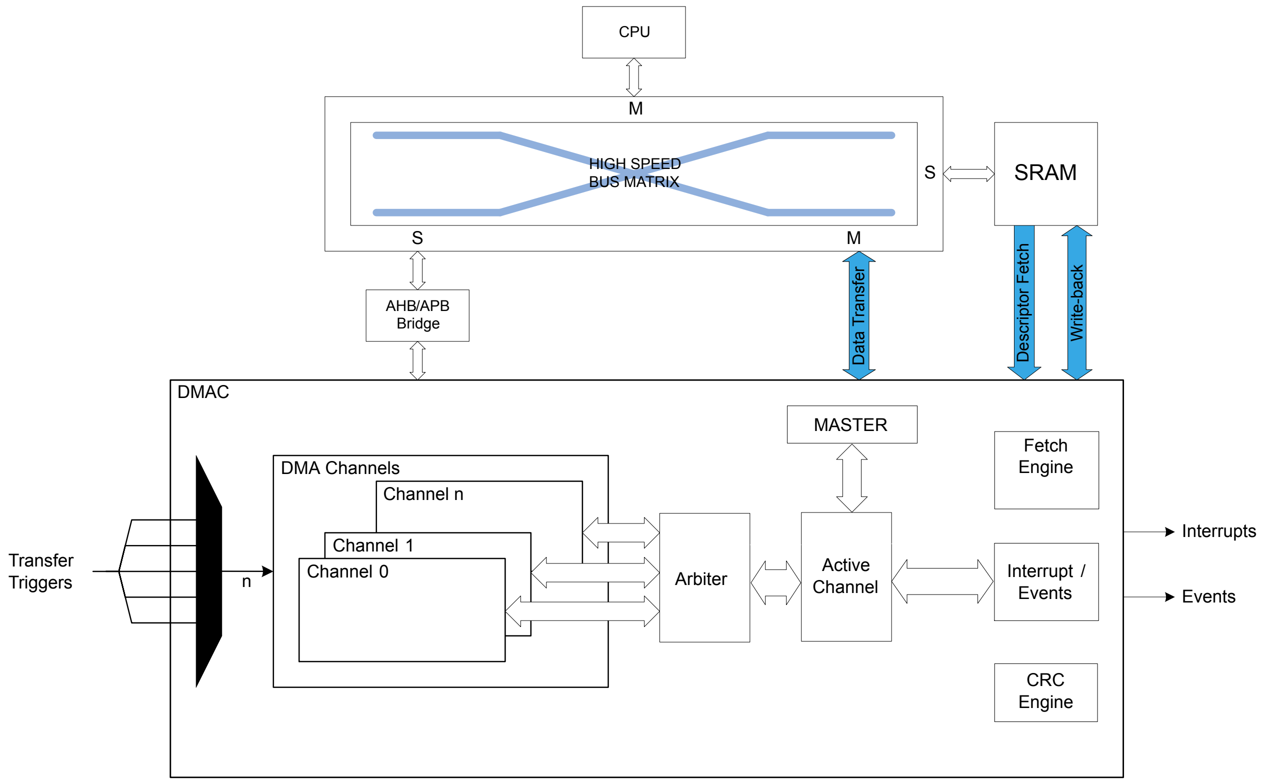
After installing the software successfully, open terminal window (like RealTerm or TeraTerm) with the COM port (**EDBG Virtual COM port**) number available in **Device Manager**.

4. Direct Memory Access Controller

This chapter covers the DMAC features and its working relevant to this application note. Refer the product datasheet for detailed description about its operation and configuration.

4.1. Block Diagram

Figure 4-1. DMAC Block Diagram



4.2. Functional Description

4.2.1. DMAC Basic Operation

The Direct Memory Access Controller (DMAC) contains both a Direct Memory Access engine and a Cyclic Redundancy Check (CRC) engine. The DMAC can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. It enables high data transfer rates with minimum CPU intervention, and frees up CPU time. With access to all peripherals, the DMAC can handle automatic transfer of data between communication modules. This allows the CPU to sleep for longer time and thus reduce the power consumption.

A complete DMA read and write operation between memories and/or peripherals is called a DMA transaction. DMA reads data from the source address before writing to the destination address. A new data is read when the previous write operation is completed. The transaction is initiated by a trigger and uses a DMA channel. The DMA trigger source can be application software, peripheral or events from Event System (EVSYS). Each read and write operations are done in blocks. The size of transfer is controlled by block transfer size and is configured in software. The size of the block can be from 1 to 64K beats. The beat can be byte, half-word or word.

4.2.2. DMAC Channels

The DMA implements 16 channels, enabling 16 independent transfers. Each DMA channel has an individual Transfer control descriptor setting that is stored in SRAM.

The transfer control descriptor defines the source and destination address, source and destination address increment settings, block transfer count, and optional event output condition selection. Source and destination addressing can be static or incremental.

Dedicated I/O registers for each channel is available that controls the trigger mode (peripheral/software), peripheral trigger source type, event input actions and channel priority level settings.

Dedicated write-back memory section is available for each active channel, to maintain the current transfer settings and status.

When enabling multiple channels, 4-level channel priority is supported, and fixed or round-robin scheme is available within each priority level.

4.2.3. DMAC Transfer Operation

Single transaction can be executed (using only one descriptor) or multiple transactions can be executed (using linked descriptor). Single or multiple block transfers can be enabled using the same DMA channel.

When DMA peripheral and respective channel are enabled, the transfer will happen upon receiving the trigger request. The transfer type can be beat, block (group of beats together forms block) or transaction (group of blocks forms transaction).

The channel is automatically disabled when DMA transfer is completed. If a single descriptor is defines for a channel the channel will be disabled when a block transfer is completed. In case of linked descriptors, the channel is disabled once the last descriptor is executed.

4.2.4. Other Features

Channel Suspend and Resume

The channel operation can be suspended or resumed at any time by software, or can be suspended when a selectable block transfer is complete.

Interrupt Request

Interrupt requests can be generated when:

- A transaction is completes
- Selectable block transfer is complete
- DMA controller detects a bus error
- A channel operation is suspended

Event Input

The event input actions are available on the least significant DMA channels. The event can be programmed to trigger:

- Transfers
- Periodic transfers
- Conditional transfers
- To suspend or resume a channel operation

Event Output

Event output selection is available for the least significant DMA channels. The pulse width of an event output from a channel is one AHB clock cycle. Events can be generated when:

- Each AHB data transfer is complete
- Selectable block transfer is complete
- The entire transaction is complete

5. Peripherals Overview

This chapter covers the overview of other peripherals relevant to this application note. Users are requested to refer to respective sections in the product datasheet for more detailed description about their working and configuration.

5.1. Event System (EVSYS)

The Event System (EVSYS) allows autonomous, low-latency, and configurable communication between peripherals. Several peripherals can be configured to emit and/or respond to signals known as events.

The exact condition to generate an event, or the action taken upon receiving an event, is specific to each module. Peripherals that respond to events are called event users. Peripherals that emit events are called event generators. A peripheral can have one or more event generators and can have one or more event users.

Communication is made without CPU intervention and without consuming system resources such as bus or RAM bandwidth. This reduces the load on the CPU and other system resources, compared to a traditional interrupt-based system.

In this application note, EVSYS is configured to use ‘DMA channel 0 transfer complete’ (DMAC CH0) as event generator and ADC start conversion (ADC START) as event user.

5.2. Analog-to-Digital Converter (ADC)

The Analog-to-Digital Converter (ADC) converts analog signals to digital values. The ADC has up to 12-bit resolution, and is capable of converting up to 1MSPS. The input selection is flexible, and both differential and single-ended measurements can be performed. An optional gain stage is available to increase the dynamic range. In addition, several internal signal inputs are available.

ADC measurements can be started by either application software or an incoming event from another peripheral in the device. Both internal and external reference voltages can be used.

The ADC may be configured for 8-, 10-, or 12-bit results, reducing the conversion time. ADC conversion results are provided left- or right-adjusted, which eases calculation when the result is represented as a signed value. It is possible to use DMA to move ADC results directly to memory or peripherals when conversions are done.

In this application note, ADC is configured for 8-bit resolution and uses DMA to transfer ADC result to destination address configured (can be peripheral or memory). Event input from DMA is used to trigger next ADC conversion. Software trigger is used for the case that is implemented without using DMA.

5.3. SERCOM – Serial Communication Interface

The SERCOM serial engine consists of a transmitter and receiver, baud-rate generator and address matching functionality. The transmitter consists of a single write buffer and a shift register. The receiver consists of a two-levels receive buffer and a shift register. The baud-rate generator is capable of running on the GCLK_SERCOMx_CORE clock or an external clock.

The serial communication interface (SERCOM) can be configured to support a number of modes; I2C, SPI, and USART. Once configured and enabled, all SERCOM resources are dedicated to the selected mode.

5.4. SERCOM – USART

The universal synchronous and asynchronous receiver and transmitter (USART) is one of the available modes in the Serial Communication Interface (SERCOM).

A data transmission is initiated by loading the DATA register with the data to be sent. The data in TxDATA is moved to the shift register when the shift register is empty and ready to send a new frame. When the shift register is loaded with data, one complete frame will be transmitted.

The Transmit Complete interrupt flag in the Interrupt Flag Status and Clear register (INTFLAG.TXC) is set, and the optional interrupt is generated, when the entire frame plus stop bit(s) have been shifted out and there is no new data written to the DATA register.

The DATA register should only be written when the Data Register Empty flag in the Interrupt Flag Status and Clear register (INTFLAG.DRE) is set, which indicates that the register is empty and ready for new data.

USART can generate DMA request when the transmit buffer (TX DATA) is empty. The request is cleared when DATA is written.

In this application, EDGB CDC (SERCOM4) is utilized to transfer ADC result data to terminal.

5.5. The System Timer (SysTick)

The System Timer is a 24-bit timer that extends the functionality of both the processor and the NVIC. Refer to the Cortex®-M0+ Technical Reference Manual for details (www.arm.com).

The timer consists of:

- A control and status register (SYST_CSR). This configures the SysTick clock, enables the counter, enables the SysTick interrupt, and indicates the counter status.
- A counter reload value register (SYST_RVR). This provides the wrap value for the counter.
- A counter current value register (SYST_CVR)

When enabled, the timer counts down from the value in SYST_CVR. When the counter reaches zero, it reloads the value in SYST_RVR on the next clock edge. It then decrements on subsequent clocks. This reloading when the counter reaches zero is called wrapping. Interrupt can be enabled which triggers for each time counter wrap around.

In this application, counter is loaded with maximum count value and is used to take time stamp while calculating the CPU utilization. SysTick runs at processor clock as source.

6. Example Implementation

This chapter explains the application implementation in detail.

The objective of this application note is to demonstrate the features listed in this document and its configuration. In addition to that, CPU utilization is calculated, when application is implemented with and without DMA. This highlights the DMAC usage in reducing the CPU load.

In the example implementation, ADC converts input analog signal to digital value and the result is transferred to USART. Light sensor in IO1 Xplained Pro is given as an input to ADC via EXT1 header.

This application is implemented in four different scenarios to cover the objective (i.e. with and without DMAC) and user will need to select the case accordingly. Separate source files have been implemented for each case. Based on the compiler option selected in conf_dma.h file, the main application will get compiled for each case accordingly. The following lines explains about each case in detail.

6.1. Peripheral to Peripheral Transfer with DMAC (ADC to USART)

The compiler option to enable this transfer type is ADC_DMAC_USART. In this case, ADC result is directly written to USART DATA register to illustrate the peripheral to peripheral DMA transfer type.

Note: File to be referred **adc_dmac_usart.c**.

6.1.1. Application Configuration and Implementation

DMAC is configured to trigger a data transfer to the destination address configured when ADC RESULT is ready (peripheral trigger source). The destination address configured here is USART DATA register address and source is ADC RESULT register address. DMA source and destination address is static in this case, as both the register addresses are fixed. The descriptor is configured in **setup_transfer_descriptor()** as below:

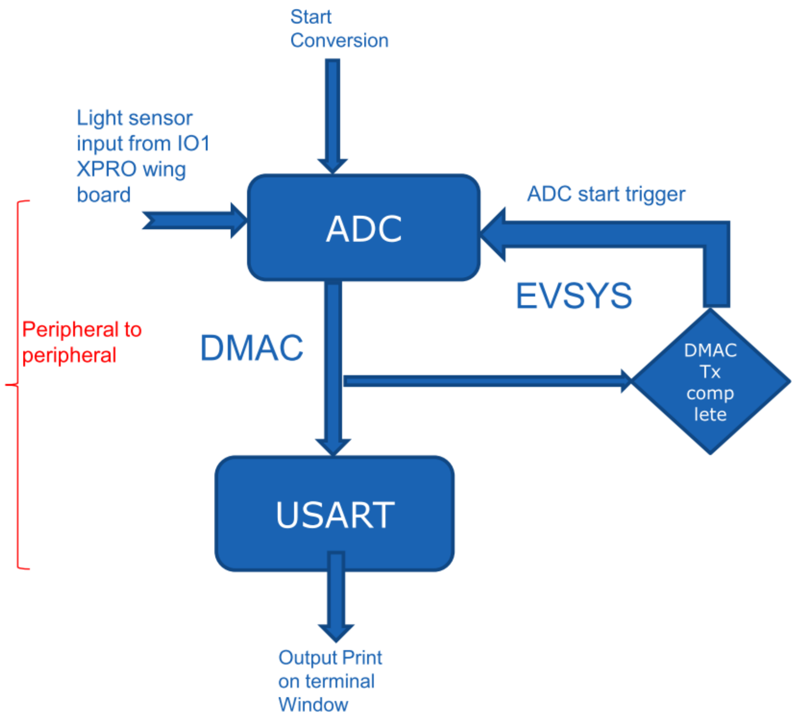
```
{
    /* DMA descriptor configuration setup */
    struct dma_descriptor_config descriptor_config;

    /* Get default configuration */
    dma_descriptor_get_config_defaults(&descriptor_config);

    /* Set beat size as byte */
    descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    /* Set block count as 1024 beats */
    descriptor_config.block_transfer_count = BLOCK_COUNT;
    /* Trigger interrupt once block transfer is complete */
    descriptor_config.block_action = DMA_BLOCK_ACTION_INT;

    switch (descriptor_num){

    case DMAC_DESCRIPTOR1_ID:
        /* Source address is static as it is ADC result register */
        descriptor_config.src_increment_enable = false;
        /*
         * Enable event for every beat transfer (I.e. byte in this case)
         * Every byte transfer occurs for each sample from ADC.
         * I.e. Event is triggered for every ADC result ready.
         */
        descriptor_config.event_output_selection = DMA_EVENT_OUTPUT_BEAT;
        /* Destination address is static as it is USART DATA register */
        descriptor_config.dst_increment_enable = false;
        /* Set source address as ADC RESULT register */
        descriptor_config.source_address = (uint32_t)(&ADC->RESULT.reg);
        /* Set destination address as USART DATA register */
        descriptor_config.destination_address = (uint32_t)(&EDBG_CDC_MODULE-
>USART.DATA.reg);
        break;

    default:
        Assert(false);
        break;
    }

    /* Create descriptor */
    dma_descriptor_create(descriptor, &descriptor_config);
}
```

For each trigger, a byte will get transferred as Beat size is configured as byte. Event output from DMA is enabled which will get generated up on each DMA transfer complete. DMAC Channel zero is used for this case and the configuration is done in **configure_dma_resource()** as below:

```
        case DMAC_CHANNEL0_ID:
            /* Trigger is enabled for each beat transfer */
            config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
            /* Peripheral trigger source is ADC result ready */
            config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
            /* Generate event once DMA transfer is done */
            config.event_config.event_output_enable = true;
            break;
```

ADC is configured as event user which will start conversion upon receiving event signal from DMAC via Event System (EVSYS).The first ADC conversion is triggered by software trigger. Once input is sampled and result is ready, it triggers DMA transfer from ADC RESULT to USART DATA register. The next ADC conversion is triggered by the event signal from DMA upon completing the transfer to USART DATA register and the cycle continues. Overall the operation is done as shown in the figure below.

Figure 6-1. DMA Peripheral to Peripheral Transfer



The whole operation is done using DMAC and EVSYS without interrupting CPU. DMAC block transfer size is configured as 1024 bytes (BLOCK_COUNT) and an interrupt is configured to flag once block transfer is complete. Once block transfer is completed, DMAC channel gets disabled automatically.

6.1.2. CPU Utilization Calculation

As explained in CPU Utilization Calculation on page 25, time stamp from SysTick is taken before starting first ADC conversion in main().

On completing 1024 byte transfer from ADC to USART, DMAC channel zero **block transfer complete** interrupt call back is called. A flag is set and time stamp is taken to indicate transfer complete (as follows):

```
static void dmac_calback_channel0(struct dma_resource *const resource)
{
    #if defined (ENABLE_PORT_TOGGLE)
    /* Use oscilloscope to probe the pin. */
    port_base->OUTTGL.reg = (1UL << PIN_PA14 % 32 );
    #endif
    /* Indicate DMA transfer has been completed */
    adc_dma_transfer_is_done = true;
    /* Get time stamp */
    time_stamp2 = SysTick->VAL;
}
```

From the time stamp, the number of cycles taken to complete the transfer is calculated. During the DMA transfer, the idle_loop_count is incremented in main loop. This will give the count of CPU idle time during the data transfer from peripheral to peripheral. After completion of the DMA transfer, the code enters an infinite loop and no other tasks including idle task is executed.
Note: Refer CPU Utilization Analysis Between Different Cases on page 29 for detailed description about the CPU utilization calculation from the results observed.

6.2. Peripheral to Memory and Memory to Peripheral Transfer with DMAC (ADC to SRAM and SRAM to USART)

The compiler option to enable this case is ADC_DMAC_MEM_MEM_USART. In this case, three DMAC channels have been used to demonstrate each transfer type. As explained in SRAM to SRAM Transfer Type on page 28, the purpose of having Memory to Memory type DMA transfer is for demonstration purpose and the application does not need this for its proper working.

Note: File to be referred for this case is **adc_dmac_mem_mem_usart.c**.

6.2.1. Application Configuration and Implementation

Channel zero is used to transfer BLOCK_COUNT (i.e. 1024 bytes in this example) number of beats from ADC RESULT register (peripheral) to SRAM buffer (Memory).

Channel0 configuration (Peripheral to Memory):

As explained in Peripheral to Peripheral Transfer with DMAC (ADC to USART) on page 16, DMAC channel 0 is configured for peripheral trigger from ADC RESULT ready. The next ADC conversion is triggered by event output from DMA channel 0 up on completing each beat transfer as follows:

```
case DMAC_CHANNEL0_ID:
    /* Trigger is enabled for each beat transfer */
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
    /* Peripheral trigger source is ADC result ready */
    config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
    /* Generate event once DMA transfer is done */
    config.event_config.event_output_enable = true;
    break;
```

DMAC block transfer complete interrupt is enabled which gets generated up on completing 1024 bytes from ADC to SRAM buffer. As the SRAM buffer will need to store 1024 bytes samples from ADC, the destination address is incremented (which is the default configuration in ASF). The source address is static as it is ADC RESULT register and the descriptor is linked to the channel 1 descriptor (dma_adc_descriptor2) for next channel operation as done below:

```
    /* Set beat size as byte */
    descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    /* Set block count as 1024 beats */
    descriptor_config.block_transfer_count = BLOCK_COUNT;
    /* Trigger interrupt once block transfer is complete */
    descriptor_config.block_action = DMA_BLOCK_ACTION_INT;

    switch (descriptor_num){

    case DMAC_DESCRIPTOR1_ID:
        /* Source address is static as it is ADC result register */
        descriptor_config.src_increment_enable = false;
        /*
         * Enable event for every beat transfer (I.e. byte in this case)
         * Every byte transfer occurs for each sample from ADC.
         * I.e. Event is triggered for every ADC result ready.
         */
        descriptor_config.event_output_selection = DMA_EVENT_OUTPUT_BEAT;

        /* Set source address as ADC RESULT register */
        descriptor_config.source_address = (uint32_t)(&ADC->RESULT.reg);
        /*
         * Set destination address as adc_result buffer in RAM.
         * NOTE : destination address increment is true as per default
configuration.
         */
        descriptor_config.destination_address = (uint32_t)(adc_result) + sizeof
(adc_result);
        /* Link to next descriptor */
        descriptor_config.next_descriptor_address = (uint32_t)
(&dmac_adc_descriptor2);

        break;
```

Once 1024 bytes samples get transferred from ADC to SRAM buffer (adc_result[]), dmac_channel0_callback() is called where the channel1 transfer is triggered.

```
 /*! \brief DMA Channel0 call back */
static void dmac_calback_channel0(struct dma_resource *const resource)
{

    #if defined (ENABLE_PORT_TOGGLE)
        /*      Use oscilloscope to probe the pin. */
        port_base->OUTTGL.reg = (1UL << PIN_PA14 % 32 );
    #endif
    /* Trigger channel1 transfer */
    dma_trigger_transfer(&dmac_adc_channel1);

}
```

Channel1 configuration (Memory to Memory):

Channel 1 is configured with software trigger and transfer type is transaction. I.e. once software triggers the transfer, complete ADC result stored in one SRAM buffer (adc_result) is transferred to another adc_result_copy buffer stored in SRAM. This retains the default configuration done in ASF. So there is no change needed at the application code.

```
        case DMAC_CHANNEL1_ID:
        /*
         * Retain default configuration for channel 1
         * I.e Transaction trigger transfer type, software trigger
         * source with even output disabled/
         */
        break;
```

The descriptor contains the source and destination address of two different SRAM buffers and both addresses are incremental (default configuration in ASF). The channel 2 descriptor (dmac_adc_descriptor3) is linked to this descriptor which would point to channel 3 at the end of channel 2 block transfer complete.

```
    case DMAC_DESCRIPTOR2_ID:
        /*
         * Set source address as adc_result buffer in RAM.
         * NOTE: source address increment is true as per default configuration.
         */
        descriptor_config.source_address = (uint32_t)(adc_result) + sizeof
(adc_result);
        /*
         * Set destination address as adc_result_copy buffer in RAM.
         * NOTE : destination address increment is true as per default
configuration.
         */
        descriptor_config.destination_address = (uint32_t)(adc_result_copy) +
sizeof (adc_result_copy);
        /* Link to other descriptor */
        descriptor_config.next_descriptor_address = (uint32_t)
(&dmac_adc_descriptor3);
        break;
```

Channel 1 transfer is triggered at the channel 0 transfer complete callback as explained already. Once Channel 1 transfer is done, channel 2 is enabled to start the next conversion (as below) which is explained in forthcoming sections.

```
/*! \brief DMA Channel1 call back */
static void dmac_calback_channel1(struct dma_resource *const resource)
{
    /* Enable and start channel2 transfer */
    dma_start_transfer_job(&dmac_adc_channel2);
}
```

Channel2 configuration (Memory to Peripheral):

Channel 2 is configured to have peripheral trigger and beat transfer type. A byte from SRAM buffer (adc_result_copy) should be written to the USART DATA register whenever it is empty. I.e. Whenever USART DATA register is empty (DRE) and is ready for new data to be written, it triggers a DMA transfer from source to destination over the channel2.

```
    case DMAC_CHANNEL2_ID:
        /* Triggers for every beat */
        config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
        /* Peripheral trigger source is USART data register empty */
        config.peripheral_trigger = SERCOM4_DMAC_ID_TX;
        break;
```

The destination address in the descriptor is incremental (default configuration in ASF) and destination address is static as it is USART DATA register. It does not point to any next descriptor as there is not any transfer going to occur.

```
    case DMAC_DESCRIPTOR3_ID:
        /* Set destination increment as static as it is USART DATA register */
        descriptor_config.dst_increment_enable = false;
        /*
         * Set source address as adc_result_copy buffer in RAM.
         * NOTE: source address increment is true as per default configuration.
         */
        descriptor_config.source_address = (uint32_t)(adc_result_copy) + sizeof
(adc_result_copy);
```

```
        /* Set destination address as USART DATA register */
        descriptor_config.destination_address = (uint32_t)(&EDBG_CDC_MODULE-
>USART.DATA.reg);
        break;
```

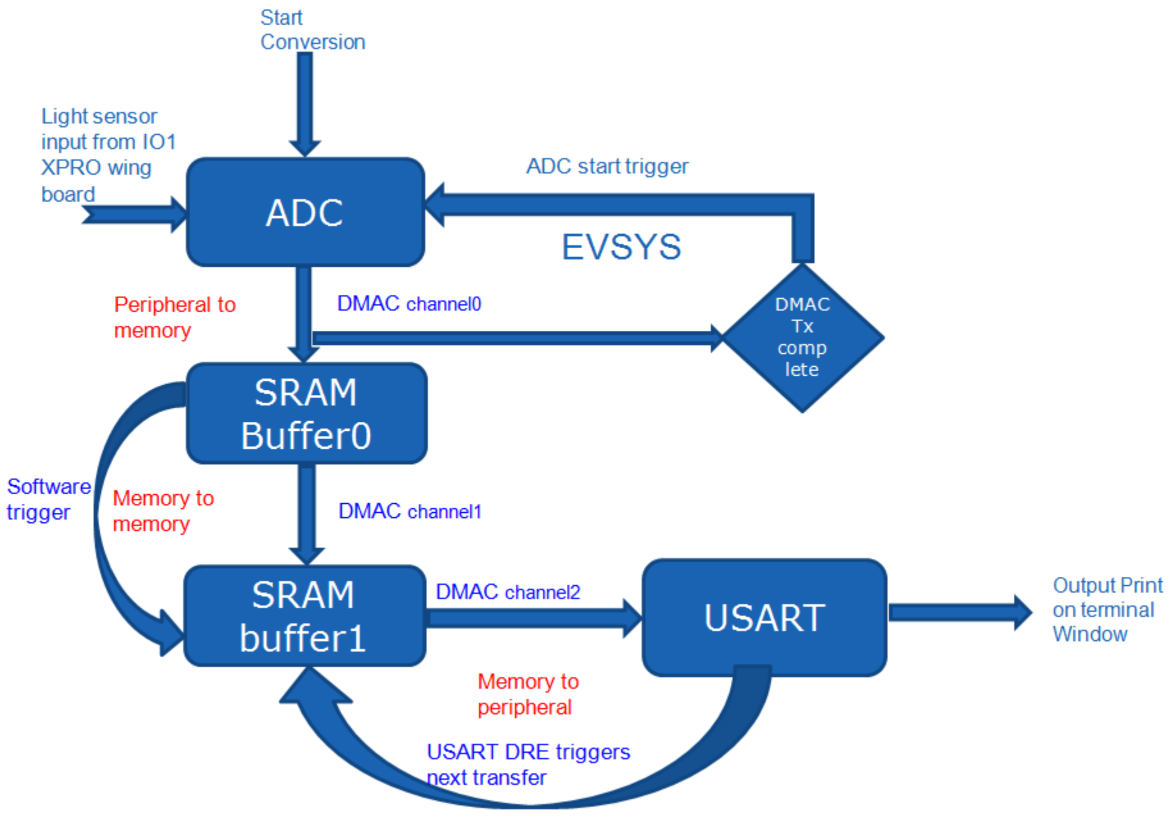
Unlike other channels, this channel should be enabled at the end of channel 1 transfer complete. The reason is the USART DRE is always set as there is not any previous communication occurs. So if this channel is enabled during the initialization, as USART DRE is already set, the DMA transfer will start immediately on channel 2 which results in wrong operation.

Once Channel 2 is enabled in the channel 1 callback, the data is sent from SRAM to USART for each DRE from USART. After it completes the transfer, a flag is set to indicate end of complete transfer and the time stamp is taken for CPU utilization calculation.

```
/*! \brief DMA Channel2 call back */
static void dmac_calback_channel2(struct dma_resource *const resource)
{
    /* Indicate DMA transfer has been completed */
    adc_dma_transfer_is_done = true;
    /* Get time stamp */
    time_stamp2 = SysTick->VAL;
}
```

Overall the application works as illustrated as follows.

Figure 6-2. DMAC Peripheral – Memory – Memory – Peripheral Transfer



6.2.2. CPU Utilization Calculation

The time stamp is taken at the DMAC channel2 call back and the idle_loop_count is noted. This would add some more overhead as it is interrupted by three different callbacks for each block transfer complete of a channel.

Note: Refer CPU Utilization Analysis Between Different Cases on page 29 for detailed description about the CPU utilization calculation from the results observed.

6.3. Peripheral to Peripheral Transfer without DMAC (ADC to USART)

This option is enabled by defining ADC_NO_DMAC_USART. In this case, the above mentioned scenarios are implemented through interrupt handling without using DMA. This is done to demonstrate the DMAC usage in reducing the CPU load.

Note: File to be referred for this case is **adc_no_dmac_usart.c**.

6.3.1. Application Configuration and Implementation

ADC interrupt is triggered for RESULT ready and the software trigger mode is chosen to start the conversion in **configure_adc()** function as follows:

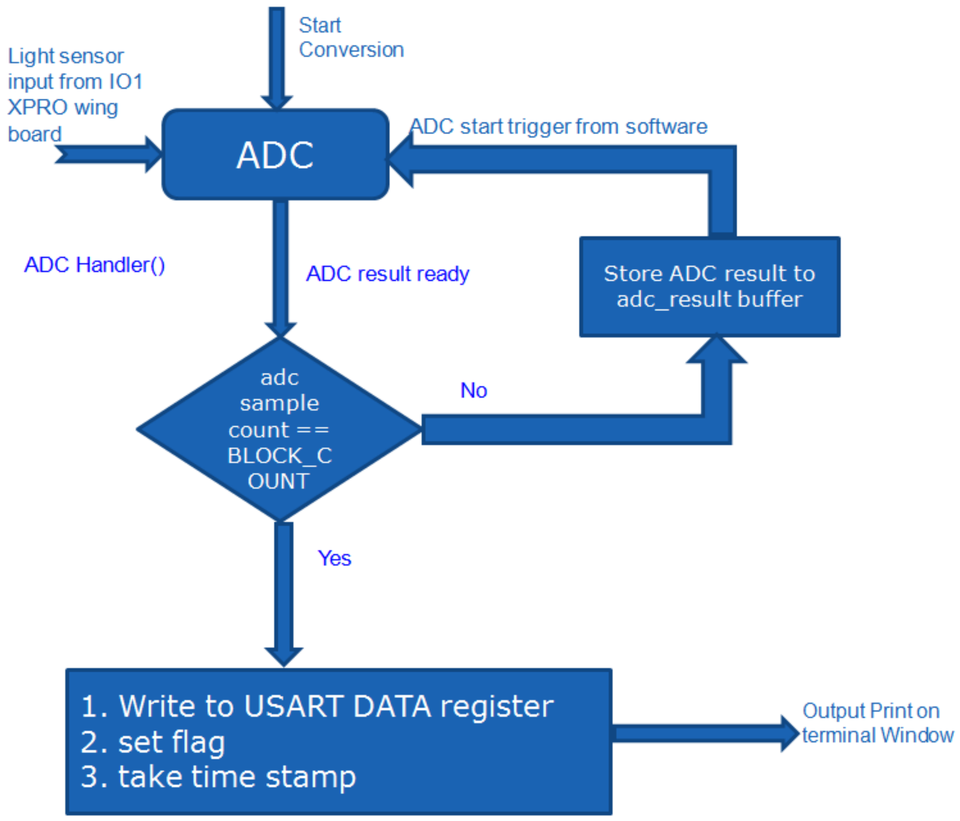
```
    /* Enable ADC Result ready interrupt */
    adc_interrupt_enable(&adc_instance,ADC_INTFLAG_RESRDY);
    /* Enable ADC module interrupt in NVIC */
    system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_ADC);
```

The first conversion is done in the main and the time stamp1 is taken. Once result is ready, ADC interrupt handler is called. In the handler, the number of ADC samples is counted through a count variable adc_sample_count. Until 'adc_sample_count' value reaches the BLOCK_COUNT (i.e.1024 bytes), the data is stored in a buffer (adc_result), and the next ADC conversion is triggered from software. Once it reaches the BLOCK_COUNT, ADC is disabled and further conversion is stopped. A flag is also set to indicate transfer is done and the data is sent to USART data register. Time stamp is also taken at this time to find CPU utilization.

```
static void _adc_interrupt_handler(void)
{
    /* ADC base address */
    Adc *const adc_hw = adc_instance.hw;

    /* get interrupt flags */
    uint32_t flags = adc_hw->INTFLAG.reg;

    /* Check if the all the samples has been done by ADC */
    if (adc_sample_count == BLOCK_COUNT){

        /* Disable ADC */
        adc_hw->CTRLA.reg &= ~ADC_CTRLA_ENABLE;

        /* Write samples to USART */
        usart_write_data(&usart_instance,adc_result,BLOCK_COUNT);

        /* Indicate conversion has been done */
        adc_conv_done = true;
        /* Get the time stamp from SysTick */
        time_stamp2 = SysTick->VAL;

    }else if (flags & ADC_INTFLAG_RESRDY) {
        /* Clear ADC interrupt */
        adc_hw->INTFLAG.reg = ADC_INTFLAG_RESRDY;
        /* Store ADC result to RAM buffer */
        adc_result[adc_sample_count] = adc_hw->RESULT.reg;
        /* Count the number of samples taken so far */
        ++adc_sample_count;
        /* Trigger next ADC conversion */
        adc_start_conversion(&adc_instance);
    }
} /* End of ADC Hander */
```

Over all application flow would work as illustrated in the following figure.

Figure 6-3. ADC to USART without DMAC


Start Conversion
Light sensor input from IO1 XPRO wing board
ADC
ADC start trigger from software
ADC Handler()
ADC result ready
Store ADC result to adc_result buffer
adc sample count == BLOCK_COUNT
No
Yes
1. Write to USART DATA register
2. set flag
3. take time stamp
Output Print on terminal Window

6.3.2. CPU Utilization Calculation

The same logic is used to calculate the CPU utilization except that in this case, interrupt is enabled and does not use DMA. The ADC result from adc_result buffer transfer to USART DATA register is managed by the SERCOM4 Handler.

The number of transfer is counted by the adc_sample_count. Once it reaches BLOCK_COUNT, time stamp and idle_loop_count is noted to calculate CPU utilization as in CPU Utilization Calculation on page 25.

6.4. Peripheral to Memory and Memory to Peripheral Transfer without DMAC (ADC to SRAM and SRAM to USART)

This option is enabled through ADC_NO_DMAC_MEM_MEM_USART. As mentioned in Peripheral to Peripheral Transfer without DMAC (ADC to USART) on page 22, this is the counterpart implementation of ADC_DMAC_MEM_MEM_USART which is done to demonstrate the DMAC usage in reducing the CPU load.

Note: File to be referred for this case is **adc_no_dmac_mem_mem_usart.c**.

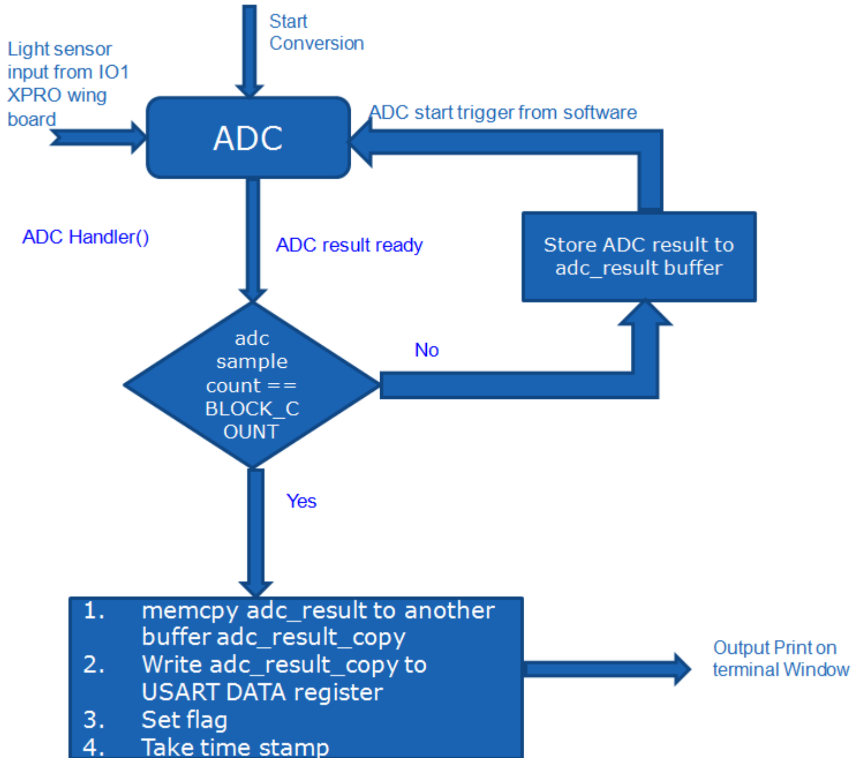
6.4.1. Application Implementation and Configuration

This scenario is same as Peripheral to Memory and Memory to Peripheral Transfer without DMAC (ADC to SRAM and SRAM to USART) on page 23 except that memory copy to another buffer adc_result_copy is done which will add some over overhead to the application.

```
    /* Check if the all the samples has been done by ADC */
    if (adc_sample_count == BLOCK_COUNT){
        /* Disable ADC */
        adc_hw->CTRLA.reg &= ~ADC_CTRLA_ENABLE;

        /* Copy adc result to another buffer */
        memcpy_ram2ram(adc_result_copy,adc_result,BLOCK_COUNT);
        /* Write samples to USART */
        usart_write_data(&usart_instance,adc_result_copy,BLOCK_COUNT);

        /* Indicate conversion has been done */
         adc_conv_done = true;
        /* Get the time stamp from SysTick */
        time_stamp2 = SysTick->VAL;

    }else if (flags & ADC_INTFLAG_RESRDY) {
        /* Clear ADC interrupt */
        adc_hw->INTFLAG.reg = ADC_INTFLAG_RESRDY;
        /* Store ADC result to RAM buffer */
        adc_result[adc_sample_count] = adc_hw->RESULT.reg;
        /* Count the number of samples taken so far */
        ++adc_sample_count;
        /* Trigger next ADC conversion */
        adc_start_conversion(&adc_instance);
    }
```

The application block diagram is shown as follows.

Figure 6-4. ADC – SRAM – SRAM – USART Transfer without DMAC


Start Conversion
Light sensor input from IO1 XPRO wing board
ADC
ADC start trigger from software
ADC Handler()
ADC result ready
Store ADC result to adc_result buffer
adc sample count == BLOCK_COUNT
No
Yes
1. memcpy adc_result to another buffer adc_result_copy
2. Write adc_result_copy to USART DATA register
3. Set flag
4. Take time stamp
Output Print on terminal Window

6.4.2. CPU Utilization Calculation

The CPU utilization is similar as done in CPU Utilization Calculation on page 23.

Note: Refer CPU Utilization Analysis Between Different Cases on page 29 for detailed description about the CPU utilization calculation from the results observed.

6.5. CPU Utilization Calculation

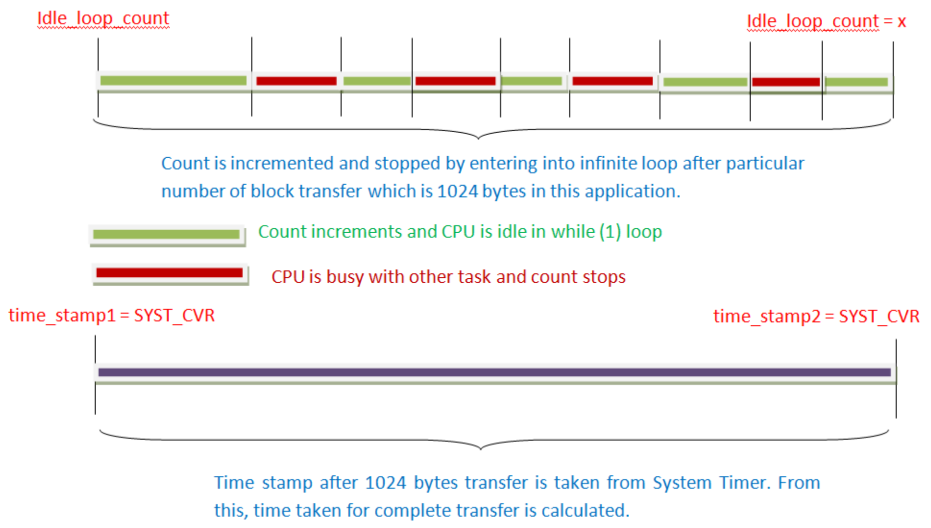
This section covers the logic implemented in this application to calculate the CPU utilization.

For calculating the CPU utilization, we need to measure total time taken for executing the data transfer routine. This is measured using the SysTick timer.

We also need to measure how much time CPU is idle task when executing the above said routine. This is measured by incrementing a variable (‘idle_loop_count’) whenever the CPU is idle. The idle counter value is converted to time scale by multiplying the count value with the time taken to increment once.

Both the total time taken by the data transfer routine and idle counter is measured for fixed number of data transfer as shown in the following figure. In this test, it is 1024 byte transfer.

Figure 6-5. CPU Utilization Calculation


Idle_loop_count
Idle_loop_count = x
Count is incremented and stopped by entering into infinite loop after particular number of block transfer which is 1024 bytes in this application.
Count increments and CPU is idle in while (1) loop
CPU is busy with other task and count stops
time_stamp1 = SYST_CVR
time_stamp2 = SYST_CVR
Time stamp after 1024 bytes transfer is taken from System Timer. From this, time taken for complete transfer is calculated.

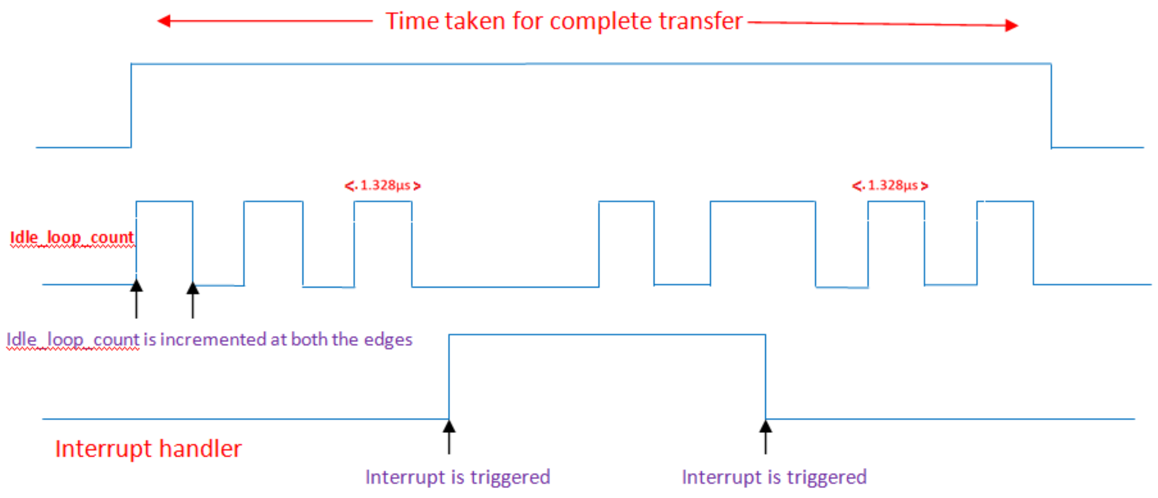
The number of cycles taken (cycles_taken) to complete the transaction can be calculated from the time stamp taken using SysTick. As SysTick runs at processor clock, the time taken for total transaction can be calculated from the cycles taken and the CPU clock frequency of the application as below.

Time taken to complete transaction = (cycles_taken/CPU clock frequency)

The idle_loop_count represents the number of times the code enters idle task. This can be used to derive the time that CPU is idle during complete transaction. To convert this count value to time scale, the time taken for each count increment should be known.

For this purpose, in the application code, two separate port pins are toggled in the idle loop and in the interrupt handler. Whenever the code enters interrupt handler, the idle loop count stops and pin toggled inside idle loop stays at same level. When the code comes out of handler, the pin toggled inside handler stays at same level and the idle loop pin starts toggle. The time taken for single toggling is calculated after removing the time taken for handler execution. This is illustrated as follows.

Figure 6-6. Calculation of Time Taken for a Single idle_loop_count Increment

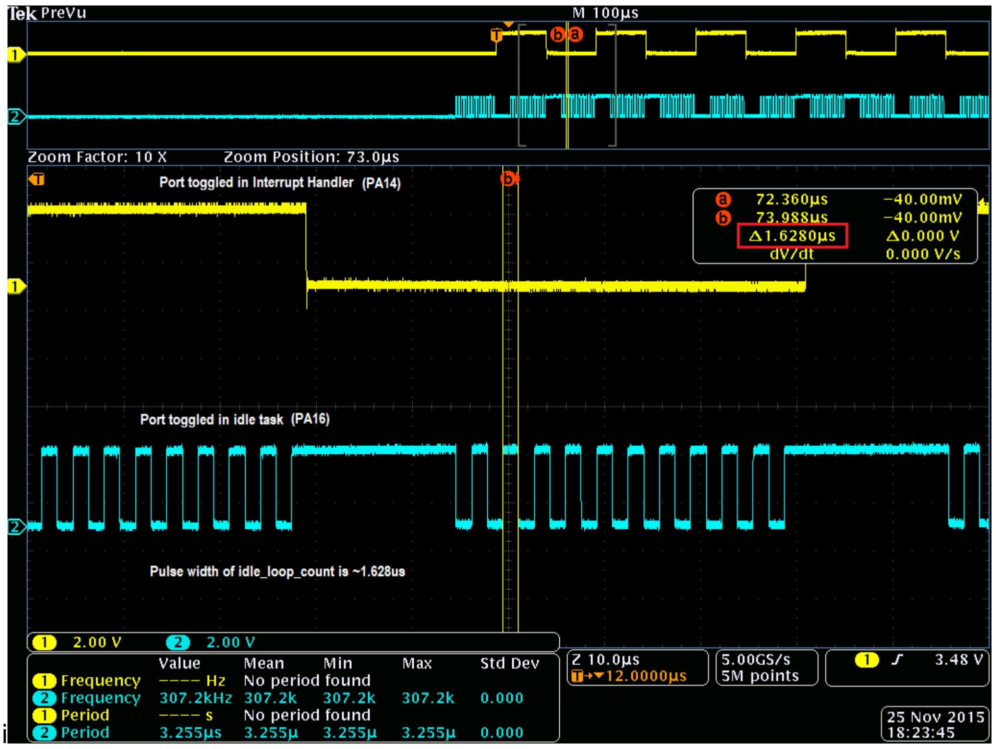


From oscilloscope, the time taken for each count increment is calculated to be ~1.628µs (as shown in the following figure). The count value when multiplied with the pulse width (i.e. 1.628µs) will give the time CPU spends inside idle task.

Note:

1. The duration of idle_loop_count and CPU idle time can be measured by probing port pins PA16 and PA14 respectively.
2. The width of idle_loop_count pulse is the time taken to increment one idle count value when there is no interrupt triggered.

Figure 6-7. Oscilloscope Shot of Idle Task


Tek PreVu
M 100µs
Zoom Factor: 10 X
Zoom Position: 73.0µs
Port toggled in Interrupt Handler (PA14)
Port toggled in idle task (PA16)
Pulse width of idle_loop_count is ~1.628us

```
a  72.360µs    -40.00mV
b  73.988µs    -40.00mV
  Δ1.6280µs     Δ0.000 V
  dV/dt         0.000 V/s

1  2.00 V    2  2.00 V

             Value      Mean     Min      Max      Std Dev
1 Frequency  ---- Hz    No period found
2 Frequency  307.2kHz   307.2k   307.2k   307.2k   0.000
1 Period     ---- s     No period found
2 Period     3.255µs    3.255µ   3.255µ   3.255µ   0.000

Z 10.0µs         5.00GS/s      1  ƒ  3.48 V
T→▾12.0000µs     5M points

25 Nov 2015
18:23:45
```

The CPU utilization analysis for each case has been done in CPU Utilization Analysis Between Different Cases on page 29.

Note: The ideal expectation is that the idle loop count should be more when using DMA than when not using it. Because, when using DMA, the CPU is not interrupted and idle task can be executed in parallel. But in practical, this cannot be the case. The reason is that, DMA will take lesser time to complete the transfer. In case of using interrupt method; it takes more time to complete the transaction. So sometimes, the time that code can spend for ideal task would be lesser for DMAC case and the idle_loop_count value can be lesser than when not using DMA. To avoid such confusion, the time taken for completing the transfer is also taken using system timer and the ratio of both is used to calculate the CPU utilization.

7. Application Limitations

7.1. USART Baudrate and ADC Sampling Frequency

In DMAC usage case, as explained already, ADC RESULT is directly written to USART DATA register. The DMA triggers ADC next conversion immediately once data is written to USART. This causes data loss on terminal window if the usart baud rate is lesser than the ADC conversion time. To avoid this, ADC is configured with lowest possible frequency and USART is configured with maximum possible baudrate.

For ADC:

The rate of conversion of ADC clock depends on the GCLK_ADC (i.e. 16MHz) and it's prescalar which is 64 in this case.

ADC clock frequency = 16MHz/64 = 250kHz ~= 4μs

Conversion time = 8 cycles (8-bit resolution) + 1 cycle (Sampling time) = 9 * 4μs ~= **36μs**

For USART:

As per section 'Baud Rate Equations' in the product datasheet: f_{baud} should be ≤ f_{ref}/S

For Asynchronous Arithmetic mode number of samples per bit (S) = 16

f_{ref} = 16MHz

So, maximum possible baudrate = 16MHz/16 = 500000

Baud rate configured = 460800 (i.e. 460800 bits sent in = 1s)

For 10 bit, it takes = (10/460800) ~= **21.7μs**

So, setting 460800 baudrate is advisable. Because once ADC sample is ready for every 36μs. USART would have sent the previous data in 21.7μs and waits for next ADC result without any data loss.

Note:

1. The 10 bits comes from USART data frame - Start bit **(1)** + Data bit **(8)** + Stop bit **(1)**.
2. Refer the device datasheet, for more details on timing calculations on ADC and USART.

7.2. SRAM to SRAM Transfer Type

For the cases ADC_DMAC_MEM_MEM_USART and ADC_NO_DMAC_MEM_MEM_USART, the transfer type Memory to Memory. I.e. Copy of adc result from one SRAM buffer (adc_result) to another SRAM buffer (adc_result_copy) is done for demonstration purpose. This application does not demand this need to make it work properly.

8. CPU Utilization Analysis Between Different Cases

After programming the firmware successfully, the results can be seen in the terminal window. The result contains the ADC result data, number of cycles taken and idle loop count in 'hex' format. This chapter explains how to derive CPU usage for each case from the results observed. The calculation is done as explained in CPU Utilization Calculation on page 25.

Note: The results shown in this application note taken are with the following conditions. The resulting idle_loop_count and cycles_taken will vary with the optimizations, frequency or any change in the application code.

- Optimization set to zero (-O0)
- Port toggling function (ENABLE_PORT_TOGGLE) is enabled in the conf_dma.h
- OSC16M internal oscillator is used with frequency selected as 16MHz

8.1. CPU Frequency Calculation

To find the time taken, CPU frequency needs to be known. This application runs at internal 16MHz (OSC16M). The accuracy on internal RC oscillator can be taken from **Electrical Characteristics** of the product datasheet. The accuracy of RC in the board used for testing was calculated to be 15.97MHz. This is done by giving the main clock i.e. GCLK0 (which runs at 16MHz) output to I/O pin for the device tested using the snippet below.

```
int main (void)
{
    struct system_pinmux_config pin_conf;
    system_pinmux_get_config_defaults(&pin_conf);
    pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
    pin_conf.mux_position = 0x07;
    system_pinmux_pin_set_config(PINMUX_PA14H_GCLK_IO0 >> 16, &pin_conf);

    system_init();
    while(true);
}
```

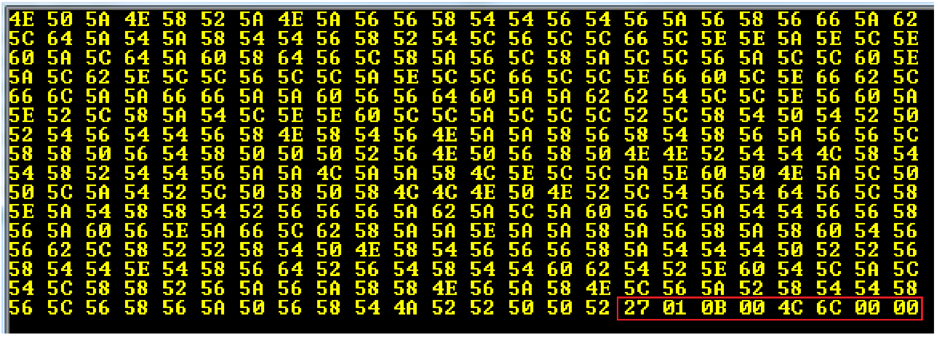
Note:

1. The I/O pin used here is PA14.
2. In **src/config/conf_clock.h**, **CONF_CLOCK_GCLK_0_OUTPUT_ENABLE** should be enabled true to enable GCLK out output to I/O pin.

8.2. CPU Idle Time Calculation from Result Observed

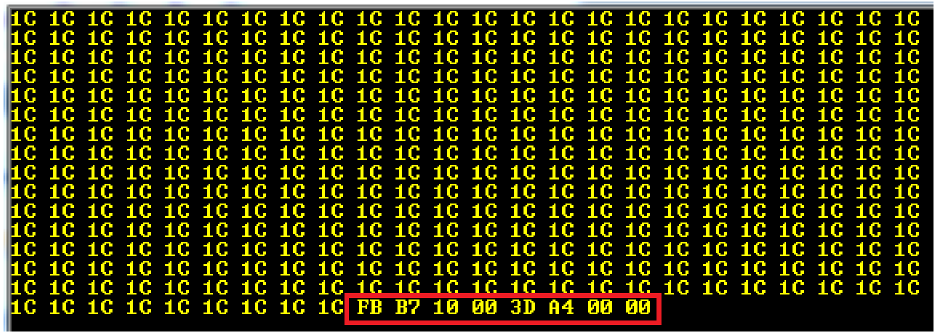
The following snap shots shows the results of various cases used in the application.

Figure 8-1. ADC_DMAC_USART Terminal Output



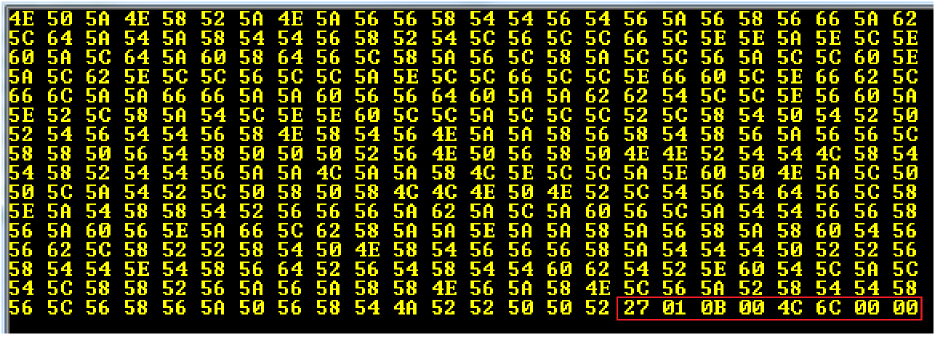
```
4E 50 5A 4E 58 52 5A 4E 5A 56 56 58 54 54 56 54 56 5A 56 58 56 66 5A 62
5C 64 5A 54 5A 58 54 54 56 58 52 54 5C 56 5C 5C 66 5C 5E 5E 5A 5E 5C 5E
60 5A 5C 64 5A 60 58 64 56 5C 58 5A 56 5C 58 5A 5C 5C 56 5A 5C 5C 60 5E
5A 5C 62 5E 5C 5C 56 5C 5C 5A 5E 5C 5C 66 5C 5C 5E 66 60 5C 5E 66 62 5C
66 6C 5A 5A 66 66 5A 5A 60 56 56 64 60 5A 5A 62 62 54 5C 5C 5E 56 60 5A
5E 52 5C 58 5A 54 5C 5E 5E 60 5C 5C 5A 5C 5C 5C 52 5C 58 54 50 54 52 50
52 54 56 54 54 56 58 4E 58 54 56 4E 5A 5A 58 56 58 54 58 56 5A 56 56 5C
58 58 50 56 54 58 50 50 50 52 56 4E 50 56 58 50 4E 4E 52 54 54 4C 58 54
54 58 52 54 54 56 5A 5A 4C 5A 5A 58 4C 5E 5C 5C 5A 5E 60 50 4E 5A 5C 50
50 5C 5A 54 52 5C 50 58 50 58 4C 4C 4E 50 4E 52 5C 54 56 54 64 56 5C 58
5E 5A 54 58 58 54 52 56 56 56 5A 62 5A 5C 5A 60 56 5C 5A 54 54 56 56 58
56 5A 60 56 5E 5A 66 5C 62 58 5A 5A 5E 5A 5A 58 5A 56 58 5A 58 60 54 56
56 62 5C 58 52 52 58 54 50 4E 58 54 56 56 56 58 5A 54 54 54 50 52 52 56
58 54 54 5E 54 58 56 64 52 56 54 58 54 54 60 62 54 52 5E 60 54 5C 5A 5C
54 5C 58 58 52 56 5A 56 5A 58 58 4E 56 5A 58 4E 5C 56 5A 52 58 54 54 58
56 5C 56 58 56 5A 50 56 58 54 4A 52 52 50 50 52 27 01 0B 00 4C 6C 00 00
```

Figure 8-2. ADC_DMAC_MEM_MEM_USART



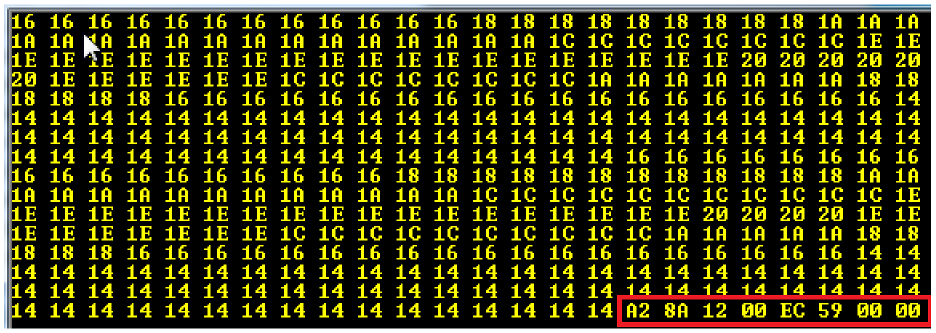
```
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
1C 1C 1C 1C 1C 1C 1C 1C 1C FB B7 10 00 3D A4 00 00
```

Figure 8-3. ADC_NO_DMAC_USART Terminal Output



```
4E 50 5A 4E 58 52 5A 4E 5A 56 56 58 54 54 56 54 56 5A 56 58 56 66 5A 62
5C 64 5A 54 5A 58 54 54 56 58 52 54 5C 56 5C 5C 66 5C 5E 5E 5A 5E 5C 5E
60 5A 5C 64 5A 60 58 64 56 5C 58 5A 56 5C 58 5A 5C 5C 56 5A 5C 5C 60 5E
5A 5C 62 5E 5C 5C 56 5C 5C 5A 5E 5C 5C 66 5C 5C 5E 66 60 5C 5E 66 62 5C
66 6C 5A 5A 66 66 5A 5A 60 56 56 64 60 5A 5A 62 62 54 5C 5C 5E 56 60 5A
5E 52 5C 58 5A 54 5C 5E 5E 60 5C 5C 5A 5C 5C 5C 52 5C 58 54 50 54 52 50
52 54 56 54 54 56 58 4E 58 54 56 4E 5A 5A 58 56 58 54 58 56 5A 56 56 5C
58 58 50 56 54 58 50 50 50 52 56 4E 50 56 58 50 4E 4E 52 54 54 4C 58 54
54 58 52 54 54 56 5A 5A 4C 5A 5A 58 4C 5E 5C 5C 5A 5E 60 50 4E 5A 5C 50
50 5C 5A 54 52 5C 50 58 50 58 4C 4C 4E 50 4E 52 5C 54 56 54 64 56 5C 58
5E 5A 54 58 58 54 52 56 56 56 5A 62 5A 5C 5A 60 56 5C 5A 54 54 56 56 58
56 5A 60 56 5E 5A 66 5C 62 58 5A 5A 5E 5A 5A 58 5A 56 58 5A 58 60 54 56
56 62 5C 58 52 52 58 54 50 4E 58 54 56 56 56 58 5A 54 54 54 50 52 52 56
58 54 54 5E 54 58 56 64 52 56 54 58 54 54 60 62 54 52 5E 60 54 5C 5A 5C
54 5C 58 58 52 56 5A 56 5A 58 58 4E 56 5A 58 4E 5C 56 5A 52 58 54 54 58
56 5C 56 58 56 5A 50 56 58 54 4A 52 52 50 50 52 27 01 0B 00 4C 6C 00 00
```

Figure 8-4. ADC_NO_DMAC_MEM_MEM_USART



```
16 16 16 16 16 16 16 16 16 16 16 16 18 18 18 18 18 18 18 18 18 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1C 1C 1C 1C 1C 1C 1C 1C 1E 1E
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 20 20 20 20 20
20 1E 1E 1E 1E 1E 1E 1C 1C 1C 1C 1C 1C 1C 1C 1A 1A 1A 1A 1A 1A 1A 18 18
18 18 18 18 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 16 16 16 16 16 16 16 16
16 16 16 16 16 16 16 16 16 16 18 18 18 18 18 18 18 18 18 18 18 18 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1E
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 20 20 20 20 1E 1E
1E 1E 1E 1E 1E 1E 1E 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1A 1A 1A 1A 1A 18 18
18 18 18 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 14 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 A2 8A 12 00 EC 59 00 00
```

For instance, take case ‘ADC_DMAC_USART’:

The last eight bytes of data represent the idle_loop_count and cycles_taken in big-endian format shown as follows. The last four bytes of result is the idle_loop_count and the next four bytes is the cycles_taken.

So idle_loop_taken = 0x00006C4C = 27724d

Cycles_taken = 0x000B0127 = 721191d

Time taken to complete transaction = (cycles_taken/CPU clock frequency)

Time taken to complete transaction = (721191/15.97)µs = 45.159ms

The time taken for each idle count is calculated to be 1.628µs as in CPU Utilization Calculation on page 25.

Total CPU idle time = idle_loop_count * 1.628µs = 27724 * 1.628µs = 45.134ms

Therefore, in 45.159ms transfer period, CPU is idle for about 45.134ms for ‘ADC_DMAC_USART’ case. In similar way, the calculation is done for other cases and the following table lists the same.

Case	Idle_loop_count	Cycles_taken	Total transfer time [ms]	CPU idle time [ms]	CPU idle time [%]
ADC_DMAC_USART	0x00006C4C	0x000B0127	45.159	45.134	99.94
ADC_DMAC_MEM_MEM_USART	0x0000899B	0x000E0391	68.608	68.449	99.76
ADC_NO_DMAC_USART	0x000059EC	0x00126687	75.510	37.476	49.63
ADC_NO_DMAC_MEM_MEM_USART	0x000059EC	0x00128AA2	76.088	37.476	49.25

It can be seen from the table that when using DMAC, CPU is idle most of the time during the data transfer. But without DMAC, the CPU is in idle mode only for some portion of data transfer and overall transfer time is also high.

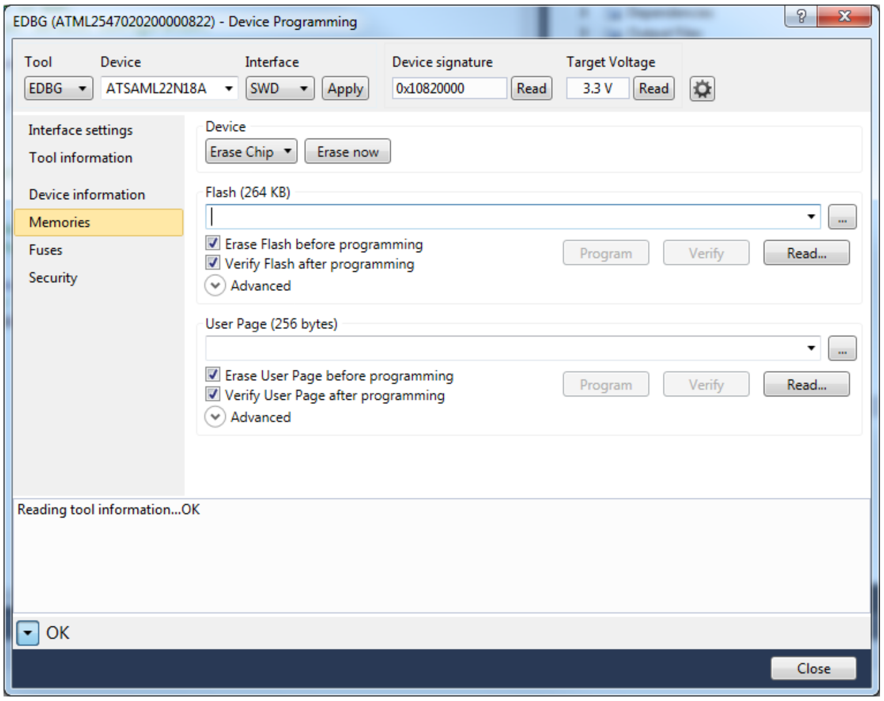
9. Execution of Application

The firmware corresponding to this application note comes with the Atmel Software Framework and it can be imported from Atmel Studio as well. The steps below explain the execution of this application.

Note: This chapter assumes that the setup is ready as per Setup on page 7 of this application note

1. Import the example in Atmel Studio from **File > New > Example Project > DMAC CPU Usage Demo – SAM L22 Xplained Pro**.
2. Choose the compiler option in **src\config\conf_dma.h** based on the execution mode needed.
3. Go to **Build > Build Solution** to compile the project.
4. Once it is compiled successfully, go to **Tools > Device Programming Window**.
5. Select appropriate tool, device and interface type and click **Apply** to connect to the kit. Check **Device Signature** and **Target Voltage** to ensure proper connection.
6. Now go to **Memories** Tab. Browse the ***.hex/elf** file location and click program to flash the device shown as follows:

Figure 9-1. Device Programming Window in Atmel Studio



EDBG (ATML2547020200000822) - Device Programming

Tool: EDBG
Device: ATSAML22N18A
Interface: SWD
Device signature: 0x10820000
Target Voltage: 3.3 V

Interface settings
Tool information
Device information
Memories
Fuses
Security

Device: Erase Chip | Erase now

Flash (264 KB)
☑ Erase Flash before programming
☑ Verify Flash after programming
Advanced

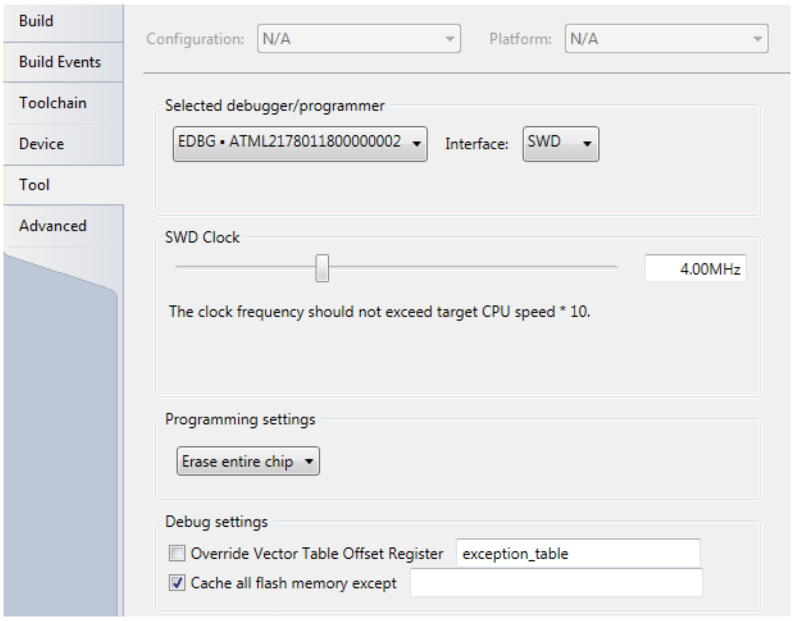
User Page (256 bytes)
☑ Erase User Page before programming
☑ Verify User Page after programming
Advanced

Reading tool information...OK

OK

7. To debug the code, right click on the project in the **Solution Explorer Window > Go to Project Properties**.
8. Go to **Tools > Debugger/programmer** as **EDBG** and **SWD** as **Interface**.

Figure 9-2. Debug Settings in Atmel Studio



Configuration: N/A Platform: N/A

Selected debugger/programmer

EDBG • ATML2178011800000002 Interface: SWD

SWD Clock

4.00MHz

The clock frequency should not exceed target CPU speed * 10.

Programming settings

Erase entire chip

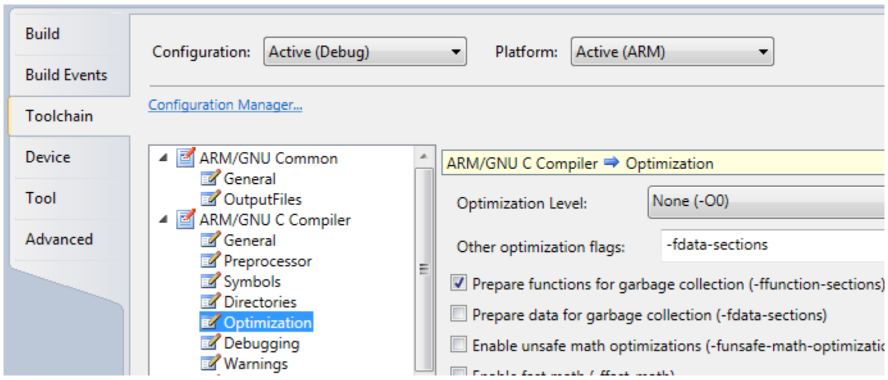
Debug settings

☐ Override Vector Table Offset Register exception_table

☑ Cache all flash memory except

9. Ensure optimization is **None** to utilize maximum debugging in **Toolchain > ARM®/GNU C Compiler > Optimization > Optimization Level > None(-O0)**.

Figure 9-3. Set Optimization Level



Configuration: Active (Debug) Platform: Active (ARM)

ARM/GNU C Compiler ➡ Optimization

Optimization Level: None (-O0)

Other optimization flags: -fdata-sections

☑ Prepare functions for garbage collection (-ffunction-sections)

☐ Prepare data for garbage collection (-fdata-sections)

☐ Enable unsafe math optimizations (-funsafe-math-optimizatic

10. Go to **Debug > Start Debugging and Break** to debug the code and click **Start without debugging** to continue programming without debugging.

10. References

10.1. ARM Documentation on Cortex-M0+ Core

- Cortex-M0+ Devices Generic User Guide revision r0p1
- Cortex-M0+ Technical Reference Manual revision r0p1

10.2. Atmel Software Framework (ASF)

Web page:

http://www.atmel.com/tools/avrsoftwareframework.aspx

Document/file:

- ASF update for Atmel Studio (.vsix) from ASF web page
- ASF update through Atmel Gallery *https://gallery.atmel.com/*
- ASF update through **Tools > Extension Manager** from Atmel Studio
- ASF standalone package for GCC makefile and IAR users
- Atmel AVR4029: Atmel Software Framework - User Guide
- Atmel AVR4030: Atmel Software Framework - Reference Manual

The ASF online documentation for the API and example usage are available at *http://asf.atmel.com*.

10.3. Atmel Studio

The latest version of Atmel Studio can be downloaded from http://www.atmel.com/tools/atmelstudio.aspx.

10.4. Device Datasheet

The device datasheet contains the block diagrams of the peripherals and details about implementing firmware for the device. It also contains the electrical specifications and expected characteristics of the device.

Datasheet is available on www.atmel.com in the Documents section of Atmel SAM L22 product page.

10.5. Hardware Tools User Guide

- For SAM L22 Xplained Pro User Guide and Schematics: http://www.atmel.com/devices/ATSAML22N18A.aspx?tab=tools
- For IO1 Xplained Pro User Guide and Schematics: http://www.atmel.com/tools/ATIO1-XPRO.aspx?tab=documents

10.6. Online Tools User Guide

Online help for each tool is available at the link http://www.atmel.com/webdoc/.

11. Revision History

Doc Rev.	Date	Comments
42627A	2/2016	Initial document release

Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**