SMART ARM-based Microcontrollers

AT07685: CPU Usage Demonstration using DMAC Application

APPLICATION NOTE

Description

Direct Memory Access Controller (DMAC) in Atmel® | SMART SAM D11 enables transfer between memories and peripherals and thus off-loads these tasks from the CPU. It enables high data transfer rates (using AHB clock) with minimum CPU intervention and frees up the CPU time.

This application note demonstrates the CPU usage when an application is designed with and without DMA. The analog data from light sensor is sampled using 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 covers the following peripheral features:

- DMA data transfer between
  - Peripheral to peripheral
  - Peripheral to memory
  - Memory to memory
  - Memory to peripheral
- 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	USB 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
SERCOM	Serial Communication Interface
SysTick	System Timer Tick
USART	Universal Synchronous and Asynchronous Receiver and Transmitter

2. Pre-requisites

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

- Atmel Studio 6.2 or later
- SAM D11 Xplained Pro
- ASF 3.27 or later

This application note covers an overview of the following peripheral. For better understanding of each peripheral, refer to the specific product datasheet.

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

3. Setup

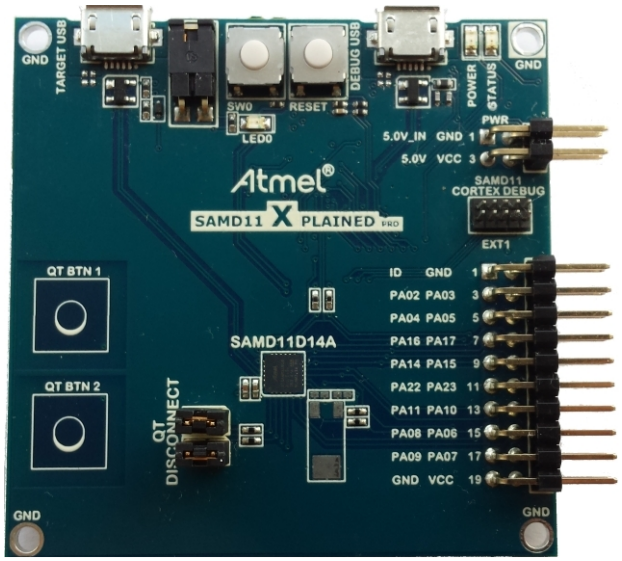
The application is developed for SAM D11 Xplained Pro board in Atmel Studio 6.2 or later. This chapter covers hardware and software setup required to test this application.

3.1. Hardware Setup

3.1.1. SAM D11 Xplained Pro

The Atmel SAM D11 Xplained Pro evaluation kit is a hardware platform for evaluating the ATSAMD11D14AM microcontroller. The SAM D11 Xplained Pro kit will be used to run the example application. This is an evaluation kit that allows connecting multiple external components using a wing connector. A wing is a self contained board that can be connected to the Xplained Pro using a wing connector. The SAM D11 Xplained Pro has one such wing connector marked as EXT1.

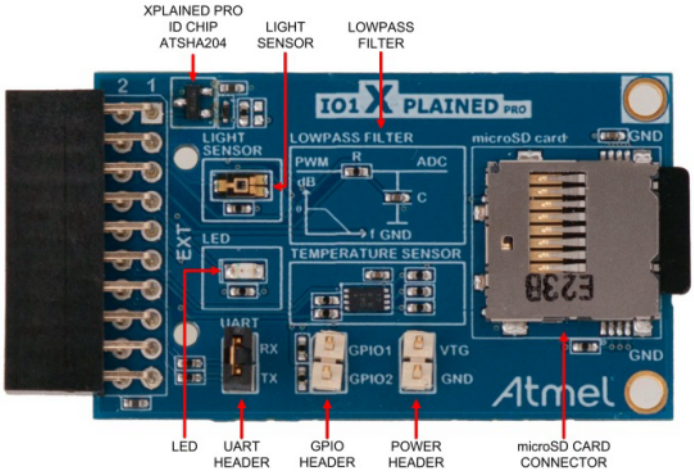
Figure 3-1 SAM D11 Xplained Pro



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 on the standard Xplained Pro extension header to further enhance the feature set of Xplained Pro MCU boards.

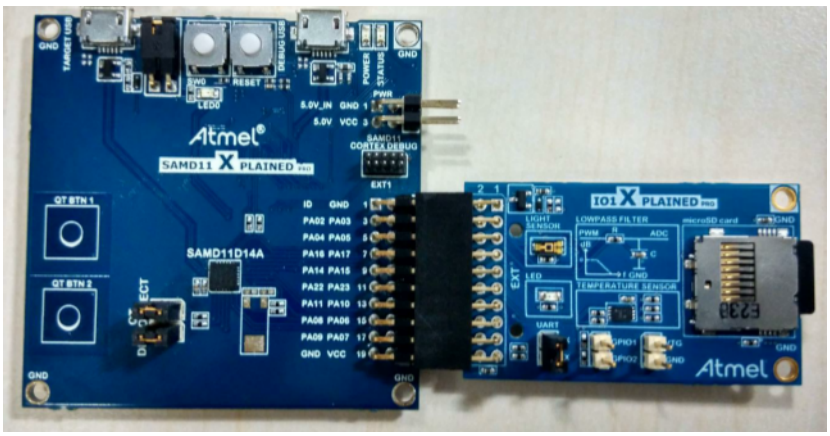
Figure 3-2 IO1 Xplained Pro Extension Board



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 SAM D11 Xplained Pro kit, only one header is available and its pin out can be referred from the board schematic file.

IO1 Xplained Pro features a 'TEMT6000' light sensor from Vishay Intertechnology. Pin3 of extension board is utilized for this purpose. The sensor data can be read by an ADC pin on Xplained Pro MCU board. In SAM D11 Xplained Pro kit, it is connected to EXT1 header as shown in the following figure. This application utilizes Light sensor on IO1 Xplained pro board as an analog input to the ADC.

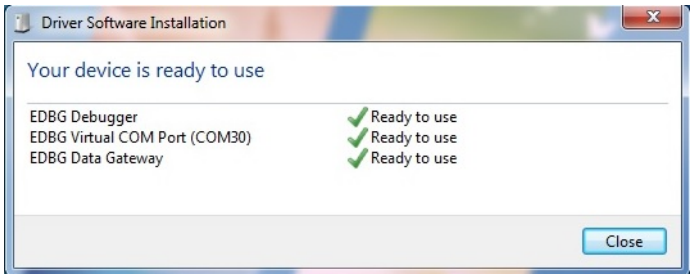
Figure 3-3 SAM D11 + IO1 Xplained Pro Board Connection



3.2. Software Setup

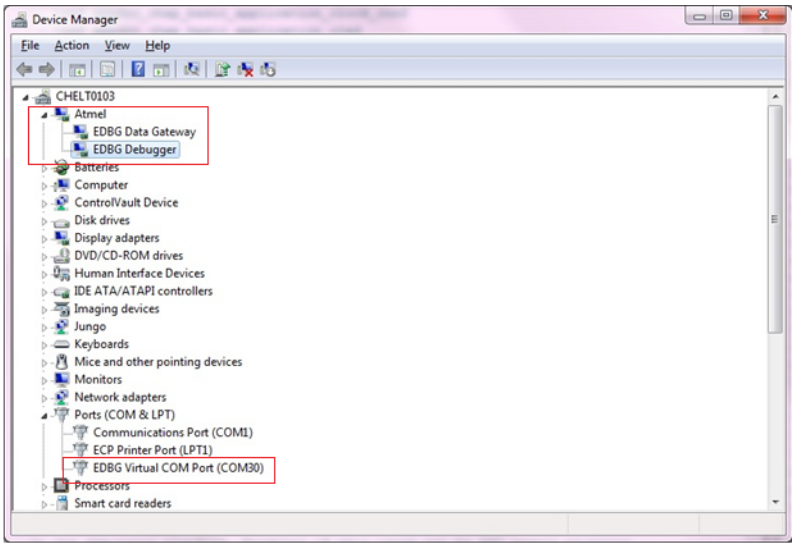
There are two USB ports on the SAM D11 Xplained Pro board - **DEBUG USB and TARGET USB**. For debugging Embedded debugger **EDBG, DEBUG USB** port has to be connected. Once the SAM D11 Xplained Pro kit is connected to the PC, the Windows® Task bar will pop-up a message as follows.

Figure 3-4 SAM D11 Xplained Pro Driver Installation



If the driver installation is successful, EDBG will be listed in the Device Manager as follows.

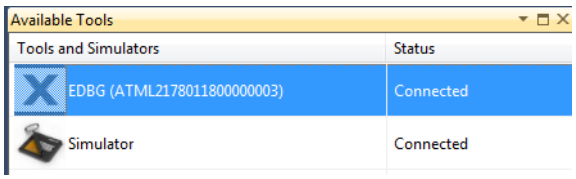
Figure 3-5 Successful EDBG Driver Installation



To ensure that the EDBG tool is getting 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** as shown in the following figure. It indicates that the tool is communicating properly with Atmel Studio.

Figure 3-6 EDBG under Available Atmel Tools



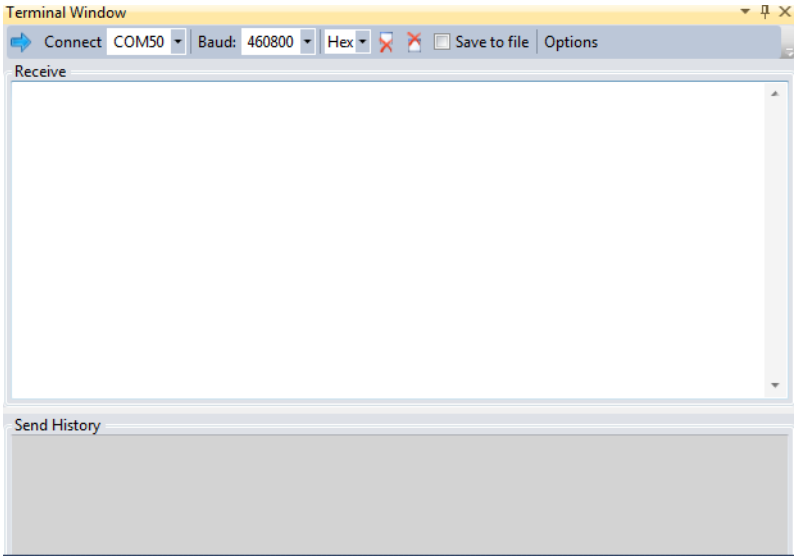
If the tool does not get displayed in **Available Atmel Tools**, disconnect the tool and reconnect again.

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

After software is successfully installed, open terminal window with the COM port (**EDBG Virtual COM port**) number detected in **Device Manager**. The terminal window can be downloaded and installed either from **Atmel Gallery** or through **Tools > Extension Manager**.

The terminal window can be opened from **View > Terminal Window**. The COM port should be opened with baudrate of **460800** with the display type as **hex**. This is because ADC result is directly written to **USART DATA** register in this application for demonstration purpose.

Figure 3-7 Terminal Window in Atmel Studio

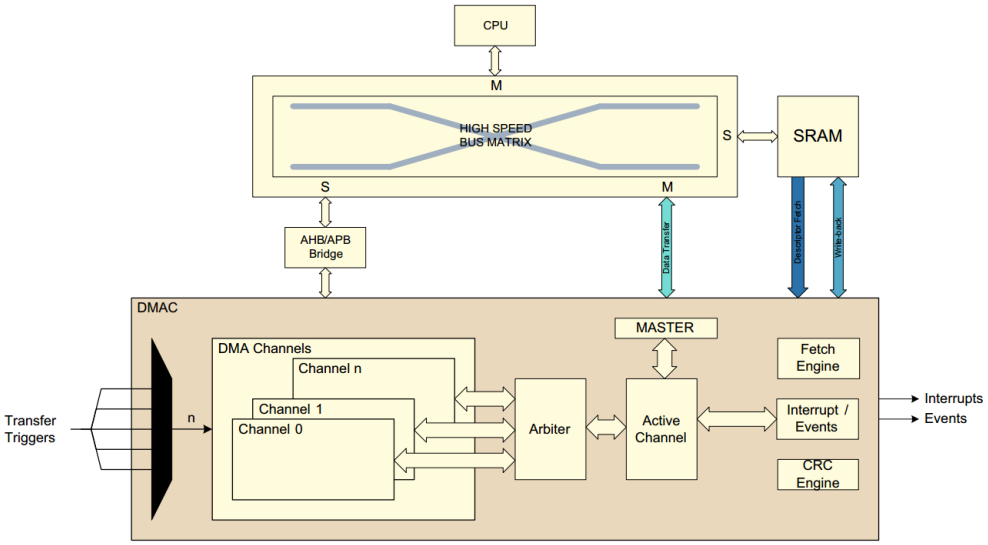


4. Direct Memory Access Controller

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) can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. It enables high data transfer rates (using AHB clock) with minimum CPU intervention and frees up CPU time. This will allow 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 six channels, enabling six 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 complete
- Selectable block transfer is complete
- DMA controller detects a bus error
- A channel operation is suspended

Event Input

One event input is available for each channel with event input support. The event can be programmed to trigger:

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

Event Output

One event output is available for each channel with event output support. 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 covers the overview of other peripherals relevant to this application note. 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 350ksps. 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. 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 (SERCOM2) 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. DMA Peripheral (ADC) – to – Peripheral (USART) Transfer

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 follows:

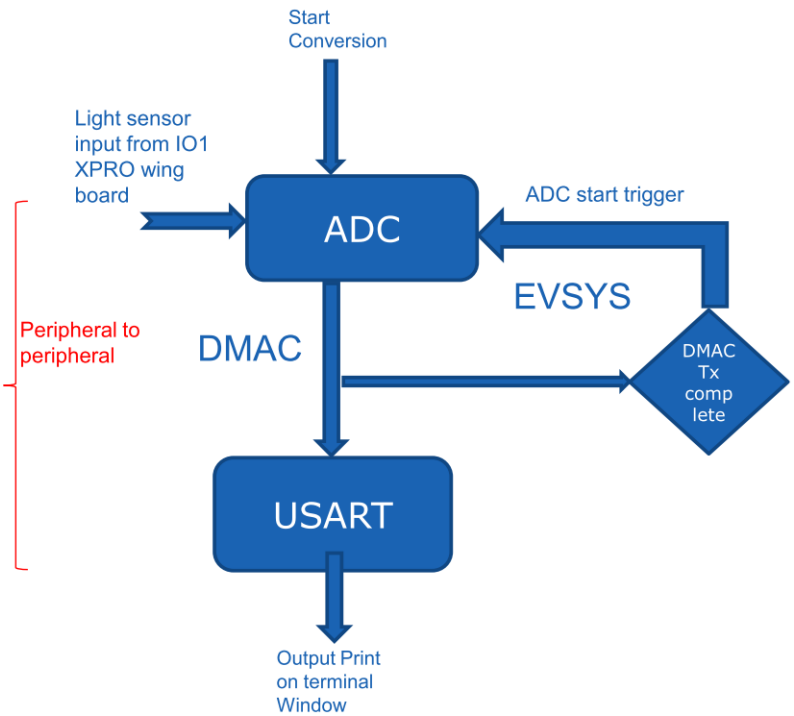
```
/* Set beat size as byte */
descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
/* Set block count as 1024 beats */
descriptor_config.block_transfer_count = BLOCK_COUNT;
/* Trigger interrupt once block transfer is complete */
descriptor_config.block_action = DMA_BLOCK_ACTION_INT;

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;
    #if defined(ADC_DMAC_USART)
        /* 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)(&SERCOM2->USART.DATA.reg);
    #endif
```

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 0 is used for this case and the configuration is done in `configure_dma_resource()` 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;
```

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. When 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 performed as shown in DMA Peripheral to Peripheral Transfer.

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 1024bytes (`BLOCK_COUNT`) and an interrupt is configured to flag when block transfer is complete. When block transfer is completed, DMAC channel gets disabled automatically.

6.1.2. CPU Utilization Calculation

As explained in CPU Utilization Calculation on page 23, time stamp from SysTick is taken before starting first ADC conversion in `main()`.

On completing 1024 byte transfer from ADC to USART, DMAC channel 0 `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 *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 Chapter 8 for detailed description about the CPU utilization calculation from the results observed.

6.2. DMAC: Peripheral (ADC) – Memory (SRAM) – Memory (SRAM) – Peripheral (USART) Transfer

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 Section 7.2, 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 0 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).

Channel 0 configuration (Peripheral to Memory):

As explained in DMAC: Peripheral (ADC) – Memory (SRAM) – Memory (SRAM) – Peripheral (USART) Transfer on page 17, 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 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;
```

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);
```

Once 1024 bytes samples get transferred from ADC to SRAM buffer (adc_result[]), dmac_channel0_callback() is called where the channel 1 transfer is triggered.

```
static void dmac_calback_channel0( struct dma_resource *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);

}
```

Channel 1 configuration (Memory to Memory):

Channel 1 is configured with software trigger and transfer type is transaction. I.e. when 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. When Channel 1 transfer is done, channel 2 is enabled to start the next conversion (as below) which is explained in next section.

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

Channel 2 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 channel 2.

```
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 = SERCOM2_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)(&SERCOM2->USART.DATA.reg);
    break;
```

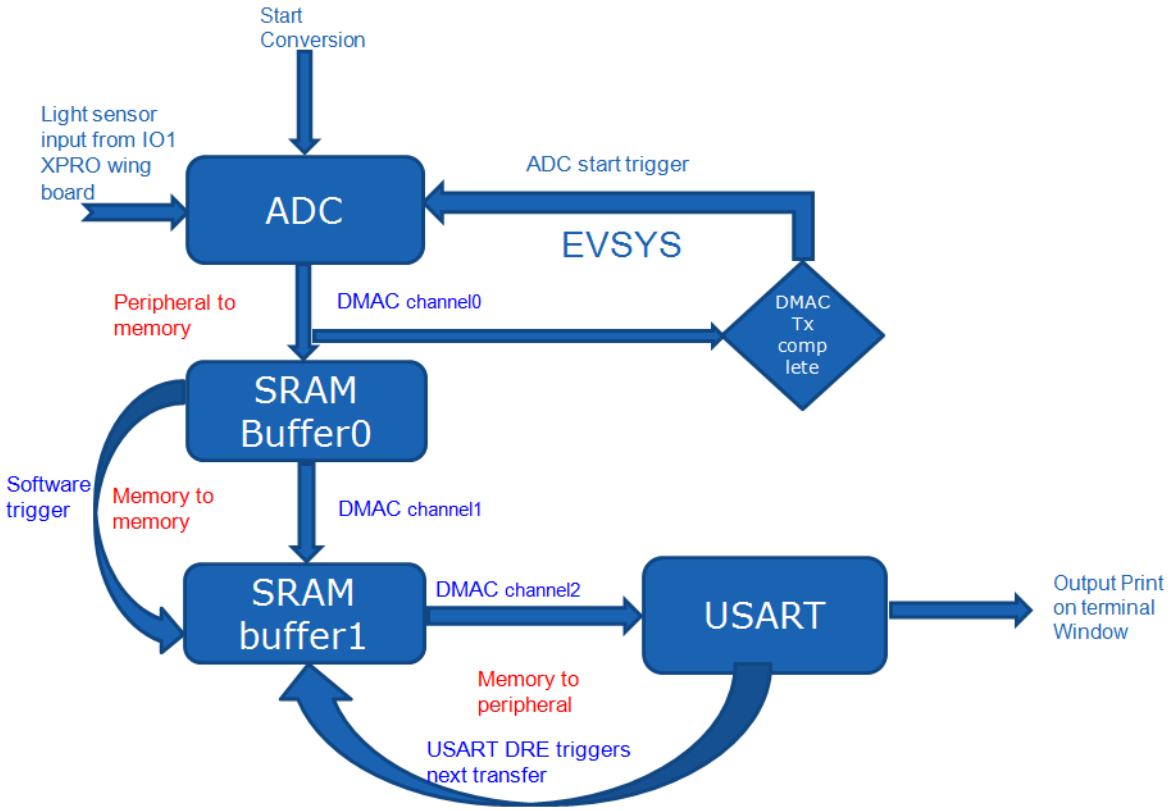
Unlike other channels, this channel should be enabled at the end of channel 1 transfer complete. The reason is the USART DRE is always set as there is not any previous communication occurs. So if this channel is enabled during the initialization, as USART DRE is already set, the DMA transfer will start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

When 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 *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 in the figure below.

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



6.2.2. CPU Utilization Calculation

The time stamp is taken at the DMAC channel 2 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 27 for detailed description about the CPU utilization calculation from the results observed.

6.3. ADC to SRAM to USART Transfer without DMAC

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:

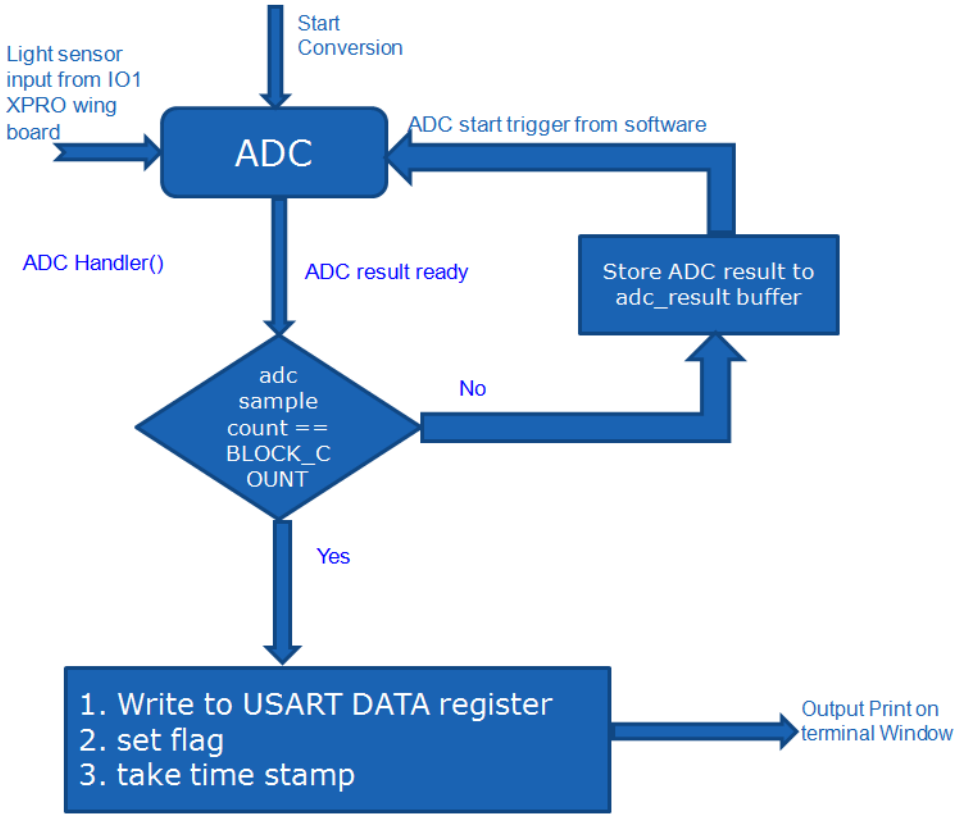
```
/* Enable ADC Result ready interrupt */
adc_interrupt_enable(&adc_instance,ADC_INTFLAG_RESRDY);
/* Enable ADC module interrupt in NVIC */
system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_ADC);
```

The first conversion is done in the main and the time stamp1 is taken. When 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. When 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.

```
/* 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);
```

Overall application flow would works as follows.

Figure 6-3 ADC to USART without DMAC



6.3.2. CPU Utilization Calculation

Note: Refer CPU Utilization Analysis Between Different Cases on page 27 for detailed description about the CPU utilization calculation from the results observed.

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 SERCOM2 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 23.

6.4. ADC – SRAM – SRAM – USART Transfer without DMAC

This option is enabled through `ADC_NO_DMAC_MEM_MEM_USART`. As mentioned in ADC to SRAM to USART Transfer without DMAC on page 20, 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`.

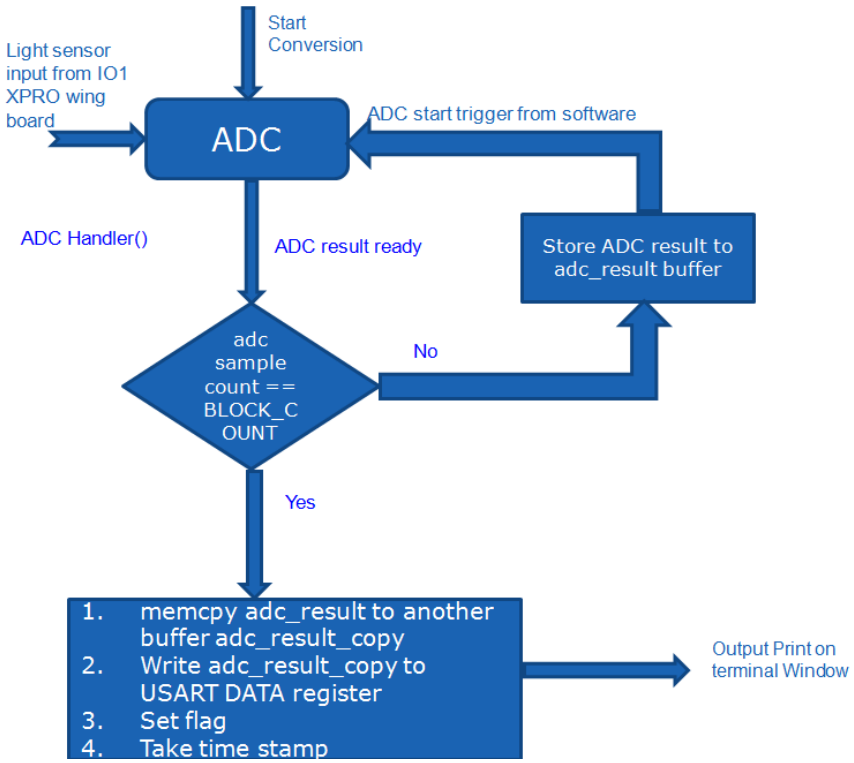
6.4.1. Application Implementation and Configuration

This scenario is same as ADC to SRAM to USART Transfer without DMAC on page 20 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 as follows.

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



6.4.2. CPU Utilization Calculation

The CPU utilization is similar as done in CPU Utilization Calculation on page 22.

Note: Refer CPU Utilization Analysis Between Different Cases on page 27 for detailed description about the CPU utilization calculation from the results observed.

6.5. CPU Utilization Calculation

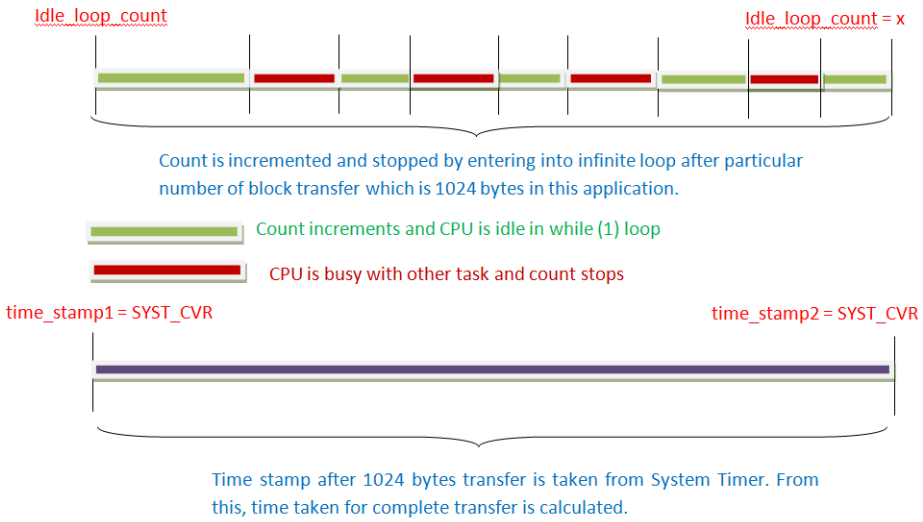
This section covers the logic implemented in this application to calculate the CPU utilization.

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



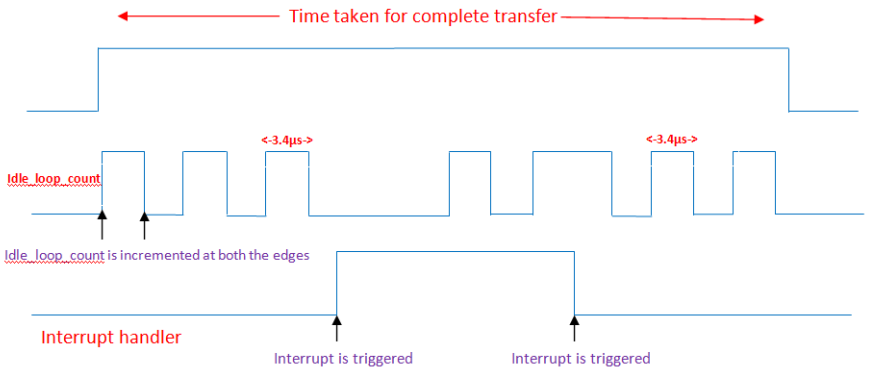
The number of cycles taken `cycles_taken` to complete the transaction can be calculated from the time stamp taken using SysTick. As SysTick runs at processor clock, the time taken for total transaction can be calculated from the cycles taken and the CPU clock frequency of the application as below.

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 in the following figure.

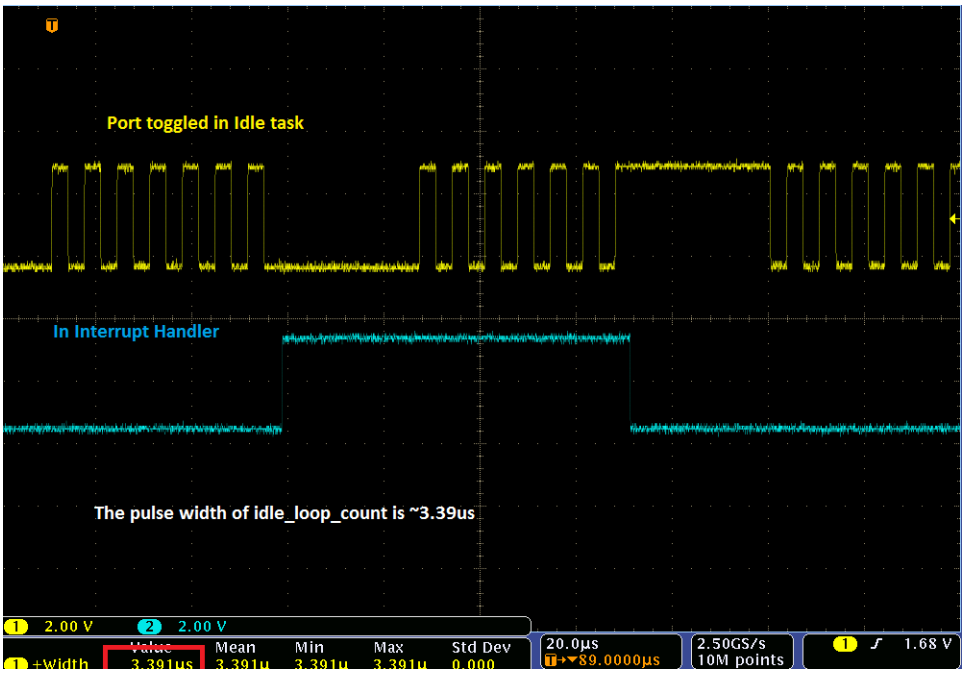
Figure 6-6 Calculation of Time Taken for a Single idle_loop_count Increment



From oscilloscope, the time taken for each count increment is calculated to be ~3.391µs, as shown in the following figure. The count value when multiplied with the pulse width (i.e. 3.391µs) will give the time CPU spends inside idle task.

Note: 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



The CPU utilization analysis for each case has been done in CPU Utilization Analysis Between Different Cases on page 27.

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. 8MHz) and it's prescalar which is 64 in this case. Default sampling time is CLK_ADC/2.

So ADC clock frequency = 8MHz/64 = 125kHz ~= 8µs

Sampling time = 16µs

Conversion time = 6 cycles = 6 * 8µs = 48µs

Total conversion time = sampling time + conversion time = ~(48 + 16)µs = **~64µs**

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} = 8MHz

So, maximum possible baudrate = 8MHz/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 64µs. USART would have sent the previous data in 21.7µs and waits for next ADC result without any data loss.
Note: The 10 bits comes from USART data frame namely **1** start bit + **8** data bit + **1** stop bit.

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. 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 Analysis Between Different Cases on page 27.

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`
- CPU runs at internal RC8M as source

8.1. CPU Frequency Calculation

To find the time taken, CPU frequency needs to be known. This application runs at internal 8MHz RC (OSC8M) which accuracy can vary from 7.94MHz to 8.06MHz (refer to 'Electrical Characteristics' section of the product datasheet). The accuracy of RC is calculated to be 7.94MHz on the board tested. This is done by giving the main clock i.e. GCLK0 (which runs at ODC8M) 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_PA08H_GCLK_IO0 >> 16, &pin_conf);


    system_init();

    //! [main]
    while (true);
}
```

Note:

1. The I/O pin used here is PA08.
2. In `src/config/conf_clock.h`, option `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_NO_DMAC_USART Terminal Output

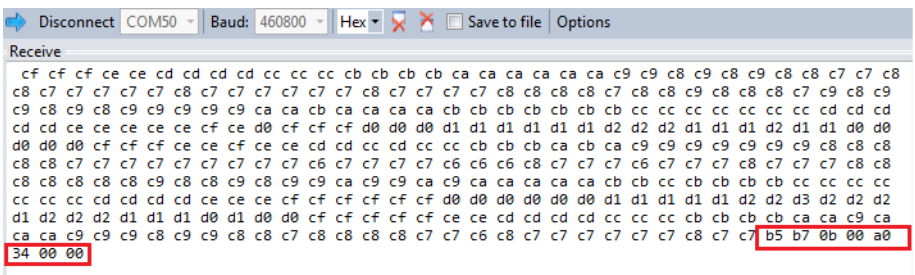


Figure 8-2 ADC_DMAC_MEM_MEM_USART

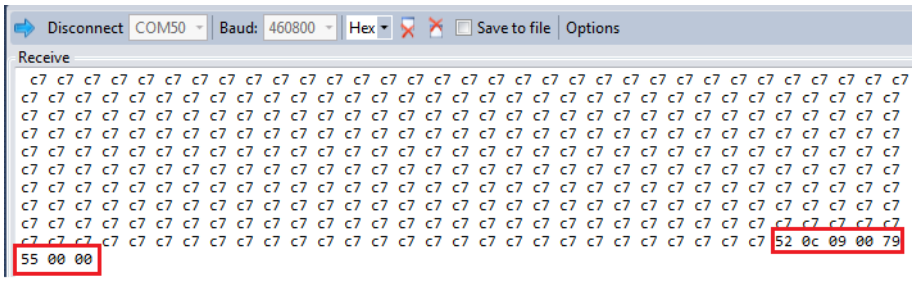
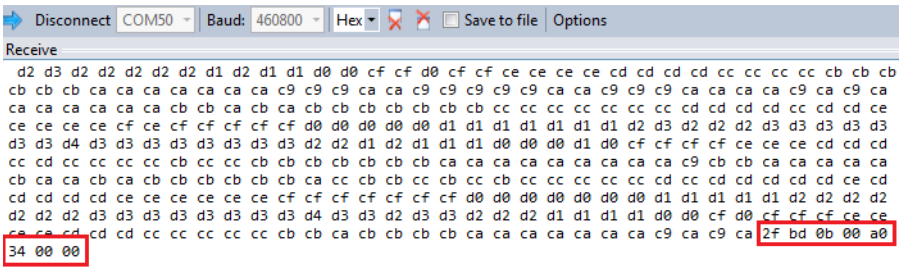


Figure 8-3 ADC_NO_DMAC_MEM_MEM_USART



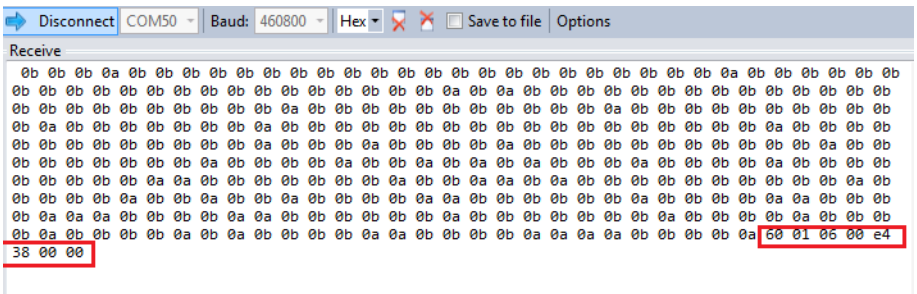
For instance, take case 'ADC_DMAC_USART':

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

So `idle_loop_taken` = 0x000038e4 = 14564d

And `cycles_taken` = 0x00060160 = 393568d

Figure 8-4 ADC_DMAC_USART Terminal Output



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

Time taken to complete transaction = (393568/7.94)µs = 49567µs = 49.567ms

The time taken for each idle count is calculated to be 3.391µs as in Section 6.1.

Total CPU idle time = `idle_loop_count` * 3.391µs = 14564 * 3.391µs = 49.386ms

Therefore, in 49.567ms transfer period, CPU is idle for about 49.386ms for `ADC_DMAC_USART` case. In similar way, the calculation is done for other cases and listed in the following table.

Table 8-1 The idle_loop_count and cycles_taken for Different Applications

Case	idle_loop_count	cycles_taken	Total Transfer Time [ms]	CPU Idle Time [ms]	CPU Idle Time [%]
ADC_DMAC_USART	0x000038e4	0x000060160	49.567	49.386	96.63
ADC_DMAC_MEM_MEM_USART	0x00005579	0x000905c2	74.682	74.198	99.35
ADC_NO_DMAC_USART	0x000034a0	0x000bb7b5	96.715	45.683	47.23
ADC_NO_DMAC_MEM_MEM_USART	0x000034a0	0x000bbd2f	96.892	45.683	47.14

From the above table, it can be seen 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.

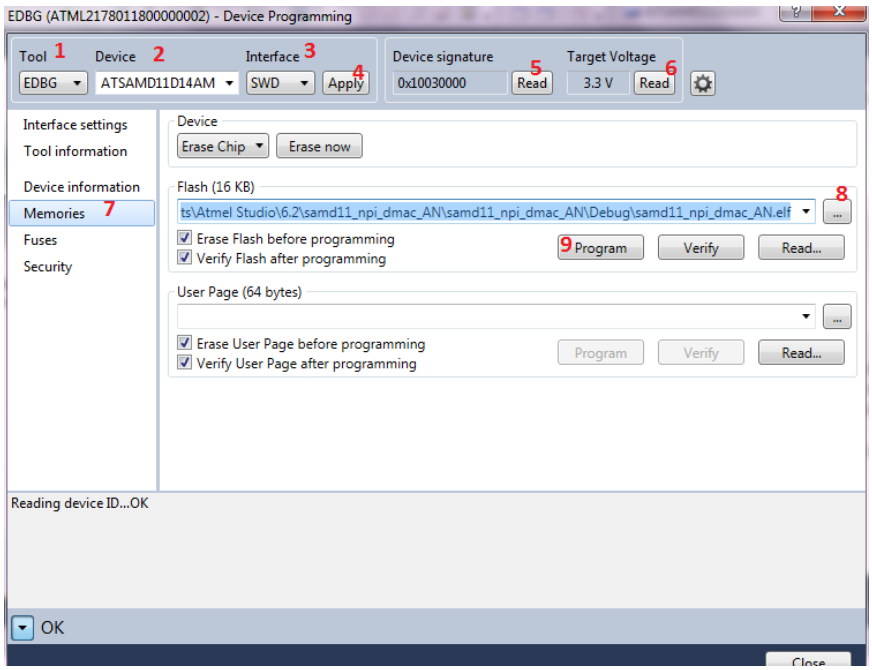
9. Execution of Application

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.

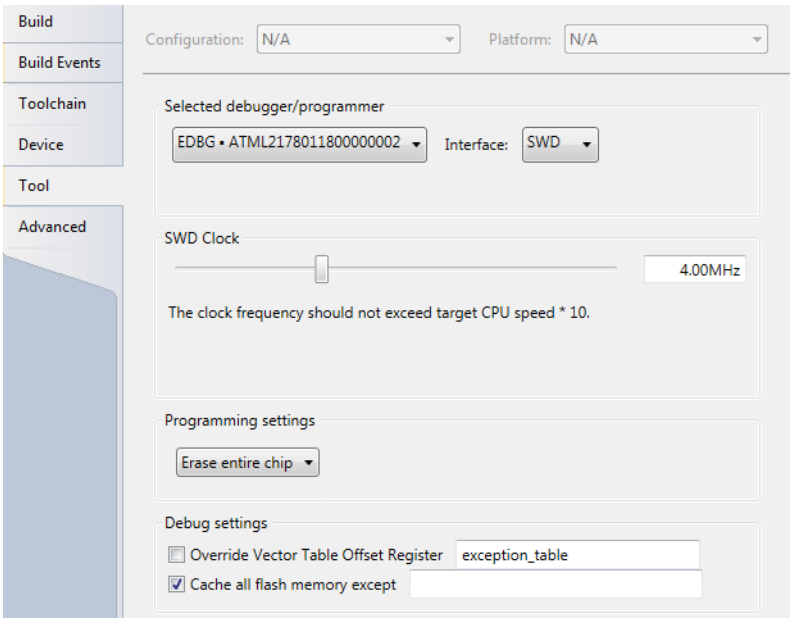
- Import the example in Atmel Studio from **File > New > Example Project > DMAC CPU Usage Demo – SAM D11 Xplained Pro**
- Choose the compiler option in `src\config\conf_dma.h` based on the execution mode needed
- Go to **Build > Build Solution** to compile the project
- Once it is compiled successfully, go to **Tools > Device Programming Window**
- 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.
- Go to **Memories** Tab. Browse the `*.hex/elf` file location and click program to flash the device as shown in the following figure

Figure 9-1 Device Programming Window in Atmel Studio



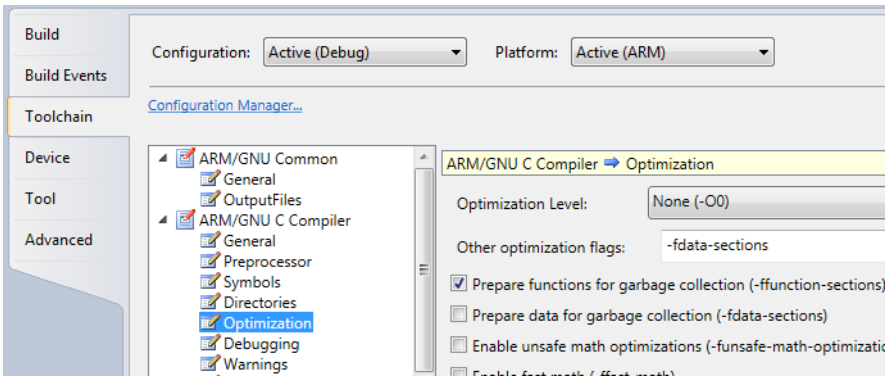
- To debug the code, right click on the project in the **Solution Explorer Window > Go to Project Properties**
- Go to **Tools > Debugger/programmer** as **EDBG** and **SWD** as **Interface**

Figure 9-2 Debug Settings in Atmel Studio



- 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



- 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. 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 D09/D10/D11 product page.

10.2. ARM Documentation on Cortex-M0+ Core

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

10.3. Atmel Studio

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

10.4. Hardware Tools User Guide

- For SAM D11 Xplained Pro User Guide and Schematics: http://www.atmel.com/devices/ATSAMD11D14A.aspx?tab=tools
- For IO1 Xplained Pro User Guide and Schematics: http://www.atmel.com/tools/ATIO1-XPRO.aspx?tab=documents

10.5. Online Tools User Guide

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

10.6. 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*.

11. Revision History

Doc Rev.	Date	Comments
42371B	12/2015	Updated for SAM D09 device
42371A	09/2014	Initial document release

Atmel | 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**