APPLICATION NOTE

AT13214: Using Cyclic Redundancy Check Calculation Unit (CRCCU) on SAM4S

SMART ARM-based Microcontrollers

Introduction

The Cyclic Redundancy Check Calculation Unit (CRCCU) is designed to perform data integrity checks of off-/on-chip memories as a background task without CPU intervention.

The CRCCU has its own DMA which functions as a Master with the Bus Matrix. Three different polynomials are available: CCITT802.3, CASTAGNOLI, and CCITT16.

In this application note, it provides three examples to demonstrate the usage of CRCCU on SAM4S and the benefits of the Hardware CRCCU module compared with the optimized software CRC algorithm.

Chapter 5 demonstrates the CRCCU Polling Mode of Flash Integrity Check.

Chapter 6 demonstrates the CRCCU Callback Mode of Flash Integrity Check.

Chapter 7 demonstrates the benefits of the Hardware CRCCU implementation compared with the optimized software CRC algorithm.

Features

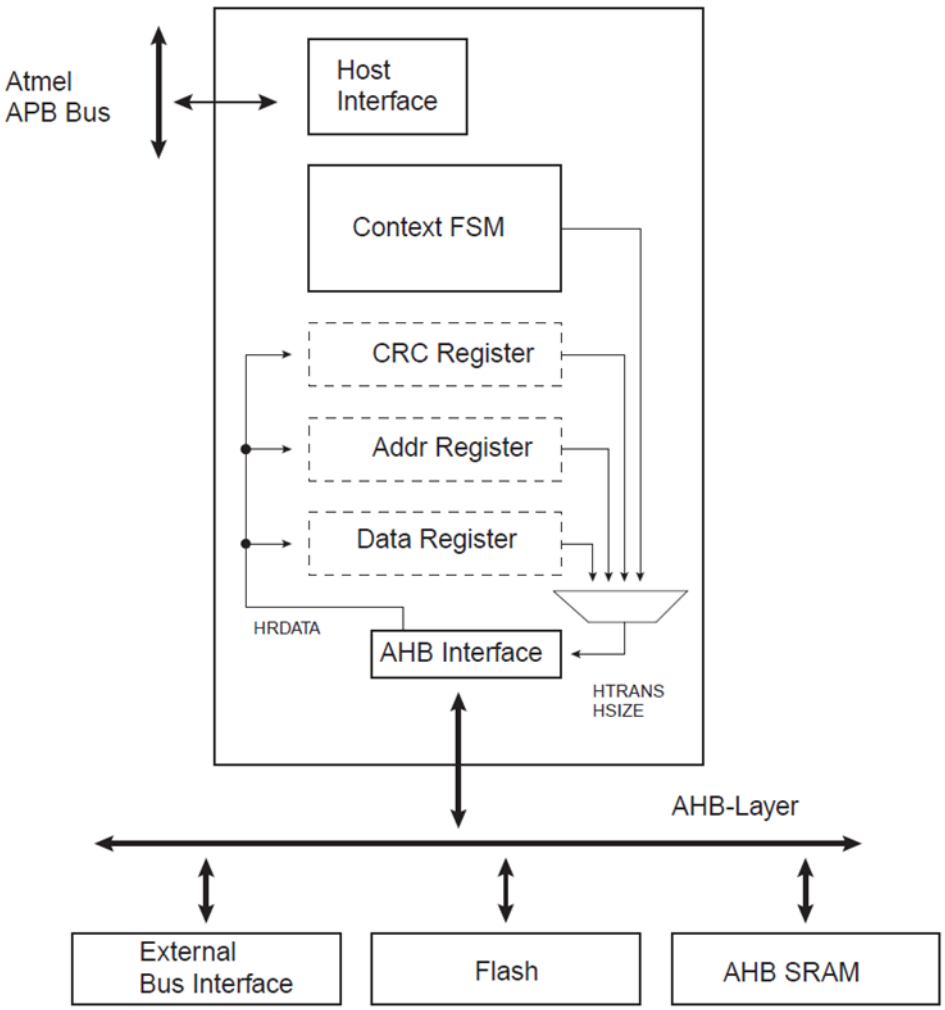
- Data Integrity Check of Off-/On-Chip Memories
- Background Task without CPU Intervention
- Performs Cyclic Redundancy Check (CRC) Operation on Programmable Memory Area
- Programmable Bus Burden

Note: The CRCCU is designed to verify data integrity of off-/on-chip memories, thus the CRC must be generated and verified by the CRCCU. The CRCCU performs the CRC from LSB to MSB. If the CRC has been performed with the same polynomial by another device, a bit-reverse must be done on each byte before using the CRCCU.

Table of Contents

1 CRCCU Block Diagram

Figure 1-1. Block Diagram



2 Product Dependencies

2.1 Power Management

The CRCCU is clocked through the Power Management Controller (PMC), hence the CRCCU clock must be enabled through the PMC configuration before it can be used.

2.2 Interrupt Source

The CRCCU has an interrupt line connected to the Interrupt Controller. In order to handle the CRCCU interrupt, the Interrupt Controller must be configured accordingly.

3 CRCCU Functional Description

3.1 CRC Calculation Unit

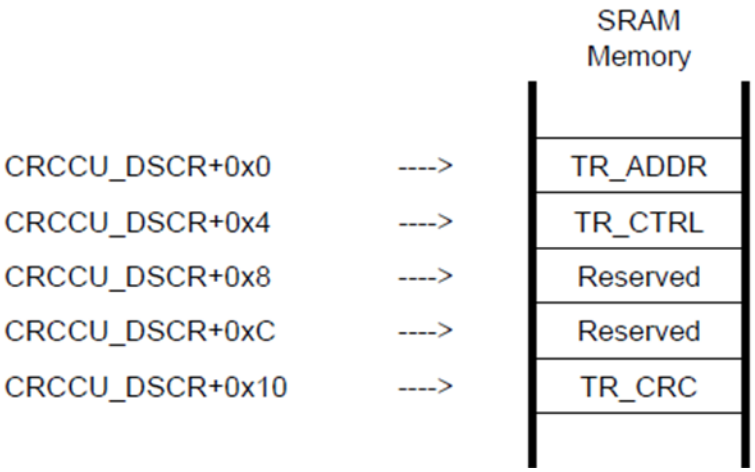
The CRCCU implements a dedicated Cyclic Redundancy Check (CRC) engine. After being configured and enabled, this CRC engine performs a checksum computation on a memory area. CRC computation is performed from the LSB to MSB. Three different polynomials are available: CCITT802.3, CASTAGNOLI and CCITT16.

3.2 CRC Calculation Unit Operation

The CRCCU has a DMA controller that supports configurable CRC memory checks. When enabled, the DMA channel reads a configured amount of data and computes CRC on the fly.

The CRCCU is controlled by two registers, TR_ADDR and TR_CTRL, which need to be mapped in the internal SRAM. The addresses of these two registers are pointed at by the CRCCU_DSCR.

Figure 3-1. CRCCU Descriptor Memory Mapping



Note: The DMA must be enabled to start CRCCU execution.

TR_ADDR defines the start address of memory area targeted for CRC calculation.

TR_CTRL defines the buffer transfer size, the transfer width (byte, half word, and word) and the transfer-completed Interrupt enable.

To start the CRCCU, the user needs to set the CRC enable bit (ENABLE) in the CRCCU Mode Register (CRCCU_MR), then configure it and finally set the DMA enable bit (DMAEN) in the CRCCU DMA Enable Register (CRCCU_DMA_EN).

After the DAM enable bit is set, the CRCCU reads the predefined amount of data (defined in TR_CTRL) located from TR_ADDR start address and computes the checksum.

The CRCCU_SR contains the temporary CRC value.

The BTSIZE field located in the TR_CTRL register (located in SRAM memory), is automatically decremented if its value is different from zero. Once the value of the BTSIZE field is equal to zero, the CRCCU is disabled by hardware. In this case, the CRCCU DMA Status Register bit DMASR is automatically cleared.

If the COMPARE field of the CRCCU_MR is set to true, the TR_CRC (Transfer Reference Register) is compared with the last CRC computed. If a mismatch occurs, an error flag is set and an interrupt is raised (if unmasked).

The CRCCU accesses the memory by single access (TRWIDTH size) in order not to limit the bandwidth usage of the system, and the DIVIDER field of the CRCCU Mode Register can be used to lower it by dividing the frequency of the single accesses.

The CRCCU traverses the defined memory area using ascending addresses.

In order to compute the CRC for a memory size larger than 256KB or for non-contiguous memory area, it is possible to re-enable the CRCCU on the new memory area and the CRC will be updated accordingly. Use the RESET field of the CRCCU_CR to reset the CRCCU Status Register to its default value (0xFFFF_FFFF).

4 Registers Memory Mapping

Table 4-1. Transfer Control Register Memory Mapping

Offset	Register	Name	Access
CRCCU_DSCR + 0x0	CRCCU Transfer Address Register	TR_ADDR	Read/Write
CRCCU_DSCR + 0x4	CRCCU Transfer Control Register	TR_CTRL	Read/Write
CRCCU_DSCR + 0xC – 0x10	Reserved	-	-
CRCCU_DSCR + 0x10	CRCCU Transfer Reference Register	TR_CRC	Read/Write

Note: These registers are SRAM mapped.

Table 4-2. Cyclic Redundancy Check Calculation Unit (CRCCU) User Interface

Offset	Register	Name	Access	Reset
0x000	CRCCU Descriptor Base Register	CRCCU_DSCR	Read/Write	0x00000000
0x004	Reserved	-	-	-
0x008	CRCCU DMA Enable Register	CRCCU_DMA_EN	Write-only	0x00000000
0x00C	CRCCU DMA Disble Register	CRCCU_DMA_DIS	Write-only	0x00000000
0x010	CRCCU DMA Status Register	CRCCU_DMA_SR	Read-only	0x00000000
0x014	CRCCU DMA Interrupt Enable Register	CRCCU_DMA_IER	Write-only	0x00000000
0x018	CRCCU DMA Interrupt Disable Register	CRCCU_DMA_IDR	Write-only	0x00000000
0x001C	CRCCU DMA Interrupt Mask Register	CRCCU_DMA_IMR	Read-only	0x00000000
0x020	CRCCU DMA Interrupt Status Register	CRCCU_DMA_ISR	Read-only	0x00000000
0x024-0x030	Reserved	-	-	-
0x034	CRCCU Control Register	CRCCU_CR	Write-only	0x00000000
0x038	CRCCU Mode Register	CRCCU_MR	Read/Write	0x00000000
0x03C	CRCCU Status Register	CRCCU_SR	Read-only	0xFFFFFFFF
0x040	CRCCU Interrupt Enable Register	CRCCU_IER	Write-only	0x00000000
0x044	CRCCU Interrupt Disable Register	CRCCU_IDR	Write-only	0x00000000
0x048	CRCCU Interrupt Mask Register	CRCCU_IMR	Read-only	0x00000000
0x004C	CRCCU Interrupt Status Register	CRCCU_ISR	Read-only	0x00000000
0x050-0x0FC	Reserved	-	-	-

5 CRCCU Polling Mode

This chapter describes how to implement the CRCCU polling mode on SAM4S step by step.

All the examples provided with this application note have been executed under the following conditions:

- Hardware: Atmel® SAM4S-EK2 board with ATSAM4SD32C
- System Clock: 8MHz, internal RC used with FW=0
- CRC Configuration: Polynomial CCITT802.3 with CRC_INIT:0xFFFFFFFF, CRC_POLY:0x04C11DB7
- Total Checked Flash Size: 32KB
- DMA transfer Width: WORD (four bytes)

Note: There are three values which can be selected for DMA transfer; BYTE, HALFWORD, and WORD. For CRCCU, WORD selection the best performance can be achieved as it transfers four bytes per DMA transition.

5.1 Define Flash Start Address and Flash Size to be Checked

Below macros define the checked start address, total checked Flash size, and DMA transfer size for each beat.

```
/* In this sample code, by default, it uses the 4BYTE align to do the CRC checking*/
#define TEST_CRCCU_BUF_ADDRESS (0x00400000)  /* Flash Start address*/
#define TEST_SIZE              (0x100000/32)  /* Flash Size 32KB */
#define TEST_CRCCU_SIZE        (TEST_SIZE/4)  /*WORD transfer by default*/
```

In the demo example, the checked start address is 0x00400000 and the total checked flash size is 32KB. The users can adjust the checked Flash start address and Flash size flexibly if they want to check on a different start address and size.

5.2 Define the CRCCU Descriptor

Define the CRCCU descriptor in main.c and it needs 512 byte aligned.

```
/** CRC descriptor */
COMPILER_ALIGNED (512)
crccu_dscr_type_t crc_dscr;
```

The user can find the CRCCU structure definition in the crccu.h file:

```
/** CRCCU descriptor type */
typedef struct crccu_dscr_type {
        uint32_t ul_tr_addr;   /* TR_ADDR */
        uint32_t ul_tr_ctrl;   /* TR_CTRL */
#if (SAM3SD8 || SAM4S || SAM4L || SAMG55)
        uint32_t ul_reserved[2];       /* Reserved register */
#elif SAM3S
        uint32_t ul_reserved[52];      /* TR_CRC begins at offset 0xE0 */
#endif
        uint32_t ul_tr_crc;    /* TR_CRC */
} crccu_dscr_type_t;
```

5.3 Enable CRCCU Clock

Enable the CRCCU peripheral clock before using this peripheral. The CRCCU peripheral ID on SAM4S is 32.

```
pmc_enable_periph_clk(CRCCU);
```

Refer to the SAM4S datasheet for the corresponding Peripheral Identifiers.

Table 5-1. CRCCU Peripheral ID

Instance ID	Instance name	NVIC interrupt	PMC clock control	Instance description
31	PWM	X	X	Pulse width modulation
32	CRCCU	X	X	CRC calculation unit
33	ACC	X	X	Analog comparator
34	UDP	X	X	USB device port

5.4 System Clock Configuration

The system clock is running at 8MHz with internal RC. The user can refer to conf_clock.h for clock configuration.

Figure 5-1. 8MHz System Clock Configuration

```
#ifndef CONF_CLOCK_H_INCLUDED
#define CONF_CLOCK_H_INCLUDED

// ===== System Clock (MCK) Source Options
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_SLCK_RC
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_SLCK_XTAL
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_SLCK_BYPASS
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_MAINCK_4M_RC
#define CONFIG_SYSCLK_SOURCE          SYSCLK_SRC_MAINCK_8M_RC
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_MAINCK_12M_RC
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_MAINCK_XTAL
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_MAINCK_BYPASS
//#define CONFIG_SYSCLK_SOURCE          SYSCLK_SRC_PLLACK
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_PLLBCK

// ===== System Clock (MCK) Prescaler Options   (Fmck = Fsys / (SYSCLK_PRES))
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_1
#define CONFIG_SYSCLK_PRES            SYSCLK_PRES_2
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_4
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_8
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_16
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_32
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_64
//#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_3

// ===== PLL0 (A) Options   (Fpll = (Fclk * PLL_mul) / PLL_div)
// Use mul and div effective values here.
#define CONFIG_PLL0_SOURCE            PLL_SRC_MAINCK_XTAL
#define CONFIG_PLL0_MUL               20
#define CONFIG_PLL0_DIV               1
```

5.5 Configure and Run CRCCU

There are three parameters of CRCCU calculation API compute_crc(): Start address, Length, and Polynomial.

For this API, it doesn't need to know whether the start address is located on the Flash or SRAM. The user just needs to pass the Flash or SRAM buffer address, data length, and Polynomial to this API for CRC calculation.

The detail procedures to configure and run CRCCU are described below:

- Before any CRC checking, the user should reset CRCCU to make sure the CRC value is initialized to 0xFFFFFFFF

  ```
  crccu_reset(CRCCU);
  ```

- Check the data length (ul_length): If it is larger than MAX_BTSIZE (0xFFFF), then the data buffer should be divided into several blocks and call CRCCU function for multiple times.

Note: In this document, the MAX_BTSIZE is the maximum BTSIZE in register TR_CTRL.

```
if( ul_length > (MAX_BTSIZE))
{
  real_calculate_length = (MAX_BTSIZE);
}
else
{
  real_calculate_length = ul_length ;
}
```

- Initialize and configure the CRC descriptor as defined in Section 5.2, the CRCCU descriptor is located in SRAM with 512 byte aligned

```
memset((void *)&crc_dscr, 0, sizeof(crccu_dscr_type_t));

crc_dscr.ul_tr_addr = (uint32_t) p_buffer;

/* Transfer width: word, interrupt enable */
crc_dscr.ul_tr_ctrl =
          CRCCU_TR_CTRL_TRWIDTH_WORD | real_calculate_length |
          CRCCU_TR_CTRL_IEN_ENABLE;

/* Configure the CRCCU descriptor */
crccu_configure_descriptor(CRCCU, (uint32_t) &crc_dscr);
```

- Configure CRCCU mode: Three modes can be selected. In this demo example, Polynomial 0x04C11DB7 is selected.

```
/* Configure CRCCU mode */
crccu_configure_mode(CRCCU, CRCCU_MR_ENABLE | ul_polynomial_type);
```

Note: The polynomial type is defined in header file crccu.h as below:

```
#define   CRCCU_MR_PTYPE_CCITT8023 (0x0u << 2) /**< \brief (CRCCU_MR) Polynom 0x04C11DB7 */
#define   CRCCU_MR_PTYPE_CASTAGNOLI (0x1u << 2) /**< \brief (CRCCU_MR) Polynom 0x1EDC6F41 */
#define   CRCCU_MR_PTYPE_CCITT16 (0x2u << 2) /**< \brief (CRCCU_MR) Polynom 0x1021 */
```

- Enable the CRCCU DMA to start the CRC calculation and CPU just needs to wait for the calculation to finish by checking the DMA Status Register bit.

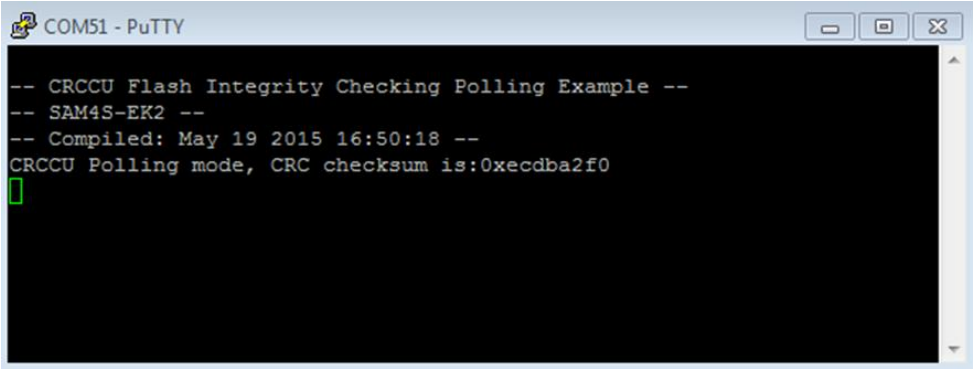
```
/* Start the CRC calculation */
crccu_enable_dma(CRCCU);

/* Wait for calculation ready */
while ( crccu_get_dma_status(CRCCU) == CRCCU_DMA_SR_DMASR) {
}
```

- Get the CRC value

```
ul_crc = crccu_read_crc_value(CRCCU);
```

5.6 Output of CRCCU Polling Mode

Figure 5-2. Output of CRCCU Polling Mode



6 CRCCU Callback Mode

The CRCCU callback mode configuration is the same as the polling mode except that it will use the CRCCU interrupt to get the final CRC value. The CPU can be released during CRCCU execution.

6.1 CRCCU Callback Interrupt Definition

The CRCCU interrupt will be triggered once the CRCCU has finished the execution. During the execution of CRCCU, the CPU is fully released for other tasks which will potentially increase the total CPU performance.

The CRCCU callback interrupt is defined in main.c as below:

```
void CRCCU_Handler(void)
{
        if((crccu_get_dma_status(CRCCU) == 0))
        {
                ul_crc=crccu_read_crc_value(CRCCU);
                printf("CRCCU Callback mode, CRC checksum is:0x%08x\n\r", ul_crc);
                crccu_get_dma_interrupt_status(CRCCU);
        }
}
```

6.2 CRCCU Callback Interrupt Setup

```
NVIC_DisableIRQ(CRCCU_IRQn);
NVIC_ClearPendingIRQ(CRCCU_IRQn);
NVIC_SetPriority(CRCCU_IRQn, 0);
NVIC_EnableIRQ(CRCCU_IRQn);
crccu_enable_dma_interrupt(CRCCU);
```

6.3 Configure and Run CRCCU

The detailed procedures to configure and run CRCCU are described below:

- Before CRC checking, the user should reset CRCCU to make sure the CRC value is initialized to 0xFFFFFFFF

  ```
  crccu_reset(CRCCU);
  ```

- Check the data length (ul_length): If it is larger than MAX_BTSIZE (0xFFFF), then the data buffer should be divided into several blocks and call CRCCU function for multiple times. Several CRCCU operations are needed if the total input data size (in the unit of 32-bit word) is larger than 0xFFFF (256kB).

  ```
  if( ul_length > (MAX_BTSIZE))
  {
    real_calculate_length = (MAX_BTSIZE);
  }
  else
  {
    real_calculate_length = ul_length ;
  }
  ```

- Initialize and configure the CRC descriptor as defined in Section 5.2. The CRC descriptor is located in SRAM with 512 byte aligned.

  ```
  memset((void *)&crc_dscr, 0, sizeof(crccu_dscr_type_t));

  crc_dscr.ul_tr_addr = (uint32_t) p_buffer;

  /* Transfer width: word, interrupt enable */
  ```

```
crc_dscr.ul_tr_ctrl =
        CRCCU_TR_CTRL_TRWIDTH_WORD | real_calculate_length |
        CRCCU_TR_CTRL_IEN_ENABLE;

/* Configure the CRCCU descriptor */
crccu_configure_descriptor(CRCCU, (uint32_t) &crc_dscr);
```

- Configure CRCCU mode: Three modes can be selected. In this demo example, Polynomial 0x04C11DB7 is selected.

```
/* Configure CRCCU mode */
crccu_configure_mode(CRCCU, CRCCU_MR_ENABLE | ul_polynomial_type);
Note: The polynomial type is defined as below in header file crccu.h
#define   CRCCU_MR_PTYPE_CCITT8023 (0x0u << 2) /**< \brief (CRCCU_MR) Polynom 0x04C11DB7 */
#define   CRCCU_MR_PTYPE_CASTAGNOLI (0x1u << 2) /**< \brief (CRCCU_MR) Polynom 0x1EDC6F41 */
#define   CRCCU_MR_PTYPE_CCITT16 (0x2u << 2) /**< \brief (CRCCU_MR) Polynom 0x1021 */
```

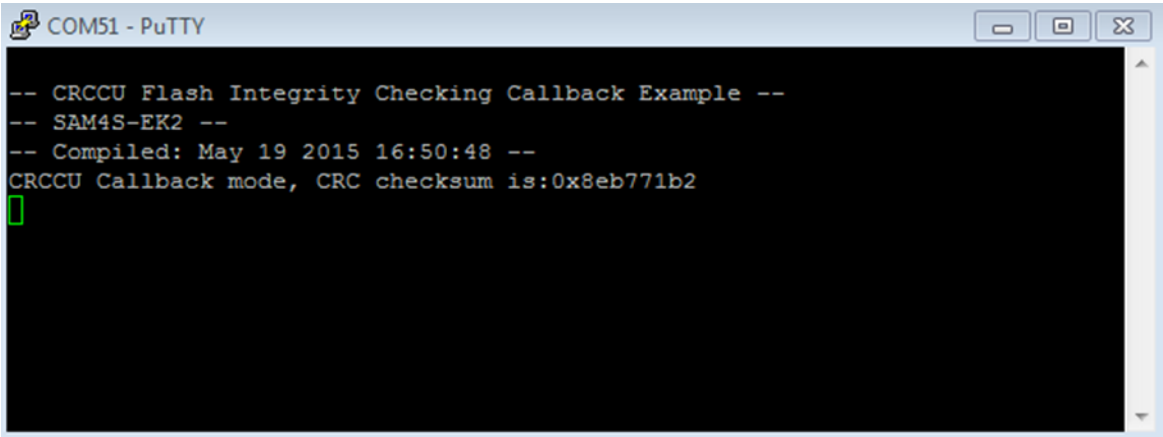
- Enable the CRCCU DMA to start the CRC calculation

```
/* Start the CRC calculation */
crccu_enable_dma(CRCCU);
```

Note: The compute_crc function in Callback mode is slightly different from the function in Polling mode. In Callback mode, the final CRC value is received from the interrupt handler.

6.4 Output of CRCCU Callback Mode

Figure 6-1. Output of CRCCU Callback Mode



7 CRCCU Compared with the Optimized Software CRC Algorithm

The CRCCU_SW_COMPARISON example shows the benefits by comparing the CRCCU with the optimized software CRC algorithm. The SW CRC32 polynomial is 0x04C11DB7, the same polynomial as CRCCU (CCITT802.3).

The SW CRC32 software algorithm utilizes a 1KB look-up table to speed up the SW CRC32 execution while it will increase 1KB Flash space.

Test Conditions:

- Hardware: Atmel SAM4S-EK2 board with ATSAM4SD32C
- System Clock: 8MHz, internal RC used with FW=0
- CRC Configuration: polynomial CCITT802.3 with CRC_INIT:0xFFFFFFFF, CRC_POLY:0x04C11DB7
- Total Checked Flash Size: 32KB
- DMA transfer Width: WORD (32 bits)
- SW CRC32: 1KB loop-up table to speed up the SW CRC32 execution
- System Clock Measurement: TC0, Channel0

7.1 Run Hardware CRCCU and SW CRC32

The detailed procedures to run hardware CRCCU and SW CRC32 are as below:

- Initialize TC0 to do the CPU Cycles measurement.

```
tc_waveform_initialize();
tc_start(TC, TC_CHANNEL_WAVEFORM);
```

Note: TC_CMR_TCCLKS_TIMER_CLOCK4 is used as the TC0 clock source, which means that for one TC count = MCK/128.

- Before running the hardware CRCCU, record the initial TC0 counter value

```
temp_value0 = TC->TC_CHANNEL[0].TC_CV ;
```

- Run hardware CRCCU with polling mode to perform 32KB Flash check
- After hardware CRCCU finished execution, record the current TC0 counter value

```
temp_value1 = TC->TC_CHANNEL[0].TC_CV ;
```

- Output the measurement data consumed by hardware CRCCU to console
- Re-initialize the TC0 to perform the SW CRC32 measurement
- Before running the hardware CRCCU, record the initial TC0 counter value

```
temp_value0 = TC->TC_CHANNEL[0].TC_CV ;
```

- Run SW CRC32 algorithm to perform 32KB Flash check

```
ul_crc1 = calculate_crc32((uint8_t *)TEST_CRCCU_BUF_ADDRESS,TEST_SIZE);//Main function to perform CRC32 by SW
ul_crc1 = reverse32(ul_crc1)^0xFFFFFFFF;/* This should be taken into consideration when using the CRC32 SW solution.*/
```

Note: There are some limitations with hardware CRCCU. The SW CRC32 checksum should first perform 32 bit reverse and then XOR 0xFFFFFFFF, then the result can be the same with hardware CRCCU, and vice versa.

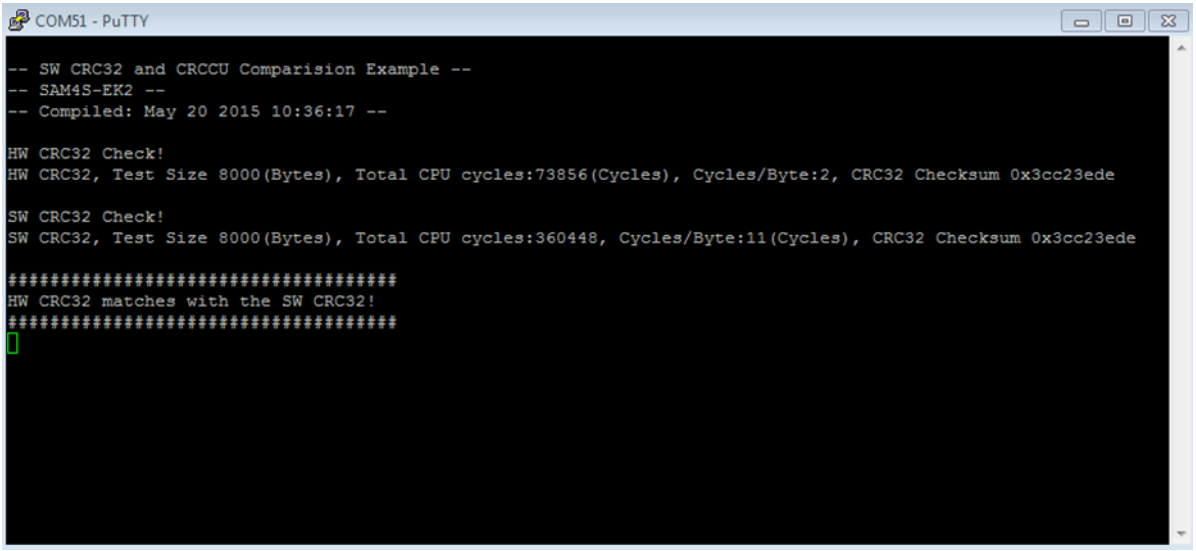
- After SW CRC32 finished execution, record the current TC0 counter value

```
temp_value1 = TC->TC_CHANNEL[0].TC_CV ;
```

- Output the measurement data consumed by SW CRC32 algorithm to console

7.2 Output of CRCCU and SW CRC32

Figure 7-1. Output of CRCCU and SW CRC32 Comparison Data



7.3 CRCCU Usage Benefits Compared with SW CRC32

Table 7-1. Comparison Data Between HW CRCCU and SW CRC32

Flash size	ATSAM4SD32C @8MHz, CRCCU		ATSAM4SD32C @8MHz, SW CRC32	
	CPU cycles	Cycles/bytes	CPU cycles	Cycles/bytes
32KB	73856	2.25	360448	11

In summary, the hardware CRCCU on SAM4S is about five times faster than the optimized SW CRC32 algorithm.

The user can benefit from large performance improvement when using the hardware CRCCU component instead of SW CRC32 algorithm.

8 Revision History

Doc Rev.	Date	Comments
42534A	09/2015	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**