Atmel

APPLICATION NOTE

Getting Started with SAM V71 Microcontrollers

Atmel | SMART SAM V71 Series

Scope

This application note is aimed at helping the reader become familiar with the Atmel® | SMART ARM® Cortex®-M7 based SAM V71 microcontrollers.

It describes in detail a simple project that uses several important features present on SAM V71 chips. This includes how to set up the microcontroller prior to executing the application, as well as how to add the functionalities themselves. After going through this guide, the reader should be able to successfully start a new project from scratch.

This document also explains how to set up and use different toolchains GNU, IAR, and MDK in order to compile and run a software project.

To be able to use this document efficiently, the reader should be experienced in using the ARM core. For more information about the ARM core architecture, please refer to the relevant documents available from www.arm.com.

Reference Documents

Type	Title	Atmel Lit. No.
Datasheet	SAM V71 Datasheet	44003

Table of Contents

1. Requirements 1

2. Getting Started 2

2.1 Specification 2

2.2 Xplained Ultra Board 3

2.3 Implementation 6

3. Running the Examples 16

3.1 GNU 16

3.2 IAR Embedded Workbench 19

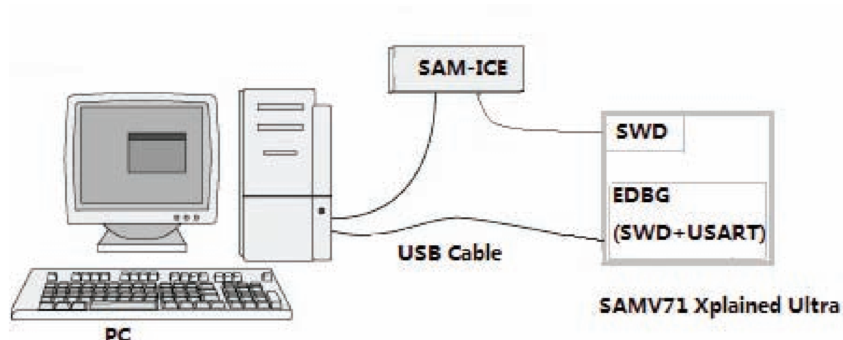
3.3 MDK-ARM 24

4. Revision History 29

1. Requirements

The cross-development environment is shown in Figure 1-1.

Figure 1-1. Development Environment



The software referenced in this application note requires several components:

- SAMV71 Xplained Ultra Evaluation Kit
- One PC running Windows® 7 or higher
- One of the following development tools:
  - IAR Embedded Workbench for ARM (later than V7.30.3)
  - MDK-ARM (later than V5.12)
  - GNU Tools for ARM Embedded Processors (later than V4.8.4)

  Note: MinGW (later than V0.6.2) is necessary for GNU.
- One of the following debuggers:
  - SAM-ICE™ (J-Link) (later than V8.0)

    SEGGER J-Link software & documentation pack (later than V4.96)
  - EDBG (this unit offers SWD and USART port)

    AtmelUSBInstaller.exe (later than 6.2.342)
  - ULINK*pro*™ and ULINK2™ for MDK

In this document, examples are used to guide you in setting up development environments for these tools.

In addition, target programming can also be done using SAM-BA® tools V2.15 or later. For more details, refer to the User Guide available in the SAM-BA package available on www.atmel.com.

2. Getting Started

This section describes how to program a basic application that helps you to become familiar with SAM V71 devices. It covers three main sections: the specification of the example (what it does, what peripherals are used), how to set up hardware development environment and how to control relevant peripherals.

2.1 Specification

2.1.1 Features

The demonstration program makes two LEDs on the board blink at a fixed rate. This rate is generated by using a timer for the first LED; a Wait function based on a 1 ms tick generates the rate for the second LED.

While this software may look simple, it uses several peripherals which make up the basis of an operating system. As such, it serves as a good starting point to become familiar with the SAM V71 microcontroller series.

2.1.2 Peripherals

In order to perform the operations described in the previous section, the software example uses the following set of peripherals:

- Parallel Input/Output (PIO) controller
- Timer Counter (TC)
- System Timer (SysTick)
- Nested Vectored Interrupt Controller (NVIC)
- Serial Port

LEDs and buttons on the board are connected to standard input/output pins of the chip; those are managed by a PIO controller. In addition, it is possible to have the controller generate an interrupt when the status of one of its pins changes; buttons are configured to have this behavior.

The TC and SysTick are used to generate two time bases, in order to obtain the LED blinking rates. They are both used in interrupt mode: the TC triggers an interrupt at a fixed rate, each time toggling the LED state (on/off). The SysTick triggers an interrupt every millisecond, incrementing a variable by one tick; the Wait function monitors this variable to provide a precise delay for toggling the second LED state.

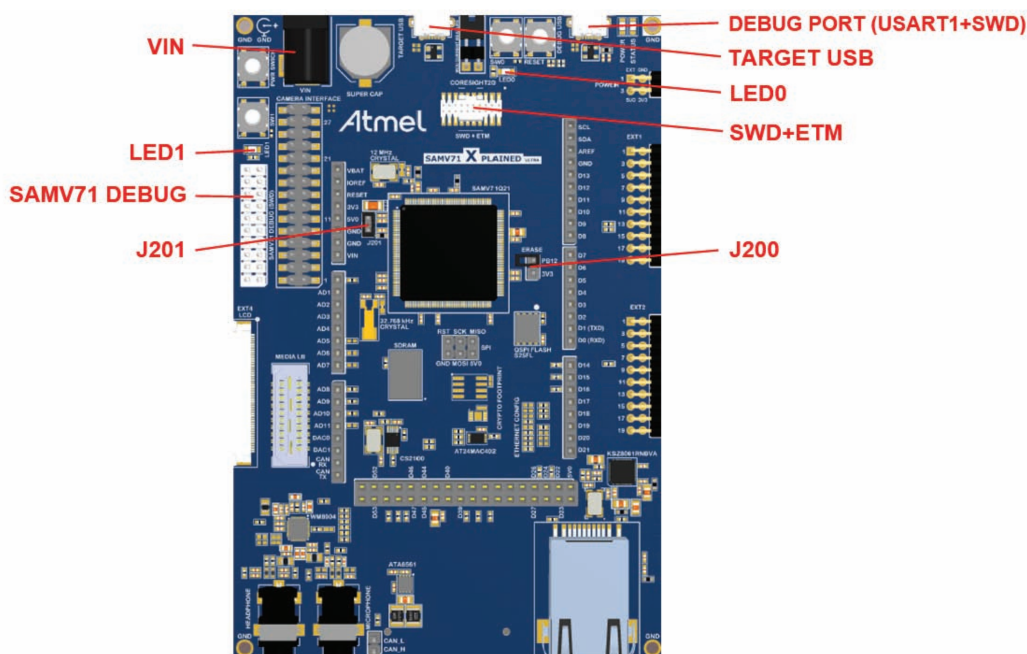
Using the NVIC is required to manage interrupts. It allows the configuration of a separate vector for each source. Three different functions are used to handle PIO, TC and SysTick interrupts.

Finally, an additional peripheral is used to output debug traces on a serial line and is helpful in debugging the program.

2.2 Xplained Ultra Board

2.2.1 Xplained Ultra Board Overview

Figure 2-1. Xplained Ultra Board Overview



2.2.2 Hardware Setup

The following instructions show how to set up the hardware development environment.

1. Jumpers setting: Open all jumpers on the board except J201.
2. Debugger tool connection: Connect the target board with PC through a programmer or debugger tool, EDBG or SAM-ICE.

   SAM-ICE (J-Link)

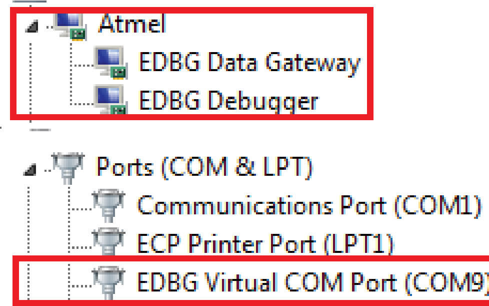
   When downloading and debugging through SWD via SAM-ICE or J-Link, *SEGGER J-Link software & documentation pack* needs to be installed on PC. This is available on https://www.segger.com.

   Connect the SAM-ICE JTAG connector to the 20-pin connector **SAMV71 DEBUG (SWD)** on the board, and connect the PC with SAM-ICE via the USB cable, as shown in Figure 2-2.

   EDBG

   1. Run *AtmelUSBInstaller.exe* to install the Atmel USB driver. This is available on https://gallery.atmel.com.
   2. Connect the target board with PC via a Micro-AB USB cable.

The following devices will show in **Device Manager**.



The USB driver for Atmel and Segger tools makes DEBUG PORT work as both a debugging port and a serial port.

- Serial port connection: necessary for debug information display in the terminal window on PC.

  USART1 can be used as a serial port, which is recognized as **EDBG Virtual COM Port (COMx)** in **Device Manager**. Open a serial debug terminal (such as Putty), and configure the relevant COM port as 115200-8-1-N-N.

- The board is powered on by USB.

  Note: External power is required when the 500 mA through the USB connector is not enough in some applications. Plug the 5V power supply adaptor in **VIN** Jacket.

Figure 2-2 shows the completed hardware set connection.

Figure 2-2. Hardware Connection

2.2.3 Booting

The SAM V71 devices feature up to 2048 Kbytes of embedded Flash and up to 384 Kbytes of internal SRAM. The Getting Started example can be compiled and downloaded to both memories.

The SRAM is accessible over the system Cortex-M bus at address 0x2040 0000 and the base address of the Flash is 0x0040 0000.

2.2.4 Erasing Flash

The user can close the jumper J200, wait for at least five seconds and then re-power the board to chiperase SAM V71.

The ERASE pin on J200 is used to reinitialize the Flash content (and some of its NVM bits) to an erased state (all bits read as logic level 1). It integrates a pull-down resistor of about 100 kΩ to GND, so that it can be left unconnected for normal operations.

When the ERASE pin is tied high during less than 100 ms, it is not taken into account. The pin must be tied high during more than 220 ms to perform a Flash erase operation.

To make sure that the erase operation is performed after power-up, the system must not reconfigure the ERASE pin as GPIO or enter Wait mode with Flash in Deep Power-down mode before the ERASE pin assertion time has elapsed. For more details, refer to the SAM V71 datasheet.

2.2.5 LEDs

There are two general-purpose LEDs (yellow) on the SAM V71 Xplained Ultra board. They are wired to pins PA23 and PC9. Setting a logical low or high level on the corresponding PIO lines turns the LEDs on and off.

The example application uses both LEDs (PA23 and PC9).

2.2.6 Serial Port

On SAM V71, the default serial port which is used to print debug information and monitor input is USART1. The port uses pins PA21 and PB04 for the RXD1 and TXD1 signals, respectively.

2.3 Implementation

As stated previously, the example defined above requires the use of several peripherals. It must also provide the necessary code for starting up the microcontroller. Both aspects are described in detail in this section, with commented source code when appropriate.

2.3.1 Initialization Before ‘main’

After the board is powered on, ROM code will run and carry out some necessary initialization. The ROM code process will not be discussed in detail here.

Most of the code of an embedded application is written in C. This makes the program easier to understand, more portable and modular.

When downloading the application via the J-link GDB server, users can set the registers of PC and stack pointer to 0x2040 0000 and 0x2040 0004 respectively in the GDB script for SAM V71 Xplained Ultra. Before you run the application, you might still want to:

- Provide exception vectors
- Initialize critical peripherals
- Initialize memory segments

These initialization requirements are described in the next sections.

2.3.1.1 Entry Point

For GNU toolchain, PC points to the start address of *Reset_Handler* at the beginning.

For IAR and MDK, PC points to the start address of *__iar_program_start* and *Reset_Handler*, respectively.

The purpose of the entry point is to:

- Set up a C environment
- Set the vector table base address
- Perform the low-level initialization
- Jump to the main application

2.3.1.2 Low-Level Initialization

Starting from the *LowLevelInit* interface, three toolchains share program flowcharts.

The first step of the low-level initialization process is to configure critical peripherals:

- Main oscillator and its PLL
- MPU
- TCM

The *LowLevelInit* function is shown as follows.

```
extern WEAK void LowLevelInit( void )
{
     SystemInit();
    _SetupMemoryRegion();
#ifdef ENABLE_TCM
    FLASHD_ClearGPNVM(8);
    FLASHD_SetGPNVM(7);
    TCM_Enable();
#else
    TCM_Disable();
#endif
}
```

The following sections explain why these peripherals are considered critical, and detail the required operations to configure them properly.

2.3.1.3 Low-Level Initialization: SystemInit

The main function of *SystemInit* is to complete the processor clock and master clock configuration.

After reset, the Main RC oscillator is enabled with the 12 MHz frequency selected and it is selected as the source of MAINCK. MAINCK is the default clock selected to start the system.

The Main oscillator and its Phase Lock Loop A (PLLA) must be configured in order to run at full speed. Both can be configured in the Power Management Controller (PMC). For details, refer to the SAM V71 datasheet.

In the example, the processor clock and master clock are 300 MHz and 150 MHz respectively by default. Example values on the SAMV71 Xplained Ultra (12 MHz crystal):

```
#define SYS_BOARD_PLLAR   (CKGR_PLLAR_ONE            |\
                          CKGR_PLLAR_MULA(0x18U)     |\
                          CKGR_PLLAR_PLLACOUNT(0x3fU) |\
                          CKGR_PLLAR_DIVA(0x1U))

#define SYS_BOARD_MCKR    (PMC_MCKR_PRES_CLK_1       |\
                          PMC_MCKR_CSS_PLLA_CLK    |\
                          PMC_MCKR_MDIV_PCK_DIV2)
```

Here:

f_{input} = 12 MHz

MAINCK = 12 MHz

PLLACK = MAINCK * MULA / DIVA = (12 * (0 × 18 + 1) / 1) MHz = 300 MHz

HCLK = PLLACK / PRES = (300 / 1) MHz = 300 MHz

MCK = PLLACK / PRES / MDIV = (300 / 1 / 2) MHz = 150 MHz

In addition, the user must set the number of wait states of the embedded Flash depending on the system frequency. When MCK is 4 MHz and FWS is 0, the number of cycles for Read/Write operations is 1.

In the example, defining FWS as 5 enables six cycles access, which is done as shown below:

```
EFC->EEFC_FMR = EEFC_FMR_FWS(5);
```

For more details, see the “Embedded Flash Wait State” table in the SAM V71 datasheet.

2.3.1.4 Low-Level Initialization: Memory Protection Unit (MPU)

The SAM V71 devices supply MPU with 16 zones as a component for memory protection.

Users can use the MPU to enforce privilege rules, separate processes and enforce access rules.

The *_SetupMemoryRegion* function completes the memory mapping by setting MPU Region Base Address Register (RBAR) and MPU Region Attribute and Size Register (RASR).

The MPU_RASR.ATTRS field defines the memory type, the cacheable and shareable properties, and the access and privilege properties of the memory region.

The System Handler Control and State Register is settled to enable memory management fault, Bus Fault, and Usage Fault exception.

At the end of the function, MPU region is enabled by setting MPU Control register.

In the example, memory regions such as ITCM, internal Flash, DTCM, SRAM, peripheral memory, SDRAM, QSPI memory and USBHS_RAM are all configured in this function.

The SRAM, for example, is divided into two parts with the same attributes.

```
void MPU_SetRegion( uint32_t dwRegionBaseAddr, uint32_t dwRegionAttr )
{
    MPU->RBAR = dwRegionBaseAddr;
    MPU->RASR = dwRegionAttr;
}
void _SetupMemoryRegion( void )
{
    uint32_t dwRegionBaseAddr;
    uint32_t dwRegionAttr;
    …………

    dwRegionBaseAddr =
        SRAM_PRIVILEGE_START_ADDRESS |
        MPU_REGION_VALID |
        MPU_DEFAULT_PRAM_REGION; //4
    dwRegionAttr =
        MPU_AP_FULL_ACCESS    |
        INNER_NORMAL_WB_NWA_TYPE( NON_SHARABLE ) |
      MPU_CalMPURegionSize(SRAM_PRIVILEGE_END_ADDRESS -
        SRAM_PRIVILEGE_START_ADDRESS) |
        MPU_REGION_ENABLE;

    MPU_SetRegion( dwRegionBaseAddr, dwRegionAttr);

    dwRegionBaseAddr =
        SRAM_UNPRIVILEGE_START_ADDRESS |
        MPU_REGION_VALID |
        MPU_DEFAULT_UPRAM_REGION; //5

    dwRegionAttr =
        MPU_AP_FULL_ACCESS    |
        INNER_NORMAL_WB_NWA_TYPE( NON_SHARABLE ) |
        MPU_CalMPURegionSize(SRAM_UNPRIVILEGE_END_ADDRESS -
        SRAM_UNPRIVILEGE_START_ADDRESS) |
        MPU_REGION_ENABLE;

    MPU_SetRegion( dwRegionBaseAddr, dwRegionAttr);
    …………
/* Enable the memory management fault, Bus Fault, Usage Fault exception */
SCB->SHCSR |= (SCB_SHCSR_MEMFAULTENA_Msk | SCB_SHCSR_BUSFAULTENA_Msk |
SCB_SHCSR_USGFAULTENA_Msk);
 /* Enable the MPU region */
    MPU_Enable( MPU_ENABLE | MPU_BGENABLE );
}
```

The user can configure a new memory region or adjust the attributes of some regions in the function, such as the cacheable properties.

2.3.1.5 Low-Level Initialization: Tightly Coupled Memory (TCM)

The SAM V71 devices embed Tightly Coupled Memory (TCM) running at processor speed.

ITCM is a single 64-bit interface, based at 0x0000 0000 (code region) and DTCM is composed of dual 32-bit interfaces interleaved, based at 0x2000 0000 (data region).

ITCM is disabled by default at reset. DTCM is enabled by default at reset with the size programmed in GPNVM bits [8:7]. When enabled, ITCM is located at 0x0000 0000, overlapping ROM or Flash depending on the general-

purpose NVM bit 1 (GPNVM). The TCM configuration can be modified with GPNVM bits [8:7]. The user can program them through the "Clear GPNVM Bit" and "Set GPNVM Bit" commands of the EEFC User Interface.

Use the following codes to configure TCM to 32 Kbytes and enable it:

```
FLASHD_ClearGPNVM(8);
FLASHD_SetGPNVM(7);
TCM_Enable();
```

Accesses made to TCM regions when the relevant TCM is disabled and accesses made to the Code and SRAM region above the TCM size limit are performed on the AHB matrix, i.e., on internal Flash or on ROM depending on remap GPNVM bit.

Accesses made to the SRAM above the size limit will not generate aborts.

The Memory Protection Unit (MPU) can be used to protect these areas as mentioned in Section 2.3.1.4.

Note that internal SRAM and TCM share the same memory space, which means that when TCM is enabled, the size available as internal SRAM is reduced proportionally.

After carrying out all of the above initialization actions, the program can jump to the main application.

2.3.2 Generic Peripheral Usage

2.3.2.1 Initialization

Most peripherals are initialized by performing the following actions:

- Disabling or reprogramming watchdog
- Enabling cache if necessary
- Enabling the peripheral clock in the PMC if necessary
- Enabling the control of the peripheral on PIO pins
- Enabling the interrupt source at the peripheral level

2.3.3 Disabling or Reprogramming Watchdog Timer (WDT)

2.3.3.1 Purpose

The Watchdog Timer (WDT) is used to prevent system lock-up if the software becomes trapped in a deadlock. It features a 12-bit down counter that allows a watchdog period of up to 16 seconds (slow clock around 32 kHz). It can generate a general reset or a processor reset only. In addition, it can be stopped while the processor is in debug mode or idle mode.

After a processor reset, the value of Watchdog Counter Value (WDV) is 0xFFF, which corresponds to the maximum value of the counter with the external reset generation enabled (bit WDT_MR.WDRSTEN at 1 after a backup reset). This means that a default watchdog is running at reset, i.e., at power-up. The user can either disable the WDT by setting bit WDT_MR.WDDIS or reprogram the WDT to meet the maximum watchdog period the application requires.

2.3.3.2 Initialization

In the example, the user can disable WDT with the *WDT_Disable* function as follows.

```
WDT_Disable( WDT ) ;
```

The operation is done by setting WDT_MR as follows.

```
pWDT->WDT_MR = WDT_MR_WDDIS;
```

2.3.4 Enabling Cache If Necessary

2.3.4.1 Purpose

The SAM V71 devices support 16 Kbytes of ICache and 16 Kbytes of DCache with Error Code Correction (ECC). All caches are disabled at reset and enabling cache benefits the performance. The user can turn on I-Cache and D-Cache if necessary.

2.3.4.2 Initialization

The user can enable cache as follows:

```
SCB_EnableICache();
SCB_EnableDCache();
```

The details of the *SCB_EnableICache* function is shown below as an example.

```
SCB->ICIALLU = 0;    // Invalidate I-Cache
SCB->CCR |=  SCB_CCR_IC_Msk;// Enable I-Cache
```

To make some regions cacheable, the following conditions should all be met:

- Enable cache as described above in the application.
- Set the attributes of the relevant regions as cacheable in *_SetupMemoryRegion* function (refer to Section 2.3.1.4).

2.3.4.3 Cache *Coherency*

Enabling cache may cause breakdown when:

- Memory locations are updated by other agents in the system.
- Memory updates made by the application code must be made visible to other agents in the system.

For example, in a system with a DMA that reads memory locations held in the data cache of a processor, a breakdown of coherency occurs when the processor has written new data in the data cache, but the DMA reads the old data held in memory.

In situations where a breakdown in coherency occurs, the software must manage the caches by using cache maintenance operations. The Clean, Invalidate and Clean, and Invalidate operations can address these issues.

Take DCache as an example, these operations are realized in several functions such as:

```
static inline void SCB_InvalidateDCache();
static inline void SCB_CleanDCache ();
static inline void SCB_CleanInvalidateDCache ();
```

2.3.5 Using the Nested Vectored Interrupt Controller (NVIC)

2.3.5.1 Purpose

The NVIC provides configurable interrupt handling abilities to the processor. It facilitates low-latency exception and interrupt handling, and controls power management.

The NVIC supports up to 240 interrupts, each with up to 256 levels of priority. The user can change the priority of an interrupt dynamically. The NVIC and the processor core interface are closely coupled, to enable low-latency Interrupt processing and efficient processing of late arriving interrupts. The NVIC maintains knowledge of the stacked, or nested interrupts to enable tail-chaining of interrupts.

2.3.5.2 Initialization

The SAM V71 uses hardware to save and restore key context state on exception entry and exit, and use a table of vectors to indicate the exception entry points.

The vector table contains the initialization values for the stack pointer, and the entry point addresses of each exception handler. The vector table is defined as the constant of '*exception_table'* for GNU toolchain.

Part of the constant is shown as follows:

```
__attribute__ ((section(".vectors")))
const DeviceVectors exception_table = {
        .pvStack = (void*) (&_estack),

        .pfnReset_Handler      = (void*) Reset_Handler,
        .pfnNMI_Handler        = (void*) NMI_Handler,
        .pfnHardFault_Handler  = (void*) HardFault_Handler,
        ………..
        .pfnSysTick_Handler    = (void*) SysTick_Handler,
        ………..
        .pfnTC0_Handler        = (void*) TC0_Handler,
        ………..
}
```

On reset, the processor initializes the vector table base address to an IMPLEMENTATION DEFINED address. The software can find the current location of the table, or relocate the table, using the Vector Table Offset Register (VTOR) as shown below.

```
pSrc = (uint32_t *) & _sfixed;
SCB->VTOR = ((uint32_t) pSrc & SCB_VTOR_TBLOFF_Msk);
```

The *_sfixed* symbol points to the vectors section which saves the vectors table. The SCB_VTOR_TBLOFF_Msk is equal to 0xFFF FFF8 on SAM V71 and bits [6:0] are RAZ (Read as Zero).

The VTOR holds the vector table address.

The processor and the NVIC prioritize and handle all exceptions. When handling exceptions, all exceptions are handled in Handler mode, and processor state is automatically stored to the stack on an exception, and automatically restored from the stack at the end of the Interrupt Service Routine (ISR). The vector is fetched in parallel to the state saving, enabling efficient interrupt entry.

Configuring an interrupt source requires six steps:

1. Implement interrupt handler if necessary.

   The first step is to re-implement the interrupt handler with the same name as the default interrupt handler in the vector table as just mentioned if necessary, so that when the corresponding interrupt occurs, the reimplemented interrupt handler will be executed instead of the default interrupt handler.

2. Disable the interrupt if it was enabled.

   An interrupt triggering before its initialization completion may result in unpredictable behavior of the system. To disable the interrupt, the Interrupt Clear-Enable Register (ICER) of the NVIC must be written with the interrupt source ID to mask it. The following interface can be used directly:

   ```
   static inline void NVIC_DisableIRQ(IRQn_Type IRQn);
   ```

3. Clear any pending interrupt.

   Setting the Interrupt Clear-Pending Register bit puts the corresponding pending interrupt in the inactive state. It is also written with the interrupt source ID to mask it. The following interface can be used directly:

   ```
   static inline void NVIC_ClearPendingIRQ(IRQn_Type IRQn);
   ```

4. Configure the interrupt priority.

   NVIC interrupts are prioritized by updating an 8-bit field within a 32-bit register (each register supporting four interrupts). Priorities are maintained according to the ARMv7-M prioritization scheme. The following interface can be used directly:

   ```
   static inline void NVIC_SetPriority(IRQn_Type IRQn,uint32_t priority);
   ```

5. Enable the interrupt at peripheral level.
6. Enable the interrupt at NVIC level.

The interrupt source can be enabled, both on the peripheral (in a mode register usually) and in the Interrupt Set-Enable Register (ISER) of the NVIC. On the side of NVIC, the following interface can be called directly:

```
static inline void NVIC_EnableIRQ(IRQn_Type IRQn);
```

Refer to *core_cm7.h* for more interfaces about NVIC which can be used directly.

2.3.6 Using the Timer Counter (TC)

2.3.6.1 Purpose

Timer Counters on SAM devices can perform several functions, e.g., frequency measurement, pulse generation, delay timing, and Pulse Width Modulation (PWM).

In this example, a single Timer Counter (TC) channel is going to provide a fixed-period delay. An interrupt is generated each time the timer expires, toggling the associated LED on or off. This makes the LED blink at a fixed rate.

2.3.6.2 Initialization

In order to reduce power consumption, most peripherals are not clocked by default. Writing the ID of a peripheral in the PMC Peripheral Clock Enable Register (PMC_PCERx) activates the peripheral clock.

The TC initialization sequence is the following:

1. Write the ID of the TC in the PMC Peripheral Clock Enable Register (PMC_PCERx):

```
PMC_EnablePeripheral(ID_TC0);
```

2. Configure TC as 4 Hz frequency by calling the function *TC_FindMckDivisor* which will find the best MCK divisor. The best divisor depends on the timer frequency and MCK.

```
TC_FindMckDivisor( 4, BOARD_MCK, &div, &tcclks, BOARD_MCK );
```

3. Configure the TC Channel Mode Register (TC_CMRx). TC channels can operate in different modes. In the example, set the TC in Capture mode by clearing the WAVE bit and enable RC Compare Trigger by setting the CPCTRG bit, , which is done in the internal *TC_Configure* function:

```
TC_Configure( TC0, 0, tcclks | TC_CMR_CPCTRG );
```

4. Configure the interrupt whenever the counter reaches the value programmed in RC. The interrupt priority is level 0 as default. At the TC level, this is done by setting the CPCS bit of the TC Interrupt Enable Register (TC_IERx):

```
NVIC_ClearPendingIRQ(TC0_IRQn);
NVIC_EnableIRQ(TC0_IRQn);
TC0->TC_CHANNEL[ 0 ].TC_IER = TC_IER_CPCS ;
```

At the end of the sequence, the program starts the counter if LED1 is enabled as shown below:

```
if ( bLed1Active ) {
    TC_Start( TC0, 0 );
}
```

2.3.6.3 Interrupt Handler

The interrupt handler for TC0 interrupt is '*TC0_Handler'* and the main purpose is to toggle the state of LED.

The first action to do in the handler is to acknowledge the pending interrupt from the peripheral. Otherwise, the latter continues to assert the IRQ line. In the case of a TC channel, acknowledging is done by reading the corresponding TC Status register (TC_SRx). The code is shown as below.

```
dummy = TC0->TC_CHANNEL[ 0 ].TC_SR;
```

It toggles the state (on or off) of one of the blinking LEDs by programming the PIO controller.

```
LED_Toggle( 1 );
```

Refer to Section 2.3.8.3 for more details.

2.3.7 Using the System Timer (SysTick)

2.3.7.1 Purpose

The system timer, SysTick, provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism.

This Getting Started example uses the SysTick to provide a 1 ms time base. Each time the interrupt is triggered, a 32-bit counter is added. A Wait function uses this counter to provide a precise way for an application to suspend itself for a specific amount of time.

2.3.7.2 Initialization

Initialization is done with the following line of code:

```
SysTick_Config( Pck/1000);
```

In this example, the SysTick clock source is the processor clock and PCK = BOARD_MCK*2. The function initializes the system timer and its interrupt, and starts the System Timer.

2.3.7.3 Interrupt Handlers

The handler for system timer is shown as follows:

```
static volatile uint32_t _dwTickCount = 0 ;
void SysTick_Handler( void )
{
    _dwTickCount ++;
    …………
}
```

Using a 32-bit counter may not always be appropriate, depending on how long the system should stay up and on the tick period. In this example, a 1 ms tick overflows the counter after about 50 days; this may not be enough for a real application. In that case, a larger counter can be implemented.

2.3.7.4 Wait Function

Using the global counter, it is very easy to implement a wait function taking a number of milliseconds as its parameter. Read the code for more details.

When called, the function first saves the current value of the global counter in a local variable. It adds the requested number of milliseconds which has been given as an argument. Then, it simply loops until the global counter becomes equal to or greater than the computed value.

The interface can be called directly to wait for several milliseconds. In this example, it is called as follows to wait for 1000 ms:

```
Wait(1000);
```

2.3.8 Using the Parallel Input/Output Controller (PIO)

2.3.8.1 Purpose

The SAM V71 devices support up to five PIO controllers and each one controls up to 32 lines. Each line can be assigned to one of four peripheral functions: A, B, C or D.

In this example, the PIO controller manages two LEDs.

2.3.8.2 Configuring LEDs

The two PIOs connected to the LEDs must be configured as output, in order to turn them on or off. First, the PIOs control must be enabled in PIO Enable Register (PIO_PER) by writing the value corresponding to a logical OR between the two LED IDs.

PIO direction is controlled by two registers: Output Enable Register (PIO_OER) and Output Disable Register (PIO_ODR). Since in this case the two PIOs must be output, the same value as before shall be written in PIO_OER.

Note that there are individual internal pull-ups on each PIO pin. These pull-ups are enabled by default. Since they are useless for driving LEDs, they should be disabled, as this reduces power consumption. This is done in the PIO Pull-Up Disable Register (PIO_PUDR).

In this example, LEDs are wired to pins PA23 and PC9. They are described in the following macros:

```
#define PIN_LED_0   {PIO_PA23, PIOA, ID_PIOA, PIO_OUTPUT_0, PIO_DEFAULT}
#define PIN_LED_1   {PIO_PC9 , PIOC, ID_PIOC, PIO_OUTPUT_0, PIO_DEFAULT}
```

Here is the code for LED configuration:

```
LED_Configure( 0 ) ;
LED_Configure( 1 ) ;
```

PIO_Configure will be called as shown below:

```
PIO_Configure( &pinsLeds[dwLed], 1 );
```

For LED0 wired to pin PA23, the program will run *PIO_SetPeripheralA* function. More details are available from the source code.

When programming the PIO, it is recommended to call *PIO_Configure* with proper parameters directly. So, the definition of relevant pins such as *PIN_LED_0* and *PIN_LED_1* is primary.

2.3.8.3 Controlling LEDs

LEDs are turned on or off by changing the level on the PIOs to which they are connected. After those PIOs have been configured, their output values can be changed by writing the pin IDs in the PIO Set Output Data Register (PIO_SODR) and the PIO Clear Output Data Register (PIO_CODR).

In addition, the PIO Pin Data Status Register (PIO_PDSR) indicates the current level on each pin. It can be used to create a toggle function, i.e., when the LED is ON according to PIO_PDSR, then it is turned off, and vice-versa. The function is described below. *PIO_GetOutputDataStatus* returns the value of PIO_PDSR. The relevant interfaces which can be called directly are listed as follows:

```
unsigned char PIO_GetOutputDataStatus(const Pin *pin);
void PIO_Clear(const Pin *pin);
void PIO_Set(const Pin *pin);
```

Refer to pio.c or pio.h for more interfaces that can be used directly.

2.3.9 Using the Serial Ports

2.3.9.1 Purpose

As mentioned before, the default serial port used as output is USART1 on the SAM V71 Xplained Ultra board.

Run *AtmelUSBInstaller.exe* to make sure that the PC recognizes the port as a serial port and then connect the board with the PC via a Micro-AB USB cable.

The example application uses USART1 to print debug information and monitor input.

2.3.9.2 Initialization

The common interfaces which can be called directly are listed as below.:

```
extern void DBG_PutChar( uint8_t c );
extern uint32_t DBG_GetChar( void  );
```

Their purpose is to output a character and input a character via the serial port, respectively. At the first of these functions, the program will check whether the console has been initialized by a static variable '*_ucIsConsoleInitialized*'. If not, *DBG_Configure* function will be called and finishes initialization and configuration.

```
        if ( !_ucIsConsoleInitialized )
        {
            DBG_Configure(CONSOLE_BAUDRSATE, BOARD_MCK);
        }
```

It is configured with a baudrate of 115200, 8 bits of data, no parity, one stop bit and no flow control as default.

2.3.9.3 Redirecting printf

The function *printf* is redirected to the serial port in software package.

For IAR toolchain, *putchar* is called by *printf*, so *putchar* is redefined by calling *DBG_PutChar* which outputs a character on the serial port as shown below:

```
extern WEAK signed int putchar( signed int c )
{
    DBG_PutChar( c ) ;
    return c ;
}
```

For MDK toolchain, the relevant interface is *fputc.* It is performed as follows:

```
int fputc(int ch, FILE *f)
{
    if ((f == stdout) || (f == stderr))
    {
        DBG_PutChar(ch) ;
        return ch ;
    }
    else
    {
        return EOF ;
    }
}
```

For GNU toolchain, the relevant interface is *_write.* It is performed as follows:

```
extern int _write( int file, char *ptr, int len )
{
    int iIndex ;
    for ( iIndex=0 ; iIndex < len ; iIndex++, ptr++ )
    {
        DBG_PutChar( *ptr ) ;
    }
    return iIndex ;
}
```

3. Running the Examples

The required software tools for building the project and downloading the binary files are introduced in Section 1.

The hardware development environment setup is described in Section 2.2.2.

3.1 GNU

To generate the binary file to be downloaded into the target, we use the *GNU Tools for ARM Embedded Processors*.

3.1.1 Set up a Development and Debug Environment

3.1.1.1 Setting up Development Tools

Note: For GNU toolchain, *GNU Tools for ARM Embedded Processosr* and *MinGW* are necessary as mentioned above. These tools are available on https://launchpad.net/gcc-arm-embedded/+download and http://sourceforge.net/projects/mingw/files/, respectively.

3.1.1.2 Setting up Necessary Tools

In this section, choose SAM-ICE (J-Link) as the debugger and J-Link GDB Server will be used.

When downloading and debugging via SAM-ICE (J-Link), *SEGGER J-Link software & documentation pack* must be installed on your PC.

3.1.2 Generating and Downloading Binary file

3.1.2.1 Building and Downloading

Makefile is the most important file while building the code.

The Makefile contains rules indicating how to assemble, compile and link the project source files to create a binary file ready to be downloaded on the target.

The makefile is divided into two parts, one for variables settings, and the other for rules implementation.

Variables

The first part of the Makefile contains variables (uppercase), used to set up some environment parameters, such as the compiler toolchain prefix and program names, and options to be used with the compiler.

For example,

```
TRACE_LEVEL = 4
```

- Defines trace level used for compilation.

```
OPTIMIZATION = -O0
```

- Level of optimization used during compilation. It is recommended to set as *OPTIMIZATION = -O0* when debugging.

```
C_OBJECTS += board_lowlevel.o
……
C_OBJECTS += main.o
```

- List of object files. The user should add corresponding C_OBJECTS variable here if *.c needs to be added.

Rules

The second part of the Makefile contains rules. Each rule is composed on the same line by a target name, and the files needed to create this target.

The first rule, ‘all’, is the default rule used by the make command if none is specified in command line.

Both SRAM and Flash configurations are compiled and binaries are generated.

```
all: $(BIN) $(OBJ) $(MEMORIES)
```

The rule, ‘$(1)’, describes how to compile source files, and link object files and library together to generate one binary file per configuration: program running in FLASH and program running in SRAM.

```
$(1): $$(ASM_OBJECTS_$(1)) $$(C_OBJECTS_$(1))
```

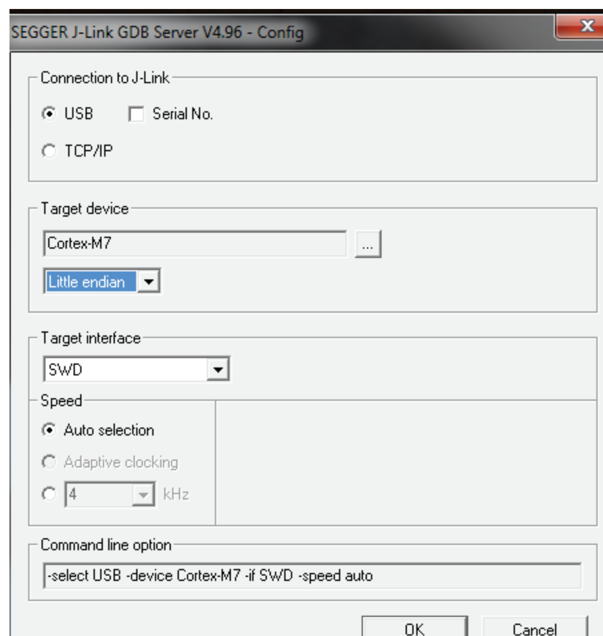
When debugging the example with GDB, the user can use the ‘debug_$(1)’ directly. The rule will use a script named "samv7-ek-sram.gdb" which is provided in the package.

```
debug_$(1): $(1)
```

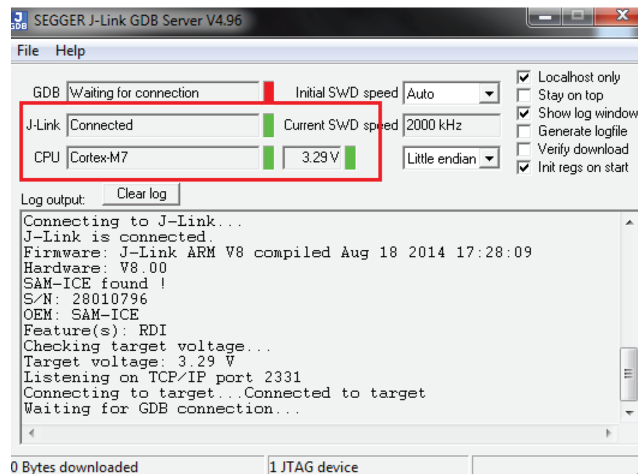
The rule, ‘clean’, is also supplied in the default makefile. The user can adjust makefile as required.

In the GDB example, for SRAM mode, follow the steps below to build, download and debug via J-Link GDB Server.

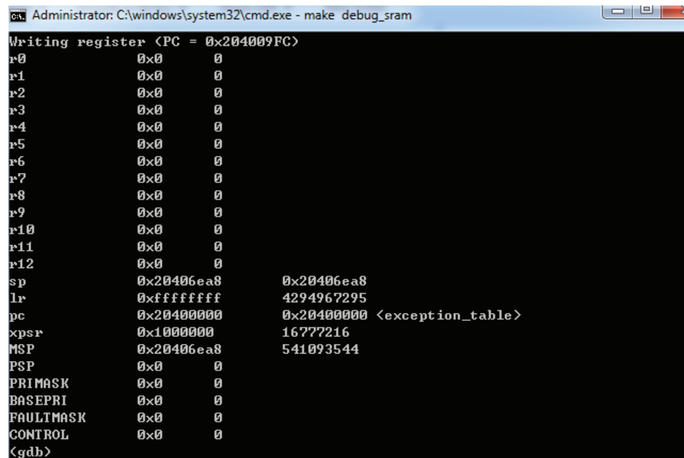
1. Connect the PC and the board via SAM-ICE with one USB cable and one 20-pin JTAG cable.
2. Serial Port connection: for printing debug information in the terminal on PC.
3. Power on the board.
4. Open the J-Link GDB Server on PC with proper configurations as follows.



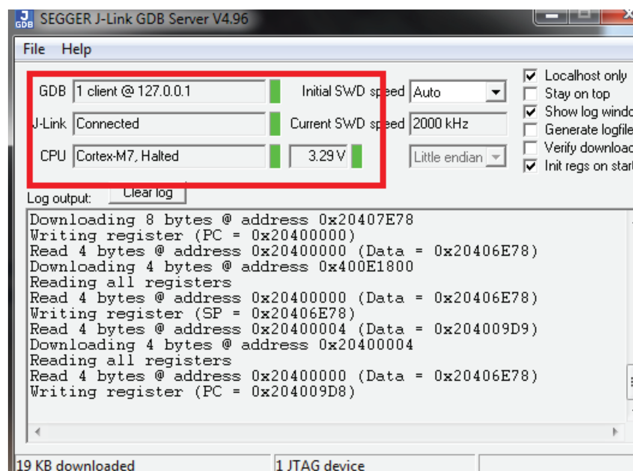
Click ‘OK’. The option “J-Link” and “CPU” in the GDB Server window will display “Connected” and “Cortex-M7”, with green indicators.



5. Go to the “\..*examples\getting-started\build\gcc*“ directory and build the example by typing “make debug_sram”. The binary is then generated and downloaded into the board.



The program starts running and will stop at the entry point of the program. At this time, the “GDB” indicator in the J-Link GDB Server window turns green.



6. Type commands to debug. For example, “c” to continue, “b main” to set a breakpoint before the *main* function.

If the program continues running without breakpoints, you will see LEDs blinking, which can be controlled by Serial Port input. Press ‘1’ to Start/Stop the LED0 blinking and press ‘2’ to Start/Stop the LED1 blinking.

For more detailed information, refer to relevant documents from gcc.gnu.org.

3.2 IAR Embedded Workbench

Note that the Getting Started example has already been ported and included in IAR® EWARM.

3.2.1 Set up a Development and Debug Environment

3.2.1.1 Setting up Development Tools

- Install *IAR Embedded Workbench*. This is available at www.iar.com.
- Install the software package by running *SAMV71_softpack_x.x_for_ewarm_7.30.exe*.

Now, the user can generate binary corresponding to SAM V71 devices via IAR.

3.2.1.2 Setting up Necessary Tools

In this section, choose EDBG or SAM-ICE (J-Link) as the debugger.

When EDBG is selected, *AtmelUSBInstaller* should be installed to enable EDBG SWD port.

When downloading and debugging via SAM-ICE (J-Link), *SEGGER J-Link software & documentation pack* needs to be installed on PC.

3.2.2 Generating and Downloading Binary File

3.2.2.1 Building the Project

Before building the project, it is necessary to check the project configuration.

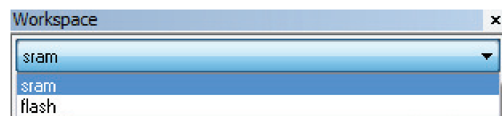
Follow the steps below:

1. Open the workspace “getting-started.eww” in directory:

\..*examples\Atmel\samv7-xPlained\examples\getting-started\build\ewarm*\.

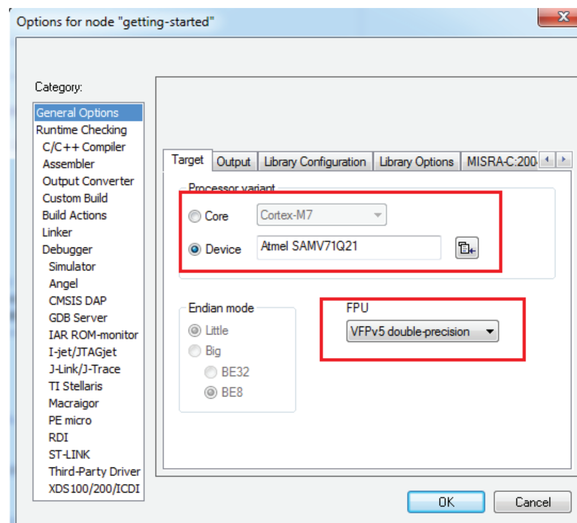
2. Select Flash or SRAM

   Two modes (flash and sram) can be selected in the drop-down menu at the up-left corner of the project window shown as below.

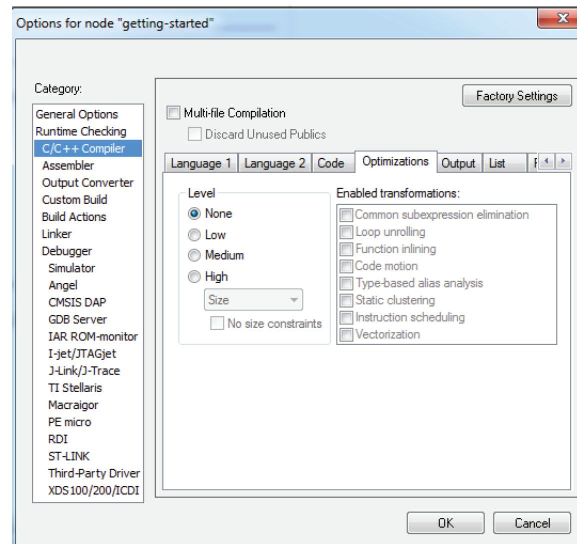


3. Check the configurations. The key settings are shown as follows:

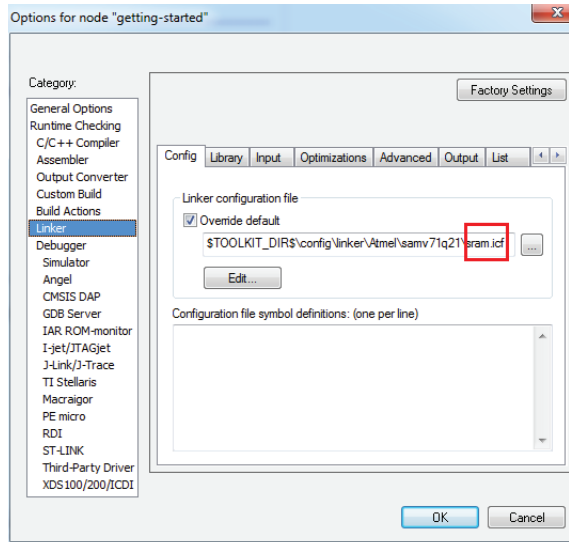
- “General Options”: select “Atmel SAMV71Q21” as Device (or select “Cortex-M7” as Core). Select “VFPv5 double-precision” as FPU setting.



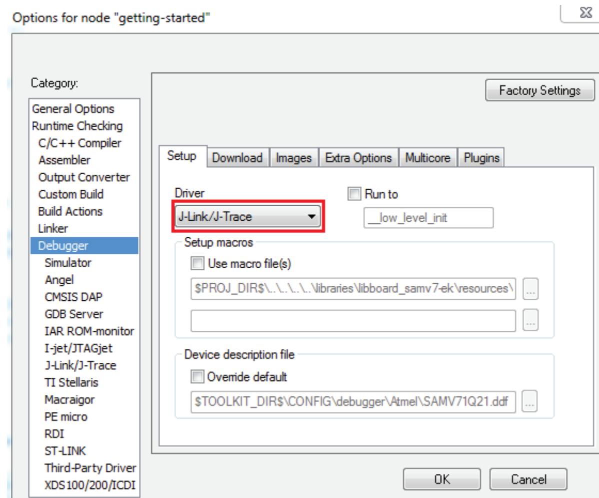
- “C/C++ Compiler”: select Optimizations setting as required.

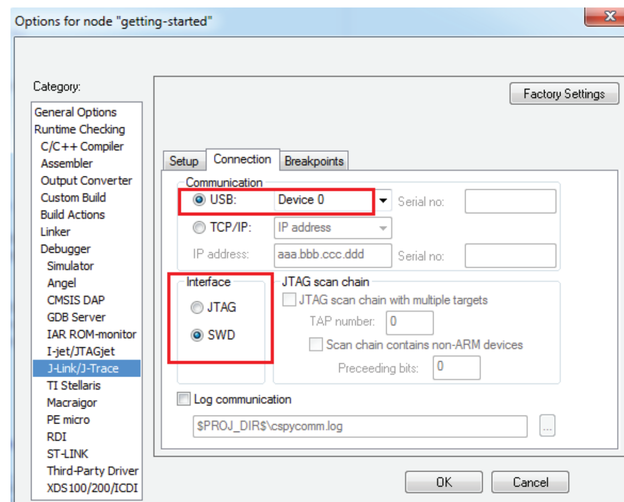


- ”Linker”: Linker configuration file must be “flash.icf” (when in Flash mode) or “sram.icf” (when in sram mode).

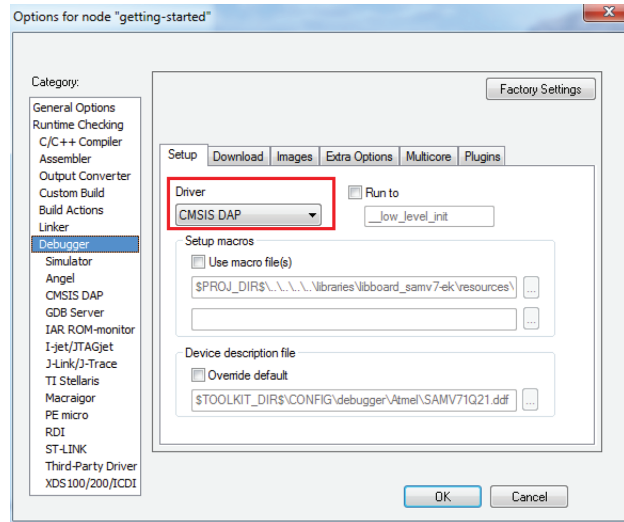


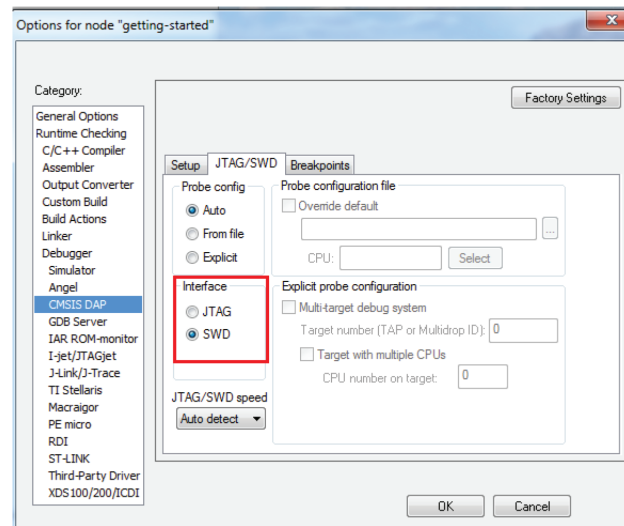
- ”Debugger”: either SAM-ICE or EDBG can be selected as the debugger to download the binary file generated by IAR. The corresponding configurations are shown below respectively.
  – Using SAM-ICE (J-Link) as the debugger





– Using EDBG as the debugger





In addition, make sure the 20-pin *SAMV71 DEBUG (SWD)* connector is unconnected when EDBG is used as the debugger.

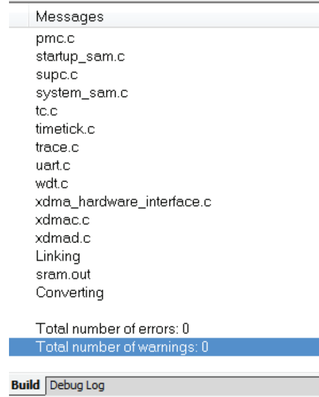
4. Build the Project

Click the **Make** button or click **Make** after right-clicking the project name to build the project.

The executable binary files: sram.bin and sram.out will be generated in directory:

Atmel

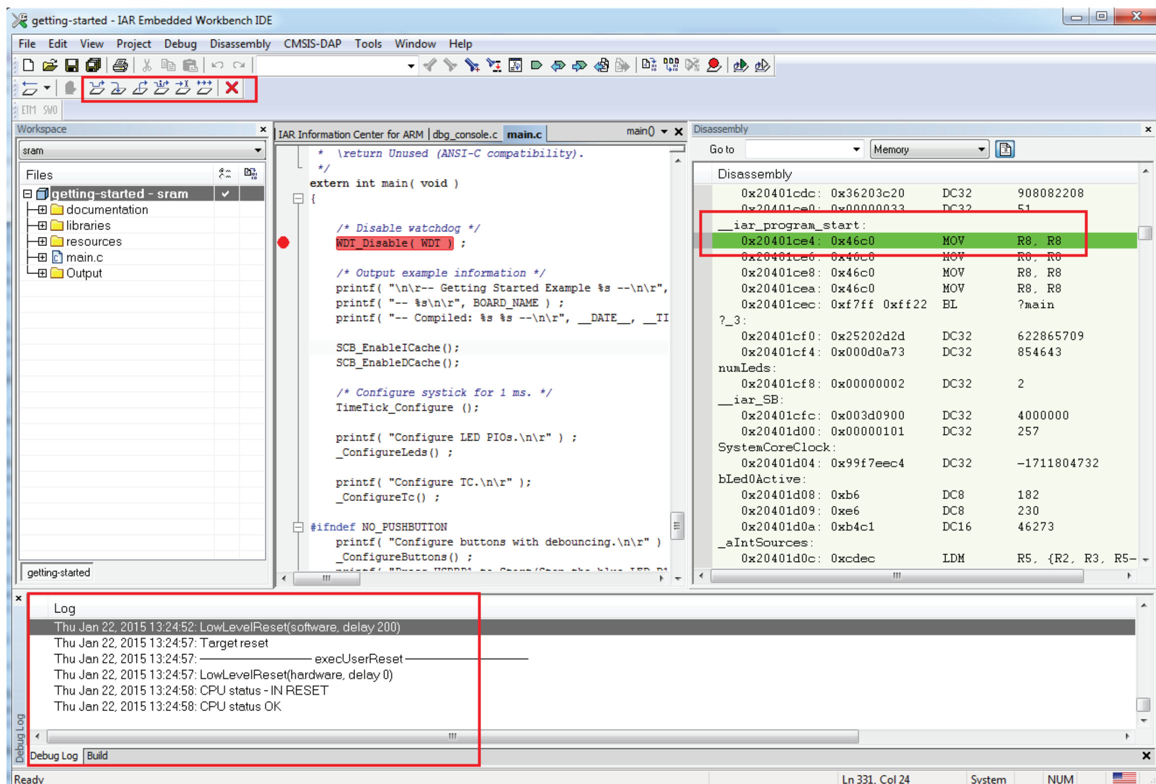
\..\getting-started\build\ewarm\sram\bin\.



3.2.2.2 Downloading the Code

After generating binary files, run the project by clicking the **Download and Debug** button .

After the code is downloaded to the corresponding memory (Flash/SRAM), the buttons on debug tool bar are available as shown below.



Then we can see LEDs blinking, which can be controlled by Serial Port input. Press ‘1’ to Start/Stop the LED0 blinking and press ‘2’ to Start/Stop the LED1 blinking.

3.3 MDK-ARM

Note that the Getting Started example has already been ported and included in Keil® MDK, but MDK-ARM V5.12 does not support SAM V71 devices so far. Refer to www.keil.com.

After installing MDK-ARM, if you find that the MDK installed on your PC does not support SAM V71 devices, you will need to install the packages *MDK5_Atmel_SAMx7_JLink_AddOn.exe* and *Keil.SAMx7_DFP.1.0.1.pack* to make sure that MDK supports SAM V71 devices. These are available from www.keil-compiler.de. More details are shown below.

3.3.1 Set up a Development and Debug Environment

3.3.1.1 Setting up Development Tools

- Install MDK-ARM IDE.
- Install two packages *Keil.SAMx7_DFP.1.0.1.pack* and *MDK5_Atmel_SAMx7_JLink_AddOn.exe* to make sure MDK supports SAM V71 devices.
- Install the software package by running *SAMV71_softpack_x.x_for_mdk_5.x.exe*.

Now, the user can generate binary corresponding to SAM V71 devices via MDK.

3.3.1.2 Setting up Necessary Tools

In this section, choose SAM-ICE (J-Link), ULINK2 or ULINK*pro* as the debugger.

When downloading and debugging via SAM-ICE (J-Link), *SEGGER J-Link software & documentation pack* needs to be installed on PC.

ULINK2 is configured as a Human Interface Device (HID) and is, therefore, directly supported by Windows operating systems. Thus no specific drivers are required when using ULINK2 as the debugger.

ULINK*pro* uses a specific Keil USB driver, which is part of the Keil tools. When connecting ULINK*pro* for the first time to the PC, Windows detects the new hardware. Follow the instructions to install the driver.

3.3.2 Generating and Downloading Binary File

3.3.2.1 Building the Project

Before building the project, it is necessary to check the project configuration.

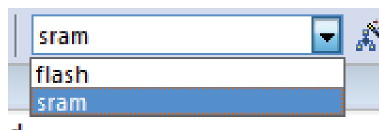
Follow the steps below:

1. Open the workspace “getting-started.uvprojx” in directory:

\..*examples\Atmel\samv7-xPlained\examples\getting-started\build\mdk*\.

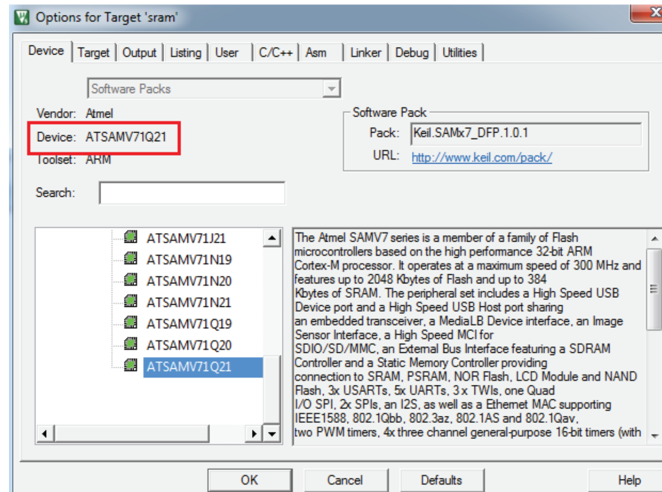
2. Select Flash or SRAM.

Two modes (flash & sram) can be selected in the drop-down menu as follows.

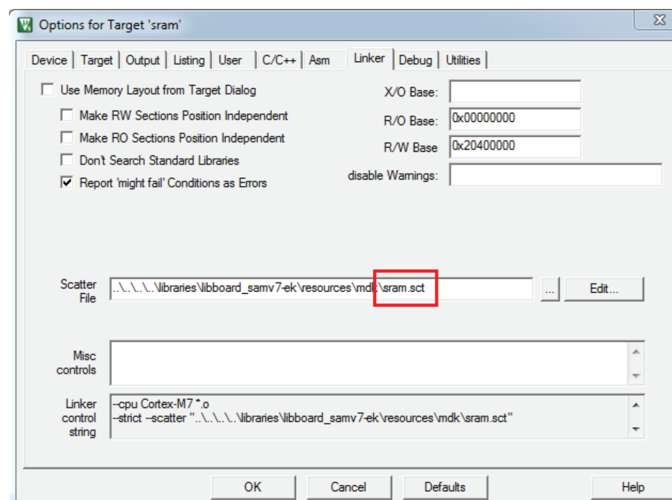


3. Check the configurations. The key default settings are shown as follows:

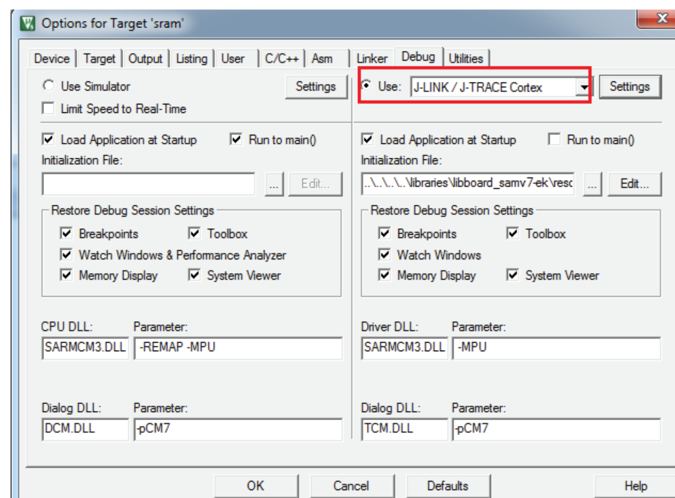
- “Device”: select “Atmel SAMV71Q21” as Device.

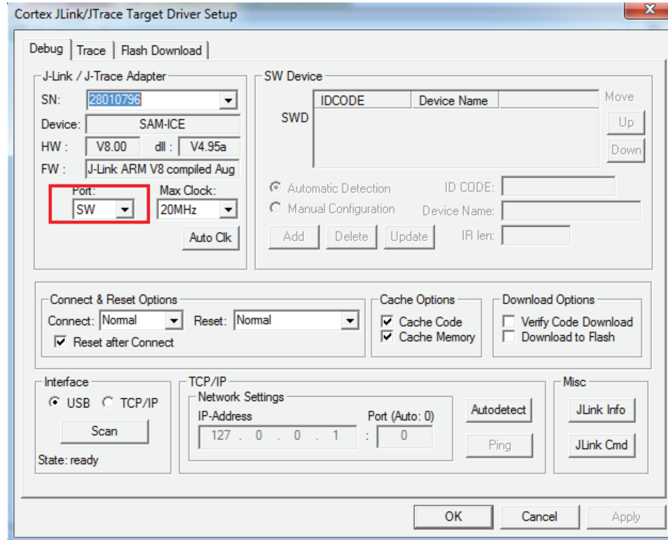


- “Linker”: Linker configuration file must be “flash.sct” (when in Flash mode) or “sram.sct” (when in SRAM mode).

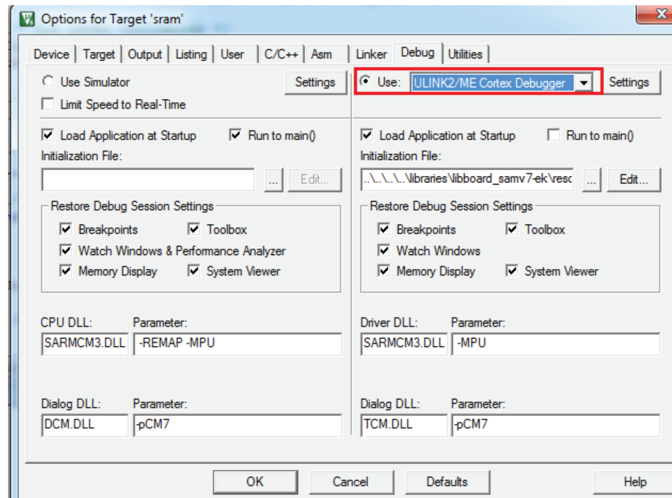


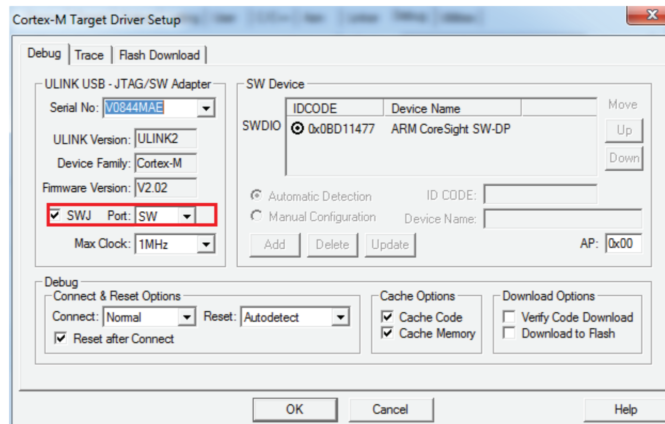
- ”Debug”: SAM-ICE, ULINK2 or ULINK*pro* can be selected as the debugger to download the binary file generated by MDK. The corresponding configurations are shown below respectively.
  – Using SAM-ICE (J-Link) as the debugger





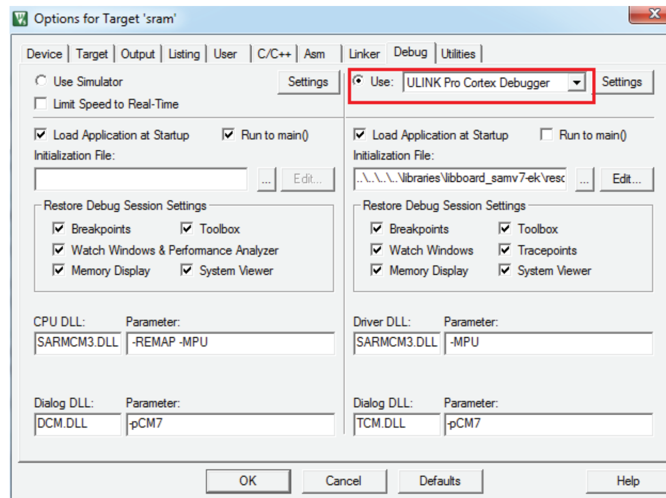
– Using ULINK2 as the debugger.

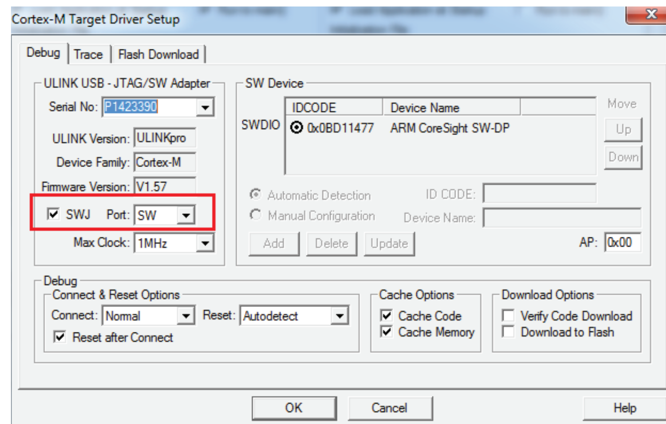




Atmel

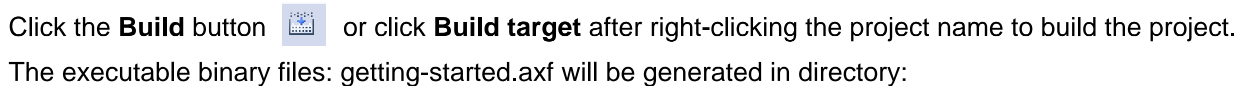
– Using ULINK*pro* as the debugger.





4. Build the Project

Click the **Build** button or click **Build target** after right-clicking the project name to build the project. The executable binary files: getting-started.axf will be generated in directory: *\..\examples\getting-started\build\mdk\Objects*.

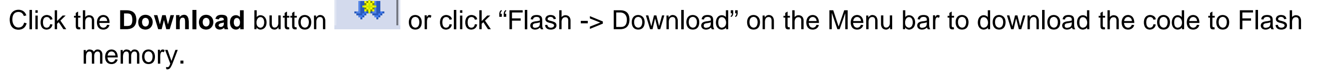


Build Output

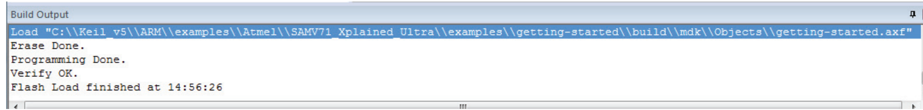
```
compiling board_lowlevel.c...
compiling board_memories.c...
compiling dbg_console.c...
compiling led.c...
compiling trace.c...
linking...
Program Size: Code=6416 RO-data=552 RW-data=44 ZI-data=12344
".\Objects\getting-started.axf" - 0 Error(s), 0 Warning(s).
```

3.3.2.2 Downloading the Code

- Flash Programming



Click the **Download** button or click "Flash -> Download" on the Menu bar to download the code to Flash memory.

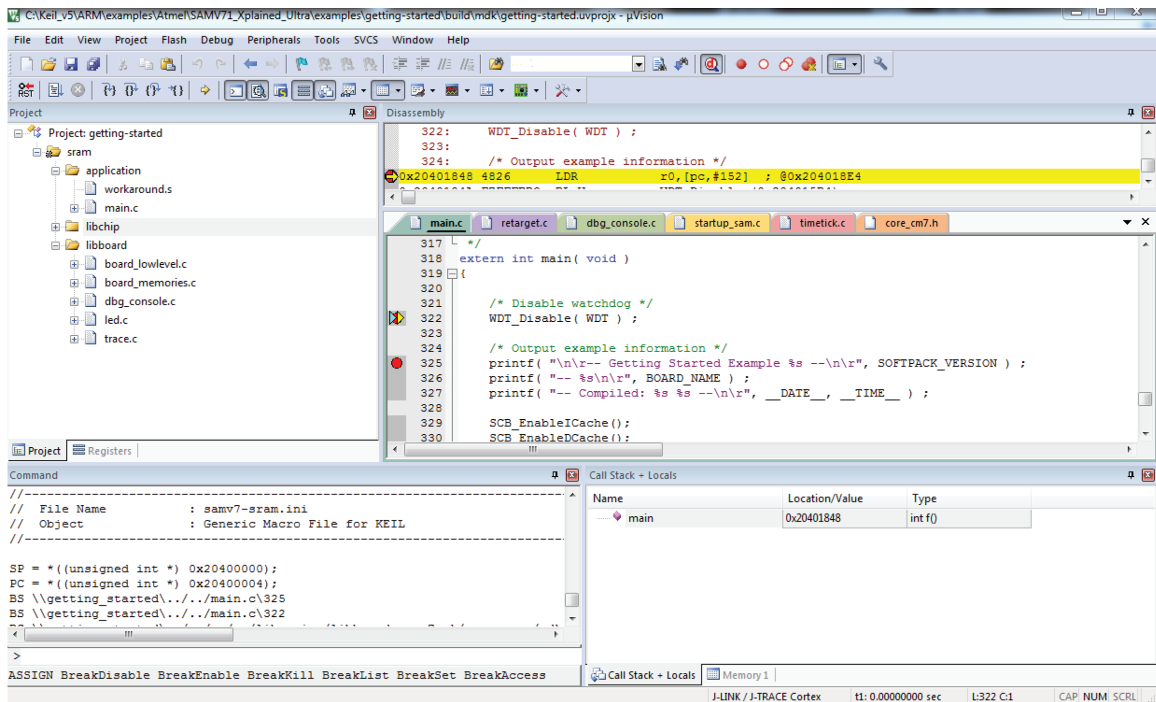


Then you can reset the board to run the program.

- Debugging in SRAM

SRAM mode is only for debugging. After generating executable files, download and debug the project by clicking the **Debug** button .

The IDE window is displayed as shown below:



4. Revision History

Table 4-1. Getting Started with SAM V71 Microcontrollers Application Note – Revision History

Doc. Rev. 44031A	Changes
08-Apr-16	Section 1. "Requirements": modified reference to SAM-BA document. Section 2.3.1.3 "Low-Level Initialization: SystemInit": Main RC oscillator default frequency corrected to 12 MHz. Section 2.3.1.5 "Low-Level Initialization: Tightly Coupled Memory (TCM)": corrected TCM states at startup.
16-Mar-15	First issue

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**

© 2016 Atmel Corporation. / Rev.: Atmel-44031B-ATARM-Getting-Started-with-SAM-V71-Microcontrollers-ApplicationNote_18-Apr-16.

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo, and others are the registered trademarks or trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death (“Safety-Critical Applications”) without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.