First published by Doulos March 2009

Contents

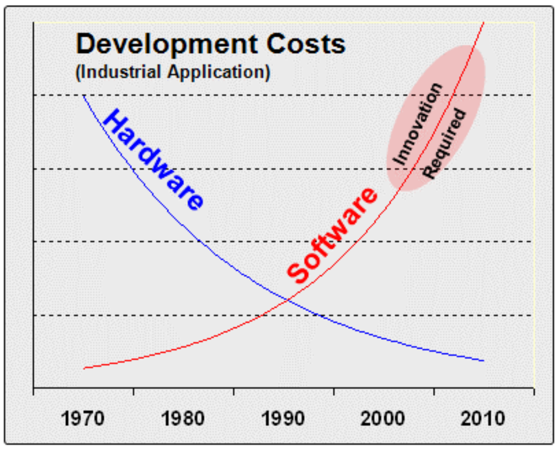
CMSIS

Introduction

The **Cortex Microcontroller Software Interface Standard (CMSIS)** supports developers and vendors in creating reusable software components for ARM Cortex-M based systems.

The ARM Cortex-M3 processor is the first core from ARM specifically designed for the Microcontroller market. This core includes many common features (NVIC, Timer, Debug-hardware) needed for this market. This will enable developers to port and reuse software (e.g. a real time kernel) with much less effort to Cortex-M3 based MCUs.

With a significant amount of hardware components being identical, a large portion of the Hardware Abstraction Layer (HAL) can be identical. However, reality has shown that lacking a common standard we find a variety of HAL/driver libraries for different devices, which, as far as the Cortex-M3 part is concerned essentially do the same thing – just differently.



The latest study of the development for the embedded market shows that software complexity and cost will increase over time, see figure left. Reusing Software and having a common standard to govern how to write and debug the software will be essential to minimising costs for future developments.

With more Cortex-M3 based MCUs about to come onto the market, ARM has recognized that after solving the diversity issue on the hardware side, there is still a need to create a standard to access these hardware components.

Figure 1 Development Costs

The result of that effort is CMSIS; a framework to be extended by vendors, while taking advantage of a common API (Application Programming Interface) for core specific components and conventions that define how the device specific portions should be implemented to make developers feel right at home when they reuse code or develop new code for ARM Cortex-M based devices.

CMSIS Structure

CMSIS can be divided into three basic function layers:

- Core Peripheral Access Layer (CPAL)
- Middleware Access Layer (MWAL)
- Device Peripheral Access Layer (DPAL)

The basic structure and the functional flow is illustrated in the Figure 2. below.

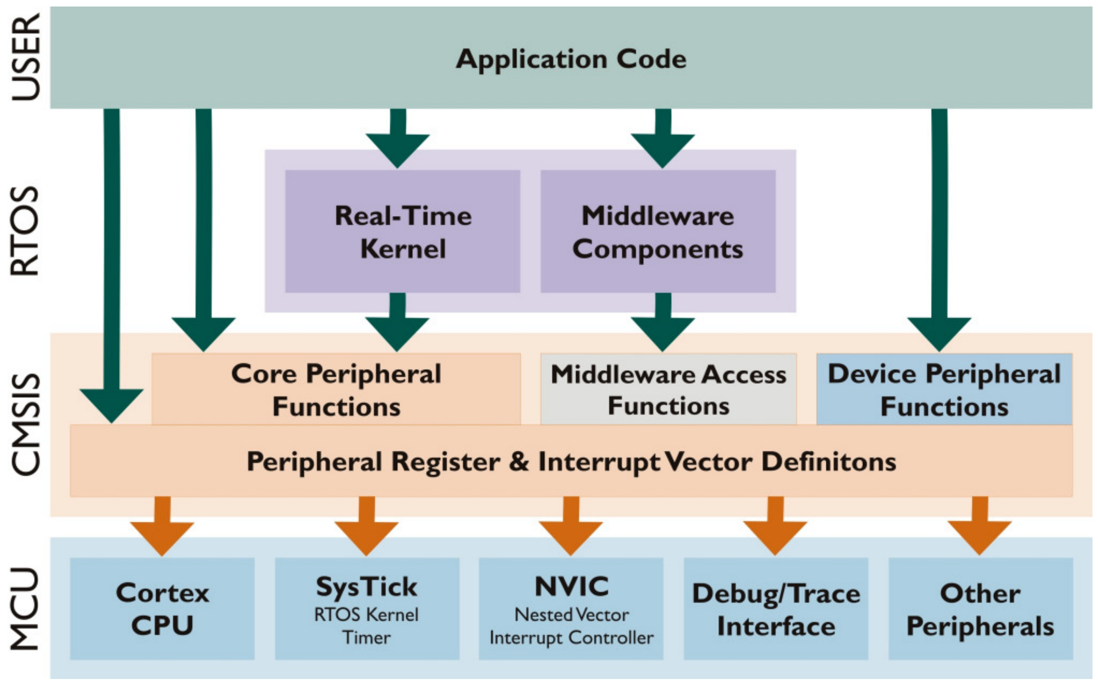

USER
Application Code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Register & Interrupt Vector Definitons
MCU
Cortex CPU
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals

Figure 2 CMSIS Structure functional flow

Core Peripheral Access Layer (CPAL)

The lowest level defines addresses, and access methods for common components and functionality that exists in every Cortex-M system. Access to core registers, NVIC, debug subsystem is provided by this layer. Tool specific access to special purpose registers (e.g. CONTROL, xPSR), will be provided in the form of inline functions or compiler intrinsics. This layer will be provided by ARM.

Middleware Access Layer (MWAL)

This layer is also defined by ARM, but will be adapted by silicon vendors for their respective devices. The Middleware Access Layer defines a common API for accessing peripherals. The Middleware Access Layer is still under development and no further information is available at this point.

Device Peripheral Access Layer (DPAL)

Hardware register addresses and other definitions, as well as device specific access functions will be defined in this layer. The Device Peripheral Access Layer is very similar to the Core Peripheral Access Layer and will be provided by the silicon vendor. Access methods provided by CPAL may be referenced and the vector table will be adapted to include device specific exception handler address.

While DPAL is intended to be extended by the silicon vendor, let's not forget about Cortex-M based FPGA products, which effectively put developers into the position of a silicon vendor.

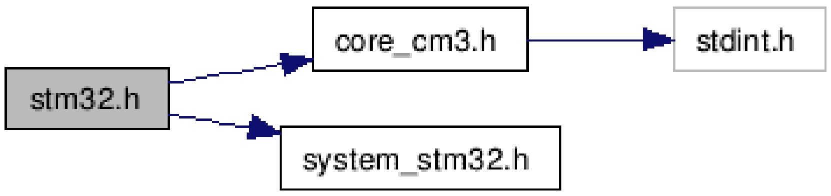
As far as MCU based systems are concerned it might make sense for developers to treat the entire PCB system as monolithic block. There is no reason to differentiate between a memory mapped register inside the MCU and a memory mapped register external to the MCU, connected via external memory interface. The benefit of applying a standard like CMSIS is that existing guidelines on how to access these devices set a clear goal on how to implement and integrate critical parts of the software. Other team members will find a familiar environment.

File Structure

File names in CMSIS are standardized as follows:

core_cm3.h	Cortex-M3 global declarations and definitions, static function definitions
core_cm3.c	Cortex-M3 global definitions
<device>*.h*	Top-level header file (device specific). To be included by application code. Includes *core_cm3.h* and *system_*<device>*.h*
system_*<device>.h*	Device specific declarations
system_*<device>.c*	Device specific definitions, e.g. SystemInit()

Application code will only include the top-level header file which implicitly pulls in all other essential



header files. The illustration below shows the flow and dependencies of the header files *stm32.h*, *core_cm3.h* and *system_stm32.h*, which are part of CMSIS release V1P0.

```
/* Configuration of the Cortex-M3 Processor and Core Peripherals */
#define __MPU_PRESENT          0 /*!< STM32 does not provide a MPU present or not*/
#define __NVIC_PRIO_BITS       4 /*!< STM32 uses 4 Bits for the Priority Levels */
#define __Vendor_SysTickConfig 0 /*!< Set to 1 if different SysTick Config
                                      is used */



#include "core_cm3.h"            /* Cortex-M3 processor and core peripherals     */
#include "system_stm32.h"        /* STM32 System                                 */
```

The <device>*.h* file is the central include file and provided by the silicon vendor. The application programmer is using that as the main include file in his C source code. Note that the ARM Cortex-M3 has some optional hardware features (e.g. the MPU, number of Interrupts and the number of the NVIC priority bits) the silicon vendors may have implemented differently. The listing above shows that STM32 implements four out of eight possible priority bits. The macro “__NVIC_PRIO_BITS” is set here to 4. STM32 does not offer a Memory Protection Unit (MPU). Accordingly, the macro “__MPU_PRESENT” has the value 0.

The next example shows the corresponding definitions for a NXP LPC17xx device. In this Cortex-M3 implementation five priority bits have been implemented and an MPU is available.

```
/* Configuration of the Cortex-M3 Processor and Core Peripherals */
#define __MPU_PRESENT          1 /*!< MPU present or not */
#define __NVIC_PRIO_BITS       5 /*!< Number of Bits used for Priority Levels */
#define __Vendor_SysTickConfig 0 /*!< Set to 1 if different SysTick Config
                                      is used */


#include "..\core_cm3.h"        /* Cortex-M3 processor and core peripherals  */
#include "system_LPC17xx.h"     /* System Header                             */
```

The “__Vendor_SysTickConfig” defined is showing in both cases the default setting. When this macro is set to 1, the SysTickConfig() function in the *cm3_core.h* is excluded. In this case the file <device>*.h* must contain a vendor specific implementation of this function.

Tool Independence

CMSIS exists in a three-dimensional space of the form vendor–device–tool chain. In order to remove one dimension (tool chain), the common files *core_cm3.c* and *core_cm3.h* contain all essential tool specific declarations and definitions.

Example:

```
/* define compiler specific symbols */
#if defined ( __CC_ARM   )
  #define __ASM            __asm          /*!< asm keyword for armcc         */
  #define __INLINE         __inline       /*!< inline keyword for armcc       */

#elif defined ( __ICCARM__ )
  #define __ASM           __asm           /*!< asm keyword for iarcc          */
  #define __INLINE        inline          /*!< inline keyword for iarcc. Only
avaiable in High optimization mode! */
  #define __nop           __no_operation  /*!< no operation intrinsic in iarcc */

#elif defined   (  __GNUC__  )
  #define __ASM             asm           /*!< asm keyword for gcc           */
  #define __INLINE          inline        /*!< inline keyword for gcc        */
#endif
```

The remaining parts of CMSIS can now simply use the macro __INLINE to define an inline function.

Currently three of the most important C-compilers are supported: ARM RealView (armcc), IAR EWARM (iccarm), and GNU Compiler Collection (gcc). This is expected to cover the majority of tool chains.

MISRA-C

Besides defining an API for Cortex-M core peripherals and guidelines on how to support device peripherals, CMSIS defines some coding guidelines and conventions. Most important is that the CMSIS code base is MISRA-C 2004 compliant, which implies that every extension should be compliant, too. MISRA-C is a set of safety rules established by the “Motor Industry Software Reliability Association” for the C programming language. Maintaining MISRA compliance can be tricky, in particular when implementing driver level software. Therefore, pragma-like exceptions in PCLint style are scattered across the source code. Be aware that other tools, e.g. MISRA checker in IAR EWARM, might flag errors. Each exception is accompanied with a comment explaining why this exception was made.

CPAL Functions

All functions in the Core Peripheral Access Layer are reentrant and can be called from different interrupt service routines (ISR). CPAL functions are also non-blocking[1] in the sense that they do not contain any wait-loops.

The majority of functions in the CPAL have been implemented in the header file *core_cm3.h* as *static inline* functions. This allows the compiler to optimize the function calls by placing the instructions that make up the called function along with other code from which the function was called.

Interrupt Service Routines

Exception handlers will get a name suffix "_Handler", while (external) interrupt handlers get the suffix "_IRQHandler". There must be a default handler for each interrupt, which executes an infinite loop. Tool specific configuration must make sure that this default handler will be used as fall-back if no user-provided handler exists[2].

Given that the Cortex-M NVIC provides byte-arrays and bit-strings to configure priorities and interrupt source en-/disable, an enumerated type IRQn _t with an element for each exception/interrupt position with the suffix "_IRQn" must be defined for each interrupt (<device>*.h*). The system handler names are common for all devices and must not be changed.

```
typedef enum IRQn
{
/******  Cortex-M3 Processor Exceptions Numbers ***********************************/
  NonMaskableInt_IRQn          = -14, /*!< 2 Non Maskable Interrupt              */
  MemoryManagement_IRQn        = -12, /*!< 4 Cortex-M3 Memory Mgmt Interrupt    */
  BusFault_IRQn                = -11, /*!< 5 Cortex-M3 Bus Fault Interrupt      */
  UsageFault_IRQn              = -10, /*!< 6 Cortex-M3 Usage Fault Interrupt    */
  SVCall_IRQn                  = -5,  /*!< 11 Cortex-M3 SV Call Interrupt       */
  DebugMonitor_IRQn            = -4,  /*!< 12 Cortex-M3 Debug Monitor Interrupt */
  PendSV_IRQn                  = -2,  /*!< 14 Cortex-M3 Pend SV Interrupt       */
  SysTick_IRQn                 = -1,  /*!< 15 Cortex-M3 System Tick Interrupt   */

/******  Device specific Interrupt Numbers ****************************************/
  UART_IRQn                    = 0,   /*!< Example Interrupt */
} IRQn_Type;
```

Listing shows the generic part of the (<device>*.h) file.*

All system handlers have negative virtual slot numbers so that they can be distinguished in functions that abstract from the differences between system handlers and external interrupt handlers. External interrupt handlers start at the index 0.

Other Coding Conventions

The CMSIS documentation recommends a few more things regarding capitalization of identifiers, commenting code.

[1] Memory barriers are exempt from that rule although they might stall the processor for a few cycles.

[2] Through __weak declaration in EWARM and RVCT armcc, __attribute__((weak)) in GCC and RVCT armcc and [WEAK] export in RVCT/armasm.

Identifiers

- Capital names to identify Core Registers, Peripheral Registers, and CPU Instructions.

  E.g.: NVIC->AIRCR, GPIOB, LDMIAEQ

- "CamelCase" (mix of upper- and lower-case letters) names to identify peripherals access functions and interrupts.

  E.g.: SysTickConfig(),DebugMonitor_IRQn

- Peripheral prefix (<*name*>_) to identify functions that belong to specific peripherals.

  E.g.: ITM_SendChar(),NVIC_SystemReset()

Comments

CMSIS uses Doxygen style comments for all definitions and encourages developers to do the same.

In particular, the comment for each function definition should at least contain

- one-line brief function overview. (Tag: @brief)
- detailed parameter explanation. (Tag: @param)
- detailed information about return values. (Tag: @return)
- detailed description of the actual function.

The example below shows the beginning of a function definition:

```
/**
 * @brief  Enable Interrupt in NVIC Interrupt Controller
 *
 * @param  IRQn_Type IRQn specifies the interrupt number
 * @return none
 *
 * Enable a device specific interupt in the NVIC interrupt controller.
 * The interrupt number cannot be a negative value.
 */
static __INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
  …
}
```

The tags can be parsed by the documentation tool Doxygen, which is used to create cross-referenced source code documentation in various formats (http://www.stack.nl/~dimitri/doxygen/index.html). The tag syntax is rather minimalistic and does not impair readability of the source code. Please consult the Doxygen Documentation for details about tag syntax.

As an alternative to regular C block comments (/* */) CMSIS explicitly allows line comments (//, so called C++-comments). If you are concerned about MISRA compliance, be aware though, that MISRA-C 2004 doesn't allow line comments according to rule 2.2.

Data Types

All data types referenced by CMSIS are based on those defined in the standard C header file *stdint.h*. Data structures for core registers are defined CMSIS header file *core_cm3.h*, along with macros for qualifying registers according to their access permissions. The rationale is that tools might be able to automatically extract that information for debug purposes.

```
#define     __I     volatile const      /*!< defines 'read only' permissions    */
#define     __O     volatile            /*!< defines 'write only' permissions   */
#define     __IO    volatile            /*!< defines 'read / write' permissions */
```

Debugging

A common requirement in software development is some sort of terminal output for debugging. Text/graphics displays in embedded devices cannot be assumed to be at hand (or might be in use), which previously left the developer with essentially two choices:

1. Use one of the ubiquitous UARTs and connect a terminal

   Issues: All UARTs might be in use, access to UART signals might not be possible for reasons that include pin-sharing, PCB layout, etc.

2. Use the semihosting mechanism

   Issue: Significant software overhead on target CPU, might not be supported in the same way by all tool chains, potential impact on timing behavior.

With Cortex-M3 the preferred method makes use of the Instrumentation Trace Macrocell (ITM), which is part of the processor macro cell and thus always present.[3] A Serial Wire Viewer (SWV) capable debug adapter can receive ITM data through the SWO (Serial Wire Out) debug pin. ITM implements 32 general purpose data channels. CMSIS builds on top of this and declares channel 0 to be used for terminal output, along with a function called ITM_SendChar() which can be used as low-level driver function for printf-style output. A second channel (31) has been reserved for OS aware debugging, which means that a kernel can use it to transmit kernel specific data which could then be interpreted by the debug tool. With this standardization, tool vendors have it much easier to implement specific debug features, such as e.g. terminal emulation for data received via ITM channel 0. Developers on the other hand can rely on this feature to dump state information, without having to configure UARTs and external terminal emulators. See our Tutorial 2 later in this document.

Access privilege can be configured for groups of ITM channels. In order to use ITM channel 0, unprivileged access must be granted, whereas ITM channel 31 is in a different group and may allow privileged access only.

Future Updates

The CMSIS developers have taken care to provide macros indicating the CMSIS version used in a project. That way provisions can be made to prevent code to be used with a different CMSIS version than originally intended.

```
#define __CM3_CMSIS_VERSION_MAIN  (0x01)      /* [31:16] main version      */

#define __CM3_CMSIS_VERSION_SUB   (0x10)      /* [15:0]  sub version       */

#define __CM3_CMSIS_VERSION       ((__CM3_CMSIS_VERSION_MAIN << 16) | \

                                   __CM3_CMSIS_VERSION_SUB)
```

[3] Always present in Cortex-M3 rev1 cores. Cortex-M3 rev2 makes ITM an optional feature.

Tutorial 1 – A First Example

In order to explain application of CMSIS in real projects, we are going to look at a simple example of a Cortex-M3 application. The program compiles for STM32 processors and a project file for the Keil µVision IDE has been provided. Porting the example to other tool chains, such as IAR EWARM is straight forward and the IAR EWARM version is provided as well. A great number of CMSIS function definitions can be found in *core_cm3.h* as "static inline" functions. Depending on the compiler optimization level, this helps getting very efficient instruction sequences rather than actual function calls, while ensuring a certain level of type safety.

After clock and GPIO initialization, the SysTick timer is configured to a period of 0.5 seconds. Whenever the handler executes it toggles the state of GPIOB[15].

Example 1 Starting point

Initially, the program was implemented using STMicroelectronics' FWLib, a library that provides access to Cortex-M3 internals and STM32 peripherals. Near to medium term, firmware libraries such as FWLib will be based on CMSIS. Parts of FWLib that will eventually form the DPAL (see above).

```
#include <stdint.h>

#include <stm32f10x_lib.h>

GPIO_InitTypeDef GPIOB_InitStruct = {
    .GPIO_Pin   = GPIO_Pin_All,
    .GPIO_Speed = GPIO_Speed_2MHz,
    .GPIO_Mode  = GPIO_Mode_Out_PP
};

int main()
{
    ErrorStatus       HSEStartUpStatus;
    RCC_ClocksTypeDef Clocks;

    /*
     * Clock initialization
     */
    RCC_HSEConfig(RCC_HSE_ON);
    HSEStartUpStatus = RCC_WaitForHSEStartUp();

    if (HSEStartUpStatus != SUCCESS) {
        while(1);
    }

    RCC_SYSCLKConfig(RCC_SYSCLKSource_HSE);
    RCC_HCLKConfig(RCC_SYSCLK_Div1);
    RCC_PCLK2Config(RCC_HCLK_Div1);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    /*
     * NVIC initialization
     */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_3);
    NVIC_SystemHandlerPriorityConfig(SystemHandler_SysTick, 7, 0);

    /*
     * GPIOB initialization
     */
    GPIO_Init(GPIOB, &GPIOB_InitStruct);
    GPIO_WriteBit(GPIOB, GPIO_Pin_All, Bit_RESET);

    /*
     * SysTick initialization
     */
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);

    RCC_GetClocksFreq(&Clocks);
    SysTick_SetReload((Clocks.HCLK_Frequency)/2);
    SysTick_ITConfig(ENABLE);
    SysTick_CounterCmd(SysTick_Counter_Enable);

    while(1);
}
```

```
void SysTickHandler(void)
{
    static BitAction toggle = Bit_SET;

    GPIO_WriteBit(GPIOB, GPIO_Pin_15, toggle);
    if (toggle == Bit_SET) {
        toggle = Bit_RESET;
    }
    else {
        toggle = Bit_SET;
    }
}
```

The two listings above show the contents of *main.c*, and stm32f10x_it.c. This latter file contains interrupt handler templates from ST's FWLib, which have to be adapted to implement project specific functionality.

Example 2 Converting to CMSIS

In a second step, the program has been converted to using CMSIS. The CMSIS version used is V1P10 as downloaded via the link above. We want to make sure to use the same CMSIS that has been used to develop the program and check the version number.

```
#include <stdint.h>

#include <stm32.h>                      // *** CMSIS change ***

#if (__CM3_CMSIS_VERSION != 0x00010010)
#    error "__CM3_CMSIS_VERSION: Unexpected CMSIS version detected"
#endif
```

Initial support for STM32 MCU is part of CMSIS and is pulled in by including the header file *stm32.h*. At the point of writing this tutorial a fully CMSIS complaint FWLib was not available so some compromises and hand adjustments hand to be made. Fore this reason, we will include both, FWLib and CMSIS files. Until vendors have full adopted CMSIS small issues will have to be dealt with when combining CMSIS with a vendor library.

In this case simply including the FWLib main header file *stm32f10x_lib.h* in addition to *stm32.h* triggers a number of error messages caused by multiple definitions of functions and macros. To avoid this, we will have to selectively include individual FWLib headers (see below). All FWLib headers depend on definitions in the files *cortexm3_core.h* and *stm32f10x_map.h*. Most of the definitions in these two header files have already been defined by CMSIS (*core_cm3.h* and *system_stm32.h*) and we have to pretend to FWLib that both header files had been included already.

```
// Prevent interference with FWLib
#define __STM32F10x_MAP_H
#include <stm32f10x_type.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_rcc.h>
```

Actual system initialization will be encapsulated by the CMSIS function SystemInit(), which has to be implemented by the silicon vendor. As a minimal requirement, this function would initialize the MCU's clock system. In case of the reference implementation in *system_stm32.c*, SystemInit() also initializes the Flash memory interface. CMSIS defines a single system variable, SystemFrequency, which is supposed to reflect the frequency of both core and SysTick timer in Hz. This concept is sufficient for a minimal implementation but will likely have to be extended for actual MCU as demonstrated by CMSIS' *system_stm32.c*, in which several variables have been defined to hold the frequency values of different clock domains in the STM32 MCU. SysTick timer and core could have different frequencies and care must be taken when using SystemFrequency in a program.

Current CMSIS does not initialize peripheral clocks and it is arguable whether it should. In any case we use the corresponding FWLib function to enable GPIOB clock.

```
    // Initialization moved to SystemInit() in system_stm32.c. Clock
    // configuration now handled by #defines. Use uVision
    // configuration wizard or text editor to change.
    SystemInit();                   // *** CMSIS change ***

    // APB peripherals still have to be enabled individually.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
```

NVIC group- and sub-priority configuration is handled by the function NVIC_SetPriorityGrouping(), where the direct encoding of the PRIGROUP field in SCB->AIRCR is used. We choose the value 4, which represents 3 bits for group (preempting) and 5 bits for the sub priority. A general formula to calculate the proper value is PRIGROUP = bitssub-1.

```
    // priority configuration: 3.5

    NVIC_SetPriorityGrouping(4);        // *** CMSIS change ***
```

Different from the initial version of the example, the CMSIS variant does not at this point set the SysTick handler priority. This is part of the SysTick initialization and will be covered later.

```
    GPIO_Init(GPIOB, &GPIOB_InitStruct);

    GPIO_WriteBit(GPIOB, GPIO_Pin_All, Bit_RESET);
```

Following NVIC set-up, we use plain FWLib functions to configure GPIO port B.

```
    SysTick_Config(SystemFrequency/2);  // *** CMSIS Change ***


    // SysTick_Config() hardcodes priority. We will overwrite this.

    NVIC_SetPriority(SysTick_IRQn, 14); // *** CMSIS change ***
```

SysTick_Config(), provided by CMSIS, programs the reload register with the parameter value. The function also selects HCLK (core clock) as clock source, enables SysTick interrupts and starts the counter. The function also fixes the SysTick handler priority to the lowest priority in the system, which is the recommended SysTick priority for use in an RTOS scheduler for instance. In our example, however, we prefer a different priority and override the hard coded value with an additional call to NVIC_SetPriority(). This function abstracts from the difference between Cortex-M3 system handlers and external interrupt handlers. All configurable system exceptions will be identified by negative IRQ numbers (see above).

```
__irq void SysTick_Handler()
{
    static BitAction toggle = Bit_SET;
    GPIO_WriteBit(GPIOB, GPIO_Pin_15, toggle);
    if (toggle == Bit_SET) {
        puts("Pin state is ON");
        toggle = Bit_RESET;
    }
    else {
        puts("Pin state is OFF");
        toggle = Bit_SET;
    }
}
```

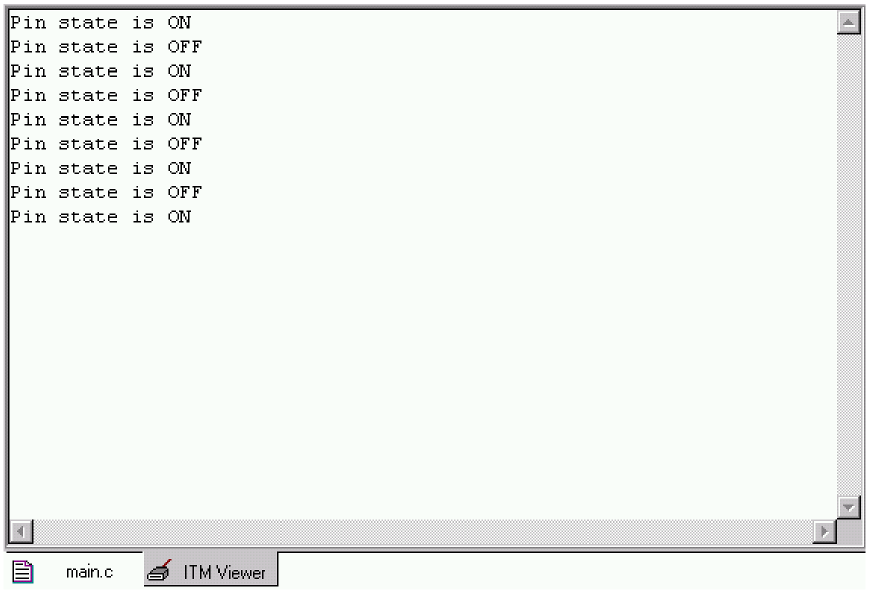
The SysTick handler code above does not need any modification. FWLib naming conventions is complying with CMSIS, in that the names of all internal exception handlers must end in "_Handler". Names of external interrupt handlers must end in "_IRQHandler". The handler implementation accesses the GPIO port via FWLib functions and definitions.

Tutorial 2 – ITM Debug

To exercise some CMSIS debug functionality for our first example from tutorial 1, we add debug output messages via calls to puts(). We will now redirect character output to Instrumentation Trace Macrocell (ITM), (remember that CMSIS reserves ITM channel 0 for this), using the CMSIS function ITM_SendChar(). A mechanism called retargeting enables us to provide our own implementation of a system function.

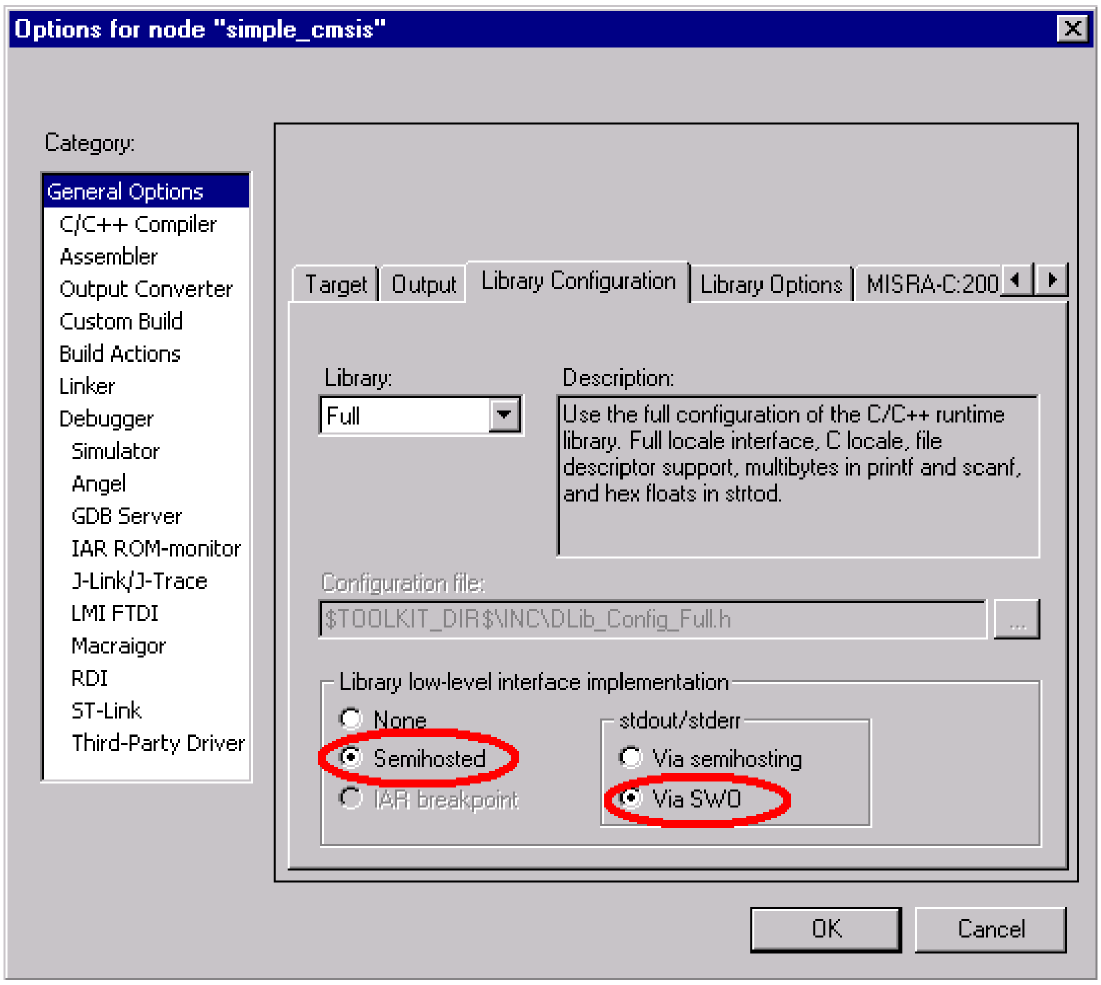
```
// retarget fputc() for debug output via ITM

int fputc(int c, FILE *stream)

{

    return (int)ITM_SendChar((uint32_t)c);

}
```

The standard C function fputc(), which will eventually be called by puts() in our SysTick handler, will be re-implemented, taking advantage of the function ITM_SendChar(). The result of this retarget can now be easily monitored in µVision ITM viewer as shown in the screenshot below.



```
Pin state is ON
Pin state is OFF
Pin state is ON
Pin state is OFF
Pin state is ON
Pin state is OFF
Pin state is ON
Pin state is OFF
Pin state is ON
```

If you are going to use the IAR EWARM tool for this example it is not necessary to manually retarget this function. Instead, in the project options dialog under “General Options” there is a tab “Library Configuration” tab, which offers checkboxes to enable this functionality. The screenshot below shows which settings are required to redirect standard output.


Options for node "simple_cmsis"

Category:

- General Options
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
  - Simulator
  - Angel
  - GDB Server
  - IAR ROM-monitor
  - J-Link/J-Trace
  - LMI FTDI
  - Macraigor
  - RDI
  - ST-Link
  - Third-Party Driver

Tabs: Target | Output | Library Configuration | Library Options | MISRA-C:200

Library: Full

Description: Use the full configuration of the C/C++ runtime library. Full locale interface, C locale, file descriptor support, multibytes in printf and scanf, and hex floats in strtod.

Configuration file: $TOOLKIT_DIR$\INC\DLib_Config_Full.h

Library low-level interface implementation: None [✓ Semihosted] IAR breakpoint

stdout/stderr: Via semihosting [✓ Via SWO]

Summary

The CMSIS will reduce the learning-curve for the application programmers by providing a consistent software framework, ensuring consistent documentation and easy deployment of boilerplate code across various compiler vendors. The consequent use and implementation of CMSIS across many silicon and middleware software partners will simplify the verification and certification process and therefore reduce future project risk. The adapted common programming techniques though CMSIS will simplify the long term maintenance due to easier to understand source code. The silicon vendors can focus on there added value and device features. All reasons together will reduce software development cost and time to get new products to the market.

RapidGain^{TM} - Designing with ARM Cortex-M3 Based Microcontrollers

RapidGain^{TM} - Designing with ARM Cortex-M3 Based Microcontrollers is unique in delivering a complete overview of the functional scope of the ARM® Cortex^{TM}-M3 processor core for popular microcontrollers (MCUs), as well as hands-on experience with the RealView Microcontroller Development Kit (MDK) – all in a single day. It includes an independent comparison of available Cortex-M3 based MCUs, which will support delegates in making the right choice for their projects.

A tightly focused and practical training event, this class enables new and prospective users to rapidly gain the experience and understanding they need to get started with ARM Cortex-M-based microcontrollers, and achieve significant initial productivity gains. You will:

- Explore the essential elements of the ARM Cortex-M3 processor core
- Gain an *independent* overview of the ARM Cortex-M3-based MCUs currently available
- Develop and simulate a design example through hands-on practice using a Microcontroller Development Kit
- Introduction to Cortex Microcontroller Software Interface Standard (CMSIS)

Who should attend?

- Developers who wish to gain practical experience in the use of ARM Cortex-M3
- Engineers who want to inform themselves about cutting-edge microcontroller technology
- Project managers who need to make informed decisions on future system platforms

Structure and Content

Introduction to ARM Cortex-M3

Structure • Programming Model • Thumb-2 Instruction Set • Interrupt Handling • Migration Path ARM7 to Cortex-M3 • Storage Architecture • Power Management • Debugging Components
Exercise: Operating the µVision RealView MDK IDE, experimenting with Cortex-M3 software in the simulator

Overview of ARM Cortex-M3-based MCUs

Description of various current or future ARM Cortex-M3-based microcontrollers
Exercise: Completing and executing software using an MCU simulator

Cortex is a registered trade mark of ARM Holdings plc.

Dates and further Information:

www.doulos.com/ARM

www.doulos.com/RapidGain