EXAMINING ARM'S CORTEX MICROCONTROLLER SOFTWARE INTERFACE STANDARD - CMSIS

Niall Cooling
Feabhas Ltd.
www.feabhas.com

Version 1.2
May 2010

ABSTRACT

ARM's introduction of their Cortex Microcontroller Software Interface Standard (CMSIS) provides a framework of reusable components for both software developers and vendors to utilize. Aspects such as power-up boot management and interrupt control have been abstracted and defined by ARM for their Cortex family of microcontroller cores. This whitepaper examines the CMSIS framework and discusses the pros and cons of such an approach.

CORTEX BACKGROUND

Before delving into CMSIS (Cortex Microcontroller Software Interface Standard), it is worth a very short overview of ARM, which will help put in context where CMSIS has come from. First and foremost we need to establish what business ARM is in. The majority of ARM's revenue comes from licensing the Intellectual Property (IP) of core processor designs. ARM themselves don't make silicon - they leave that to their licensees.

ARM have been incredibly successful with their "classic" processor cores

- The ARM11 and ARM9 are ARM's high-end embedded cores. These cores are suitable to run high end operating systems such as embedded Linux, Windows CE and Symbian. Over 5 Billion ARM9 processors have been shipped so far and are widely used by TI in their OMAP family.
- The ARM7 is ARM's current low-end core and is widely used across a range of applications, most successfully in many mobile phones. The ARM7 was first introduced in 1994, and is ARM's most successful core to date. Over 10 Billion+ ARM7's have been shipped to date.

Not content to sit on their laurels, ARM has designed a complete new set of processor architectures to replace the aging ARM7, ARM9 and ARM11 families. These new cores are all part of the Cortex Family, broken down as (1):

- Cortex-A (Application) Series
    - ARM11 replacement
- Cortex-R (Real-time) Series
    - ARM9 replacement
- Cortex-M (Microcontroller) Series
    - ARM7 replacement

Currently CMSIS only applies to Cortex-M series processor cores. The Cortex-M family currently has four variants (Figure 1): the M0, M1, M3 and M4. Central to the design of CMSIS is that all these processor designs share a common core architecture.

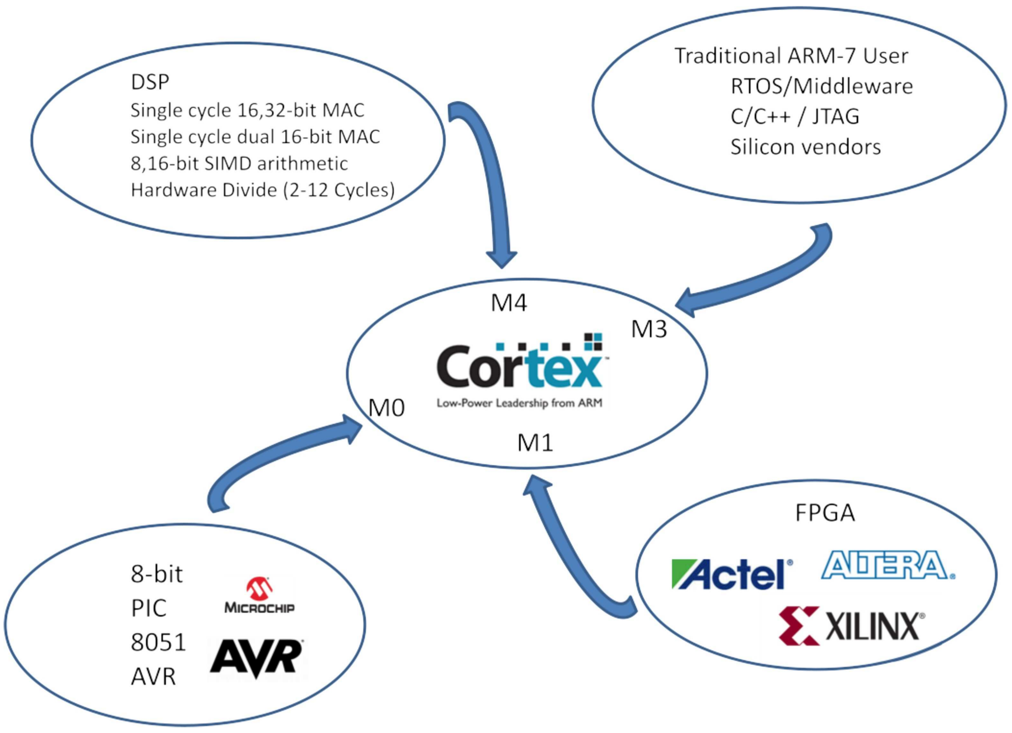

FIGURE 1 CORTEX-M FAMILY

The Cortex-M common features across the family are:

- RISC architecture
- Nested Vectored Interrupt Controller (*NVIC*)
- Common register set (general purpose registers, etc.)
- System Interrupt for tick timing (*SysTick*)

WHAT IS CMSIS?

What do ARM Say? According to ARM's website, CMSIS for a Cortex-M Microcontroller System (2) defines:

- A common way to access peripheral registers and a common way to define exception vectors.
- The register names of the Core Peripherals and the names of the Core Exception Vectors.
- A device independent interface for RTOS Kernels, including a debug channel.
- Interfaces for middleware components (TCP/IP Stack, Flash File System).

Simply put, CMSIS is a collection of source files (.c, .h and assembler) to create a minimal board support package (BSP) for Cortex-M series processors. In addition CMSIS attempts to define a Hardware Abstraction Layer (HAL) for common peripherals in addition to the Cortex-M core devices.

The whole project is heavily C Programming centric (C++ should work, but this may be implementation specific - this will be addressed later).

This paper is based on Version 1.30 of CMSIS. Version 2.0 is in development and due for release in Q3 of 2010. Version 2.0 has added support for the Cortex-M4.

KEY COMPONENTS OF CMSIS

CMSIS has three component parts:

- Core Peripheral Access Layer
- Instrumented Trace Macrocell (ITM)
- Middleware Access Layer

In addition CMSIS defines certain coding rules and conventions, for example, *CMSIS C code should conform to MISRA 2004 rules* (3). All code should use the standard data types defined in the ISO C99 header file `<stdint.h>` (4).

Other recommendations include using CamelCase (5) names to identify peripherals access functions and interrupts.

CORE PERIPHERAL ACCESS LAYER

The Core Peripheral Access Layer defines a number of assembly-implemented C routines for accessing the core registers and certain key instructions in the instruction set. As the NVIC is common across all Cortex-M processors (even the memory address the registers reside at are fixed) then CMSIS abstracts access to the NVIC through a set of functions. Finally all Cortex-M processors have a predefined system timer (SysTick) used to generate interrupts at a regular interval. CMSIS defines a function for configuring SysTick.

INSTRUMENTED TRACE MACROCELL (ITM)

An Instrumented Trace Macrocell (ITM) provides a new way for developers to output data to a debugger (it can do a number of other things but that is outside the scope of CMSIS). To provide a common mechanism, CMSIS defines a function for Cortex-M3 ITM Debug Access, called `ITM_SendChar`. It also defines additional debug access through `ITM_ReceiveChar`. This allows simple `printf/scanf` support for debugging.

MIDDLEWARE ACCESS LAYER

Additionally the Middleware Access Layer attempts to define standard interfaces to common devices external to the Cortex-M core, for example Ethernet Controllers, UARTs (Universal Asynchronous Receiver Transmitter) and SPI (Serial Peripheral Interface).

KEY ASPECTS OF CMSIS

The generic structure of the CMSIS files is shown in Figure 2.

First, the root directory indicates the version of the CMSIS (e.g. CMSIS_V1P30 is CMSIS version 1.30). Next there is a high level directory for each Cortex family (e.g. M0, M3). For each Cortex-M family, there is a core C source and header file.

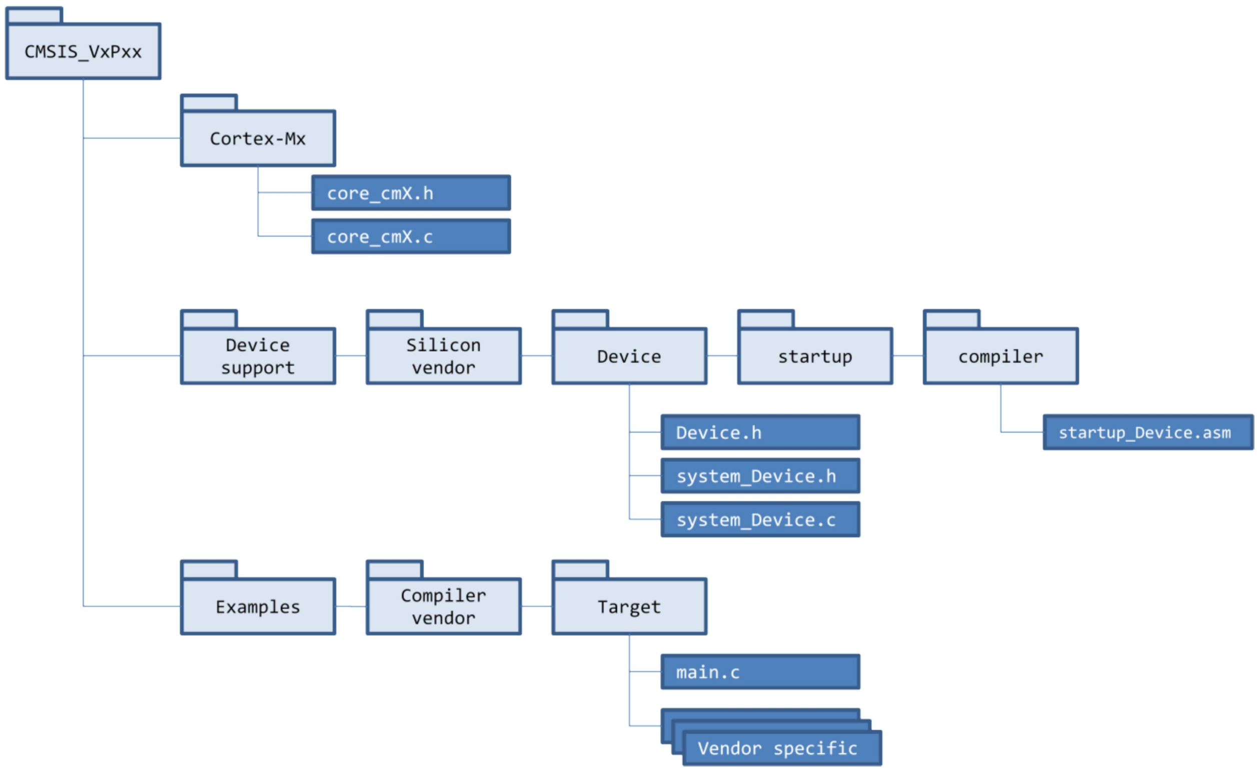

FIGURE 2 CMSIS GENERIC FILE STRUCTURE

Following on from the core files are files for specific silicon vendor's (e.g. NXP, ST, etc.) devices. Central to the concept of a device is the file "`Device.h`". This is the key configuration file for the given Cortex-M implementation. In addition it also defines register structures and access macros for major peripherals above and beyond the Cortex-M core.

Within `Device.h` are three very important pre-processor symbols configured on a per-device basis. These define:

- Whether the Memory Protection Unit (MPU) is present (`__MPU_PRESENT`)
- The number of bits used for priority levels in the NVIC (`__NVIC_PRIO_BITS`)
- Whether SysTick configuration function is supplied by the silicon vendor or whether the default (in `core_cmX.h`) is used (`__Vendor_SysTickConfig`)

Also `Device.h` defines an enum specifying the Cortex-M processor exceptions numbers.

Next in this directory is the file `system_Device.h`. This file has a common set of declarations:

- `void SystemInit (void)`
- `uint32_t SystemCoreClock`
- `void SystemCoreClockUpdate (void)`

The accompanying definitions are found in `system_Device.c`. The `SystemInit` function is designed to be the entry point for the executable code in CMSIS. It is worth noting that when the Cortex-M processor resets it initially reads two 32-bit values from memory:

- starting value for the stack pointer at address 0x00000000
- starting value for the program counter at address 0x00000004

The reset handler defined by 0x00000004 is expected to do some basic housekeeping and then call `SystemInit`. It is intended that `SystemInit` should do any board specific configuration such as setting up the Phase-Lock-Loop (PLL) and any memory access models where required.

`SystemCoreClock` is a global variable that contains the system core clock frequency. This mainly is used to setup the SysTick timer.

For a given device, there is then a compiler specific assembler file `startup_Device.asm`. This defines the Interrupt Vector Table (IVT) and the reset handler code.

NXP LPC17XX

As an example of this generic file structure, we shall take a look at the NXP LPC17xx (6). The LPC17xx family are Cortex-M3 based microcontrollers. The specific device we are going to use as a reference is the LPC1768 (7).

In addition we shall use the IAR Embedded Workbench as the example compiler (8).

The directory and file structure for the NXP LPC17xx / IAR configuration is shown in Figure 3.

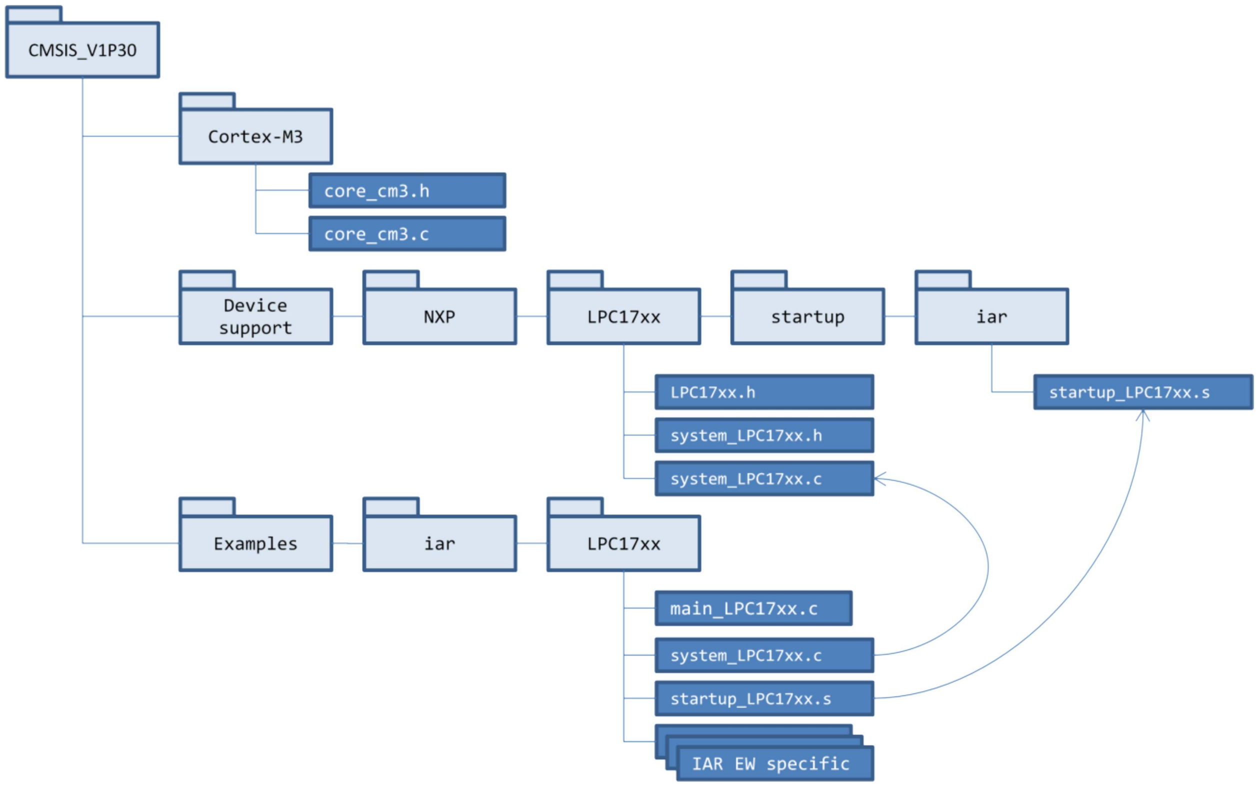

FIGURE 3 NXP LPC17XX / IAR STRUCTURE

As the LPC17xx is Cortex-M3 based, the core files are "core_cm3.h" and "core_cm3.c". The files "`core_cm3.h`" and "`core_cm3.c`" are standard across all vendor devices that have the Cortex-M3 at their core. A fragment of "`core_cm3.h`" is shown below.

```
#include <stdint.h>                           /* Include standard types */

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

typedef struct
{
  __IO uint32_t ISER[8];                      /*!< Offset: 0x000  Interrupt Set Enable Register
*/
       uint32_t RESERVED0[24];
  __IO uint32_t ICER[8];                      /*!< Offset: 0x080  Interrupt Clear Enable Register
*/
       uint32_t RSERVED1[24];
  __IO uint32_t ISPR[8];                      /*!< Offset: 0x100  Interrupt Set Pending Register
*/
       uint32_t RESERVED2[24];
  __IO uint32_t ICPR[8];                      /*!< Offset: 0x180  Interrupt Clear Pending Register
*/
       uint32_t RESERVED3[24];
  __IO uint32_t IABR[8];                      /*!< Offset: 0x200  Interrupt Active bit Register
*/
       uint32_t RESERVED4[56];
  __IO uint8_t  IP[240];                      /*!< Offset: 0x300  Interrupt Priority Register (8Bit wide)
*/
       uint32_t RESERVED5[644];
  __O  uint32_t STIR;                         /*!< Offset: 0xE00  Software Trigger Interrupt Register
*/
}  NVIC_Type;

/* Memory mapping of Cortex-M3 Hardware */
#define SCS_BASE            (0xE000E000)                          /*!< System Control Space Base Address
*/
#define NVIC_BASE           (SCS_BASE +  0x0100)                  /*!< NVIC Base Address
*/
#define NVIC                ((NVIC_Type *)          NVIC_BASE)     /*!< NVIC configuration struct
*/

#define __NOP                __nop

#if (!defined (__Vendor_SysTickConfig)) || (__Vendor_SysTickConfig == 0)
static __INLINE uint32_t SysTick_Config(uint32_t ticks)
{
…
}
```

Looking at the fragment, the first line has the include directive for the C99 library `<stdint.h>` as required by the CMSIS standard. The type uint_32t is defined within `<stdint.h>.` The next three #defines set up common symbols used to define register access; Input (__I) for read-only, Output (__O) for write-only, and Input-Output (__IO) for read-write registers. Unfortunately C cannot define "write-only" registers, so __O is a slight misnomer. In theory these symbols could be used to statically check code for attempts to read a write-only register.

Next we have a typedef'd structure for the central onboard devices register set. It is important to note that this is relying on the structure being packed accordingly (pedantically that cannot be guaranteed, but in reality it is okay for ARM compilers due to the ARM ABI (9)). The device shown here is the common Nested Vectored Interrupt Controller (NVIC).

As part of the Cortex-M memory map, ARM has pre-defined from address 0xE0000000 to the top of memory (0xFFFFFFFF)as a system region. The Cortex-M core peripherals are all defined within this system region (note these are common across all M3 implementations). For example, the first set of registers for the NVIC are called Interrupt Set-Enable Registers (ISER). There are up to eight of these, starting at address 0xE000E100. The `#define` for `SCS_BASE` acts as an offset for all system peripherals. The `NVIC_BASE` is defined at the appropriate address offset from the `SCS_BASE` (+0x100). Finally we see a symbol (`NVIC`) defined that enables a programmer, via the struct, to access the NVIC's registers using a standard syntax, i.e.

`NVIC->ISER[0]`

Following on we see an example of one of the common ARM op-codes (no operation - `NOP`) being defined. This example is IAR dependent and we'll see how other compilers are supported later.

Finally, there are the static inline definitions for key device configurations, such as the NVIC and the SysTick.

One last part of the core functionality is a set of assembler functions to access core registers within the Cortex-M3. For example PRIMASK is a 1-bit register which, when set, masks all exceptions apart from the Non-Maskable Interrupt (NMI) and the hard-fault exception. An accessor function is declared in core_cm3.h

```
uint32_t __get_PRIMASK(void)
```

and defined in core_cm3.c

```
__ASM uint32_t __get_PRIMASK(void)
{
   mrs r0, primask
   bx lr
}
```

Again, the definition of this function is IAR compiler specific.

Following on from the core files are the device specific grouping. In this example we shall examine the LPC17xx device specific files. The central file `LPC17xx.h` defines an enum containing the exception numbers for the common Cortex-M3 exceptions and the LPC17xx specific interrupts (show below). The Cortex-M3 common exceptions are all negative values. Device-specific interrupt numbers always start at zero (0). In the example all LPC17xx microcontrollers have a Watchdog timer (WDT) defined as interrupt number 0.

```
typedef enum IRQn
{
/******  Cortex-M3 Processor Exceptions Numbers ***************************************************/
  NonMaskableInt_IRQn          = -14,      /*!< 2 Non Maskable Interrupt                          */
  MemoryManagement_IRQn        = -12,      /*!< 4 Cortex-M3 Memory Management 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                */
/******  LPC17xx Specific Interrupt Numbers *******************************************************/
  WDT_IRQn                     = 0,        /*!< Watchdog Timer Interrupt                          */
  TIMER0_IRQn                  = 1,        /*!< Timer0 Interrupt                                  */
  TIMER1_IRQn                  = 2,        /*!< Timer1 Interrupt                                  */
  TIMER2_IRQn                  = 3,        /*!< Timer2 Interrupt                                  */
  TIMER3_IRQn                  = 4,        /*!< Timer3 Interrupt                                  */
  UART0_IRQn                   = 5,        /*!< UART0 Interrupt                                   */
  UART1_IRQn                   = 6,        /*!< UART1 Interrupt                                   */
  UART2_IRQn                   = 7,        /*!< UART2 Interrupt                                   */
  UART3_IRQn                   = 8,        /*!< UART3 Interrupt                                   */
  PWM1_IRQn                    = 9,        /*!< PWM1 Interrupt                                    */
```

For device-specific peripherals, such as the WDT, LPC17xx.h defines a typedef'd struct and access macros similar to the NVIC as described earlier. Referring to the LPC17xx System Memory Map shown in Figure 4, the LPC1768 has two Advanced Peripheral Buses (APB), APB0 and APB1. Each APB peripheral area is 1 MB in size and is divided to allow for up to 64 peripherals. Each peripheral is allocated 16kB of space. This simplifies the address decoding for each peripheral. From the system memory map it can be observed that the APB0 base address is 0x40000000, and the WDT is the first peripheral in that region.

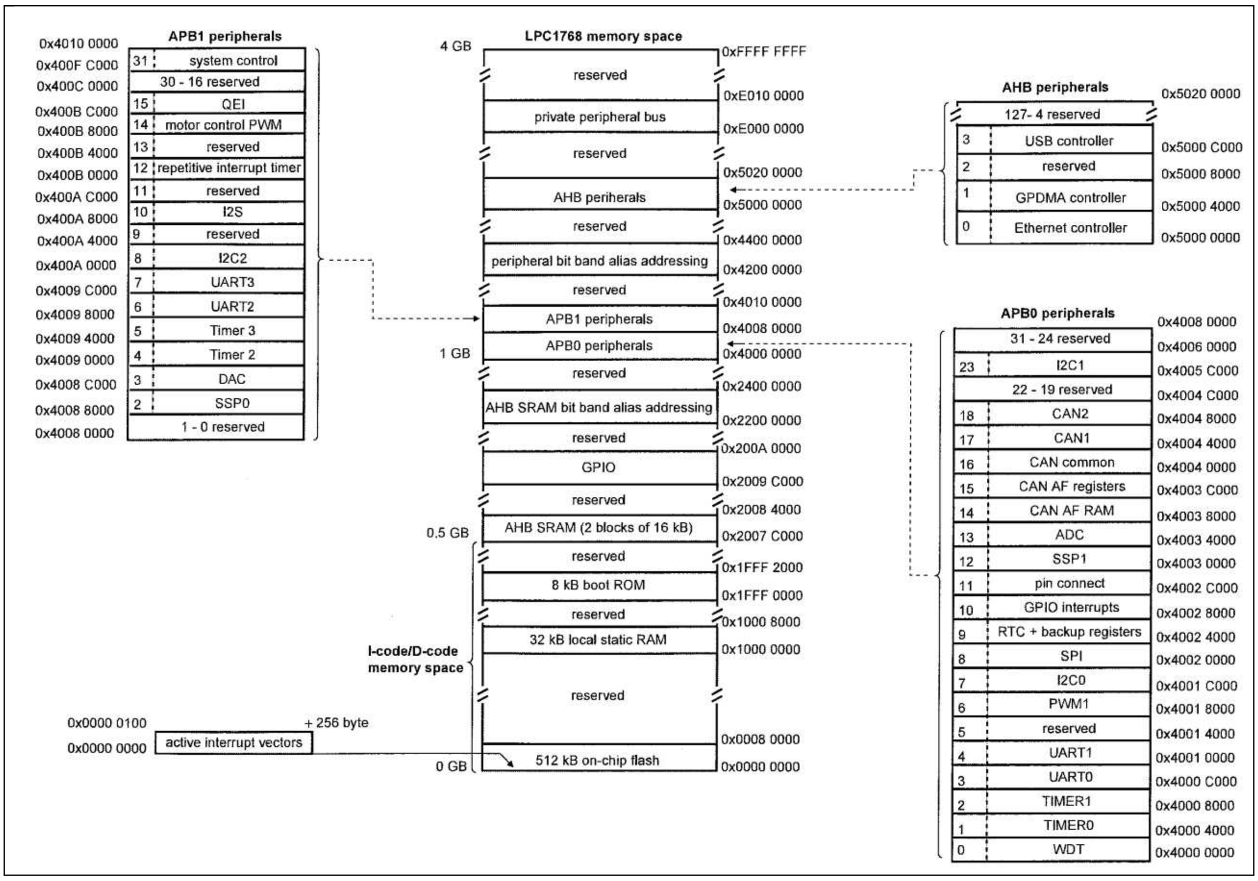

FIGURE 4 LPC17XX SYSTEM MEMORY MAP

Access definition for LPC17xx system peripherals are defined within `LPC17xx.h` with the fragment for the WDT definitions shown below.

```
/*------------- Watchdog Timer (WDT) ------------------*/
typedef struct
{
  __IO uint8_t  WDMOD;
       uint8_t  RESERVED0[3];
  __IO uint32_t WDTC;
  __O  uint8_t  WDFEED;
       uint8_t  RESERVED1[3];
  __I  uint32_t WDTV;
  __IO uint32_t WDCLKSEL;
} LPC_WDT_TypeDef;

#define LPC_APB0_BASE         (0x40000000UL)

/* APB0 peripherals                                                */
#define LPC_WDT_BASE          (LPC_APB0_BASE + 0x00000)

#define LPC_WDT               ((LPC_WDT_TypeDef       *) LPC_WDT_BASE      )
```

The setup for the LPC17xx specific WDT follows a similar pattern to the NVIC. The initial struct defines the WDT's registers. Next we have the definition of the APB0 based address, and the WDT's base address. Finally we have the symbol defined allowing standard access to the registers through the struct, i.e.

`LPC_WDT->WDMOD`

Also of note in the file are the `#defines` that configure the LPC17xx as having an MPU, using 5 bits for the priority level, and using the default SysTickCofig (defined in `core_cm3.h`):

```
/* 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
*/
```

File `system_LPC17xx.h` has, as stated previously, the common set of declarations:

- `void SystemInit (void)`
- `uint32_t SystemCoreClock`
- `void SystemCoreClockUpdate (void)`

The file `system_LPC17xx.c` has the accompanying definitions. For the LPC17xx the `SystemInit` implementation sets up a variation of two PLLs, the peripheral clock sources and a Flash accelerator. The actual configuration is managed through conditional compilation with the controlling symbols also being defined in `system_LPC17xx.c`.

`SystemCoreClock` contains the system core clock frequency, which is also configured via a set of macros based on the PLL configuration. The `SystemCoreClockUpdate` function resets the `SystemCoreClock` based on the current PLL configuration, but is a run-time configuration rather than a compile time configuration.

Finally under the `startup->iar` subdirectory is the file `startup_LPC17xx.s`. There are four major parts to this file:

1. Import and export symbols
2. Interrupt Vector Table definition
3. Initial Reset Handler definition
4. Default interrupt handlers

At the head of the file is `SECTION` directive (`.intvec`) that is used by the linker to ensure the IVT is placed at the correct address in memory. Next are a series of exported symbol declarations (`PUBLIC`) and imported global symbols (`EXTERN`). Shown below are the external symbols `SystemInit` (from `system_LPC17xx.c`) and `__iar_program_start` (which we shall see the relevance of shortly).

```
SECTION .intvec:CODE:NOROOT(2)

EXTERN  __iar_program_start
EXTERN  SystemInit
PUBLIC  __vector_table
```

Next is the actual definition of the IVT. `DCD` is an IAR assembler directive to generate a 32-bit (double word) Constant Data. The first entry is the starting stack address. Here IAR use a directive that calculates the end of the `CSTACK` memory segment. The `CSTACK` is defined in the linker script (again we shall see this later). The second entry is the program counter initial value. This is set to the `Reset_Handler`, an assembler routine defined within the same file.

```
__vector_table
        DCD     sfe(CSTACK)
        DCD     Reset_Handler

        DCD     NMI_Handler
        DCD     HardFault_Handler
        DCD     MemManage_Handler
        DCD     BusFault_Handler
        DCD     UsageFault_Handler
__vector_table_0x1c
        DCD     0
        DCD     0
        DCD     0
        DCD     0
        DCD     SVC_Handler
        DCD     DebugMon_Handler
        DCD     0
        DCD     PendSV_Handler
        DCD     SysTick_Handler

        ; External Interrupts
        DCD     WDT_IRQHandler            ; 16: Watchdog Timer
        DCD     TIMER0_IRQHandler         ; 17: Timer0
        DCD     TIMER1_IRQHandler         ; 18: Timer1
        DCD     TIMER2_IRQHandler         ; 19: Timer2
```

Referring back to the enum defined in `LPC17xx.h`, WDT is set as enum value zero. All the DCD's before the `WDT_IRQHandler` are common Cortex-M3 vector offsets. The WDT is the first LPC17xx specific entry (i.e. there are 16 entries in the IVT before the WDT).

The next significant section of the `startup_LPC17xx.s` file is the definition for the `Reset_Handler`. This simply calls the `SystemInit` function and then calls the IAR common entry point for the C runtime setup (zeroing the bss, copying the idata, etc.) .

```
Reset_Handler
        LDR     R0, =SystemInit
        BLX     R0
        LDR     R0, =__iar_program_start
        BX      R0
```

Finally we have a set of default handlers for all defined interrupts in the IVT. Here the `SysTick_IRQHandler` (as defined in the IVT with the DCD entry one before the WDT) is implemented as an infinite loop. Therefore any unhandled interrupt will end up in a handler of this form. The most interesting keyword to note here is "`PUBWEAK`". `PUBWEAK` defines this label and code as a "weak" linkage symbol. A weak symbol allows a compiler to substitute this definition with another (strong) definition if found during link time. However, if no symbol is found this one will be used (eliminating the problem of unresolved symbols). This will become clear later when we see how a strong symbol is created.

```
PUBWEAK SysTick_Handler
        SECTION .text:CODE:REORDER(1)
SysTick_Handler
        B SysTick_Handler
```

This covers the files under the directory structure Device Support. The final group of files are found at `Example/IAR/LPC17xx`. Here there is a pre-built project for a given compiler (IAR) and development board with an example of the specific device (NXP LPC17xx). The examples for the LPC17xx are all based around the ARM/Keil MCB1700 Evaluation Board. The particular chip used on that board is the LPC1768.

Looking back at Figure 3 NXP LPC17xx / IAR structure, the files `system_LPC17xx.c` and `startup_LPC17xx.s` are copies of the files found under the Device Support directory. There are also a number of IAR EW specific files that are used by the IAR development environment; they define the:

- Compiler include path for the M3 and LPC17xx common files
  - `../../../CoreSupport`
  - `../../../DeviceSupport/NXP/LPC17xx`
- Linker make config file
  - `$TOOLKIT_DIR$\CONFIG\generic_cortex.icf`

The file "`generic_cortex.icf`" is used by the linker for resolving addresses for the different program segments (code, static data, etc.). Examining this file (shown below) we can see two of the symbols we looked at earlier. First the **.intvec** section is placed at address 0x00000000. Next the **CSTACK** is defined to be 0x400 in size and placed in RAM.

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__   = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__     = 0x0007FFFF;
define symbol __ICFEDIT_region_RAM_start__   = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__     = 0x2000FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__   = 0x400;
define symbol __ICFEDIT_size_heap__     = 0x800;
/**** End of ICF editor section. ###ICF###*/
define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];
define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };
initialize by copy { readwrite };
do not initialize  { section .noinit };
place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };
place in ROM_region   { readonly };
place in RAM_region   { readwrite, block CSTACK, block HEAP };
```

The file that brings the example together is `main_LPC17xx.c`. There are two parts specific to CMSIS. First is the definition of a function `SysTick_Handler`. At link time this function will become the "strong" definition for this symbol, and get placed in the IVT by the linker.

```
void SysTick_Handler(void) {
  msTicks++;                        /* increment counter necessary in Delay() */
}
```

Also, at the start of main is a call to `SysTick_Config` defined in `core_cm3.h` (as symbol `__Vendor_SysTickConfig` is set to 0 in `LPC17xx.h`) and using `SystemCoreClock` from `system_LPC17xx.c`.

```
if (SysTick_Config(SystemCoreClock / 1000)) { /* Setup SysTick Timer for 1 msec interrupts  */
    while (1);                                  /* Capture error */
  }
```

We now have an overview of the CMSIS structure for one particular device and one particular compiler.

WHAT IS DIFFERENT FROM COMPILER TO COMPILER?

If we look to use a different compiler for the same device then what, specifically, are going to be the differences in CMSIS?

Within the headers and the C source files, different compiler variants are managed through conditional compilation with compiler-specific directives. Currently CMSIS supports four compilers:

- ARM (Keil)
    - `__CC_ARM`
- IAR
    - `__ICCARM__`
- GCC
    - `__GNUC__`
- Tasking
    - `__TASKING__`

CORE_CM3.H / C

Each compiler has a set of extended keywords; the two important ones used in CMSIS are:

- asm – this is classed as a "Common Extension" in C99 (J.5.10)
- inline – this is defined as a function specifier (6.7.4)

In `core_cm3.h`, the extended keywords redefined to a common symbol by compiler specific #defines as shown in Table 1 common #defines.

TABLE 1 COMMON #DEFINES

#define	ARM	IAR	GCC	Tasking
__ASM	__asm	__asm	__asm	__asm
__INLINE	__inline	inline	inline	inline

When programming embedded systems it can be useful to get access processor specific operators. These are referred to as *intrinsic operations*; examples for the Cortex-M3 include:

- `nop` - no operation
- `wfe` - wait for event
- `wfi` - wait for interrupt

The intrinsic operations are also managed using conditional compilation, as shown in Table 2 intrinsics.

TABLE 2 INTRINSICS

#define	ARM	IAR	GCC	Tasking
nop	__nop	__no_operation	__ASM volatile ("nop");	implemented as intrinsic
wfi	__wfi	__ASM ("wfi");	__ASM volatile ("wfi");	implemented as intrinsic
wfe	__wfe	__ASM ("wfe");	__ASM volatile ("wfe");	implemented as intrinsic

Core register access requires assembler-based functions. These could be supplied in assembler files, but for maintenance it is preferable to implement these using C function wrappers and the asm directive. For example, the function `uint32_t __get_PRIMASK(void) is` declared in `core_cm3.h` (e.g. `__get_PRIMASK) and` is conditionally defined in `core_cm3.c`. For example, previously we saw the IAR version; the code snippet below shows the GCC version of the same function.

```
#elif (defined (__GNUC__)) /*------------------ GNU Compiler ---------------------*/

uint32_t __get_PRIMASK(void)
{
  uint32_t result=0;
  __ASM volatile ("MRS %0, primask" : "=r" (result) );
  return(result);
}
```

STARTUP_DEVICE.S

As already mentioned CMSIS uses assembler files to manage startup. The startup assembler files (all called `startup_<Device>.s`) are all compiler specific. Key items are the:

- IVT definitions
- Implementation of default ISR handlers
    - "weak" definitions
- Import/Export directives
    - E.g. `SystemInit`

One important facet to note is that the implementers of CMSIS have chosen to use anonymous struct/union in the type definitions of device registers. For example, the snippet of code below shows the use of anonymous unions.

```
typedef struct
{
  union {
  __I  uint8_t  RBR;
  __O  uint8_t  THR;
  __IO uint8_t  DLL;
       uint32_t RESERVED0;
  };
  union {
  __IO uint8_t  DLM;
  __IO uint32_t IER;
  };
…

} LPC_UART_TypeDef;
```

The anonymous union shown here allows the members RBR, THR and DLL to all resolve to the same address (as you'd expect from a union). However, the difference is that by being part of an anonymous union they don't create an extra level of indirection, so if we have our pointer to the struct defined:

```
#define LPC_UART0             ((LPC_UART0_TypeDef     *) LPC_UART0_BASE    )
```

Then we can access the symbols directly as:

LPC_UART0->RBR
LPC_UART0->THR
LPC_UART0->DLL

Note, however, anonymous structures/unions are a defined part of the C++ language but they are not part of the current C standard (which, ironically, breaks rule 1.1 of MISRA-C:2004).

NXP LPC17XX COMPILER SPECIFICS

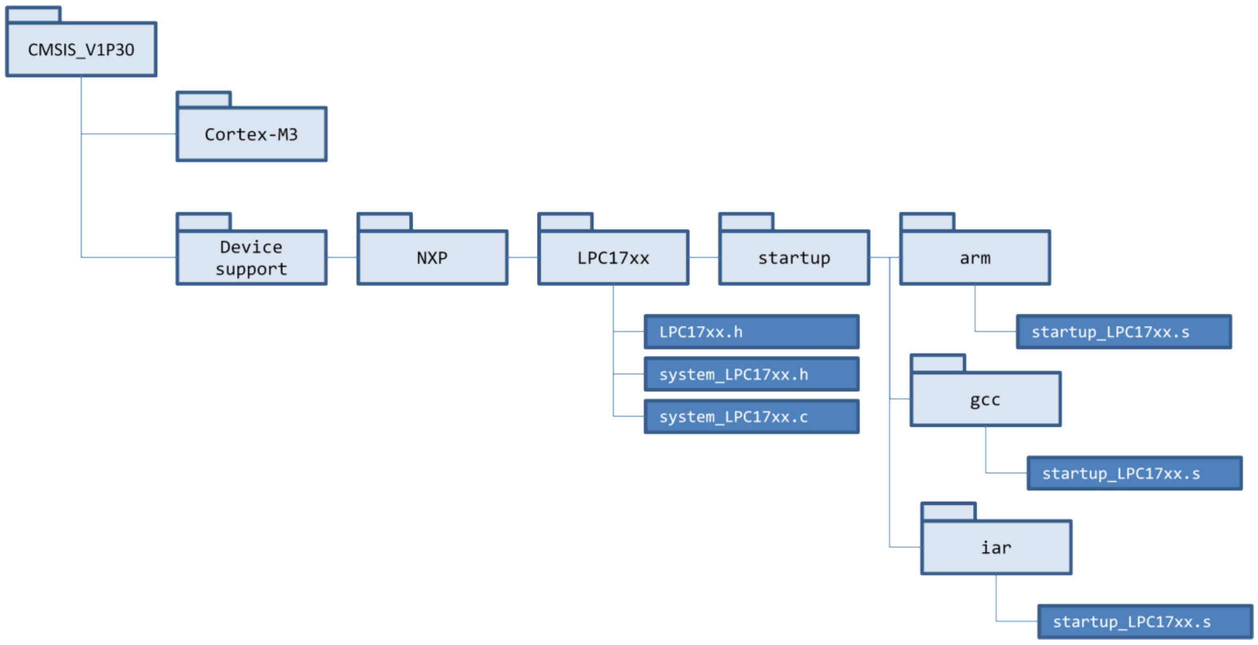

FIGURE 5 LPC17XX COMPILER SPECIFIC FILES

Referring to Figure 5 LPC17xx compiler specific files, the files `system_LPC17xx.h` and `system_LPC17xx.c` have no compiler dependences. However `LPC17xx.h` requires support for anonymous struct/union, but only for the ARM(Keil) compiler.

```
#if defined ( __CC_ARM   )
#pragma anon_unions
#endif
```

Even though Tasking is one of the compiler directives used within the CMSIS core files, there are no examples using Tasking for Device Support or Examples.

STARTUP_LPC17XX.S

The key variations in `startup_LPC17xx.s` for the different compilers are shown in Table 3 startup_lpc17xx.s variations.

TABLE 3 STARTUP_LPC17XX.S VARIATIONS

	IVT definition	default ISR handlers	Import	Export

arm	`DCD SysTick_Handler`	`SysTick_Handler PROC`
		`EXPORT  SysTick_Handler [WEAK]`
		`      B       .`
		`      ENDP`
			`IMPORT  SystemInit`	`EXPORT  __Vectors`

iar	`DCD SysTick_Handler`	`PUBWEAK SysTick_Handler`
		`      SECTION .text:CODE:REORDER(1)`
		`SysTick_Handler`
		`      B SysTick_Handler`
			`EXTERN  SystemInit`	`PUBLIC  __vector_table`

gcc	`.long SysTick_Handler`	`.weak   SysTick_Handler`
		`    .type   SysTick_Handler, %function`
		`SysTick_Handler:`
		`    B       .`
		`    .size   SysTick_Handler, . -`
		`SysTick_Handler`
			`EXTERN  SystemInit`	`.globl`
				`__cs3_interrupt_vector_cor`
				`tex_m`

Also in `startup_LPC17xx.s` we have the Reset Handlers. Shown below are the examples for ARM:

```
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

and GCC:

```
__cs3_reset_cortex_m:
    .fnstart
    LDR     R0, =SystemInit
    BLX     R0
    LDR     R0,=_start
    BX      R0
```

EXAMPLES DIRECTORY

Finally, in the examples directory we can see for each compiler there is a separate directory structure. The example code is all based on targeting the Keil MCB1700 Evaluation Board. Note that the GCC example is based around the Code Sorcery G++Lite environment (10).

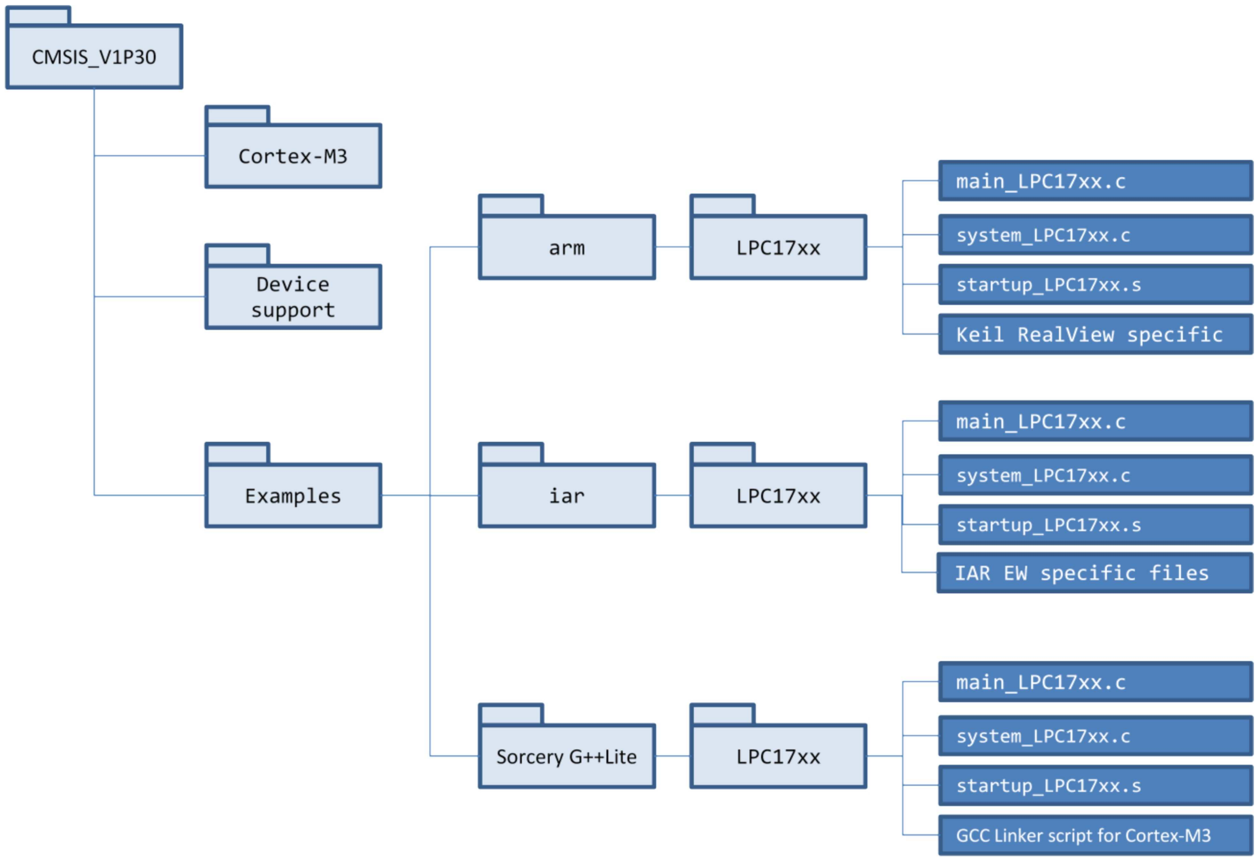

FIGURE 6 COMPILER SPECIFIC DEVICE SUPPORT

WHAT IS DIFFERENT FROM CHIP TO CHIP?

In the previous section we examined the differences in CMSIS for different compilers. Here we take an alternative viewpoint examining the differences when comparing two devices. Here we shall use the Cortex-M3 and IAR EW as common factors. We shall compare support for CMSIS for the NXP LPC17xx compared with the STMicro STM32F10x.

NXP LPC17XX

The NXP LPC17xx family of microcontrollers supports variations on how much on-chip Flash and SRAM memory there is and the support for various peripheral devices such as Ethernet, USB, CAN, I^{2}S, etc. (Figure 7 NXP LPC17xx Architecture).

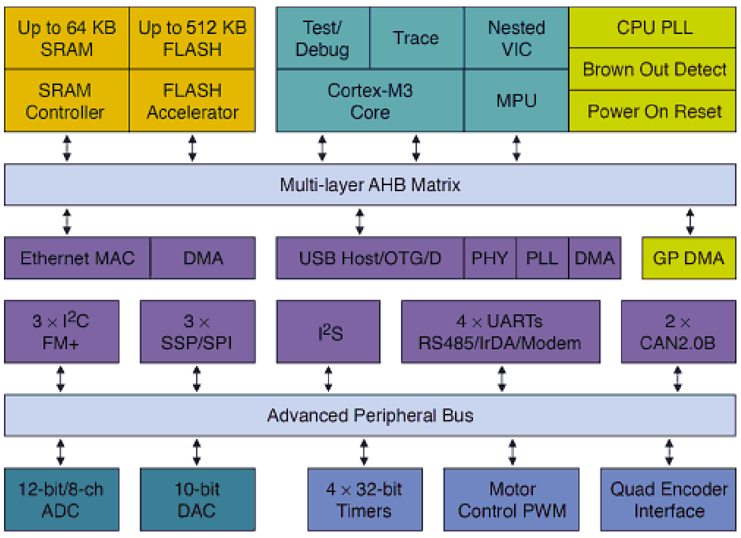

FIGURE 7 NXP LPC17XX ARCHITECTURE

The CMSIS codebase covers all family variants within the generic device-specific files:

- `LPC17xx.h`
- `system_LPC17xx.h`
- `system_LPC17xx.c`

and the compiler-specific startup file (`startup_LPC17xx.s`).

These files have been written to support the “broadest” microcontroller currently in the family (LPC1768) which supports the most memory (Flash and SRAM) and has support for all the possible peripherals.

STM32F10X

STMicroelectronics define the STM32F10x family in the following way (11):

- Connectivity line (107 and 105)
- Performance line (103)
- USB Access line (102)
- Access line (101)
- Value line (100)

However, CMSIS define the STM32F10x family the following way:

- STM32F10X_CL: Connectivity line devices
- STM32F10X_HD: High density devices
- STM32F10X_LD: Low density devices
- STM32F10X_MD: Medium density devices

The density definition runs perpendicular to the line definitions (i.e. there are high, medium and low density versions of the performance line devices).

DEVICE SUPPORT

Within Device Support the first key difference is that all the NXP code has ARM copyright notices, whereas the ST code has an STMicroelectronics copyright notice throughout. Also compiler support for the STM32 includes a variation on GCC for the ride7 environment (Figure 8).

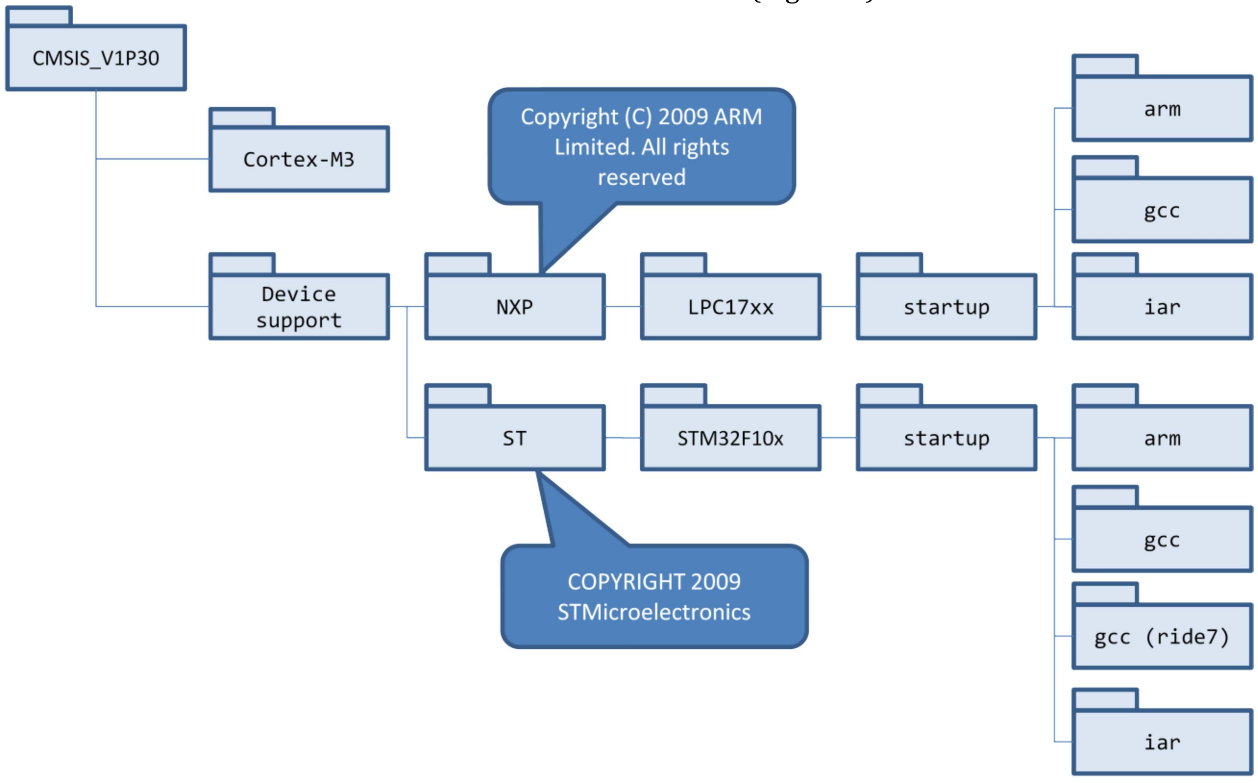

FIGURE 8 DEVICE SUPPORT

The ride7 (12) environment is supplied with STM32 Primer devices (13).

One slightly unusual part of the Device Support area is for the startup files. For arm, iar and gcc (ride7), all four device variants have their own startup assembler file:

- `startup_stm32f10x_cl.s`
- `startup_stm32f10x_hd.s`
- `startup_stm32f10x_ld.s`
- `startup_stm32f10x_md.s`

whereas for gcc, only two of the four are supported:

- `startup_stm32f10x_cl.s`
- `startup_stm32f10x_hd.s`

I'm sure there is a good reason…

Comparing Device Support for IAR for the two devices, the major difference (Figure 9) is there is only one `startup_LPC17xx.s` file; whereas there are the four different `startup_stm32f10x_XX.s` files.

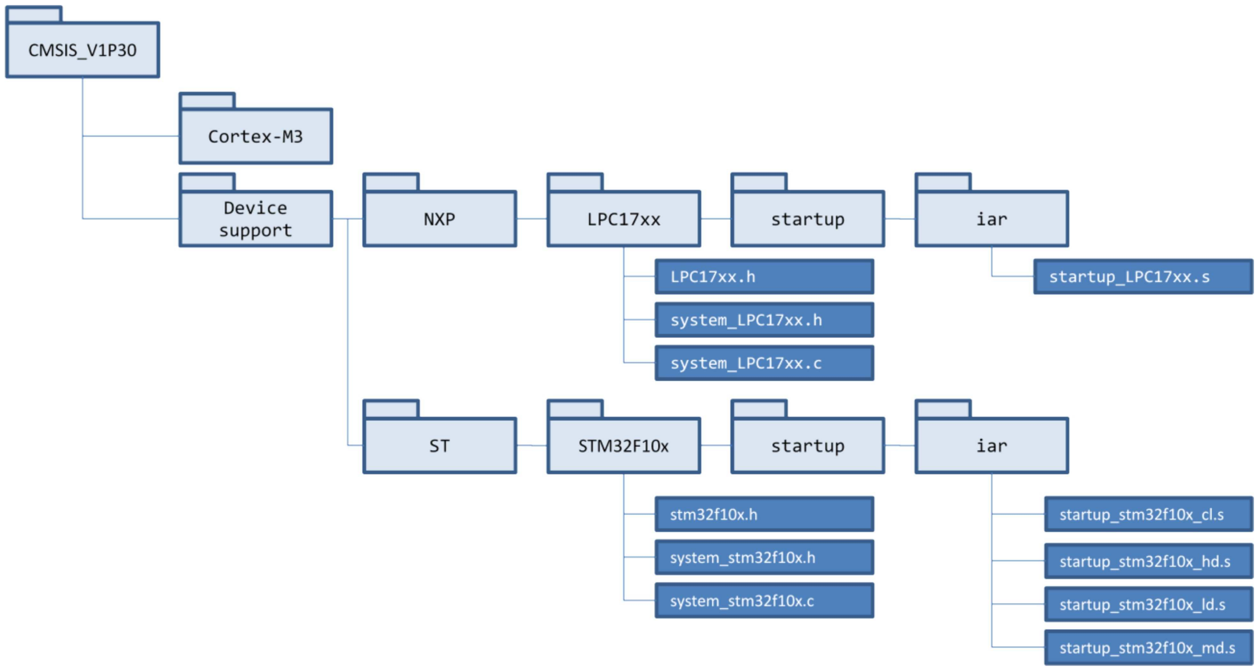

FIGURE 9 DEVICE SUPPORT M3/IAR

How does the `stm32f10x.h` differ from the `LPC17xx.h`?

- The STM32 does not provide an MPU - `#define __MPU_PRESENT 0`
- The STM32 uses 4 Bits for the Priority Levels of the NVIC - `#define __NVIC_PRIO_BITS 4`
- It doesn't define its own SysTick Config - `#define __Vendor_SysTickConfig 0`

As with LPC17xx.h, the Cortex-M3 common initial 16 interrupt vectors for the M3 are defined followed by further defines for the STM32 specific interrupts. With the STM32F10x family, interrupt numbers 0-18 are common, whereas 19-42/62/67 are device (CL, HD, LD & MD) specific. Conditional compilation is used, based on device type, to manage these variants in the interrupt structure. Further there are the STM32 typedefs for peripherals types and macros for peripheral access.

`system_stm32f10x.h` is almost identical to `system_LPC17xx.h`, the only differences are in comments (e.g. copyright notice). Incorrectly, the `system_stm32f10x.h` doesn't include `<stdint.h>` but uses `uint32_t`.

`system_stm32f10x.c` defines `SystemInit`. The supplied code initializes the flash interface and the PLL and updates the `SystemFrequency` variable. `SystemCoreClockUpdate` is an empty routine. `SystemCoreClock` is configured using conditional compilation.

IAR EXAMPLES

The examples supplied for IAR include two variants of the STM32 processor, one for the STM32F103 and one for the STM32F107. The 103 example is based on a high density device variant, whereas the 107 is based on a connectivity line device.

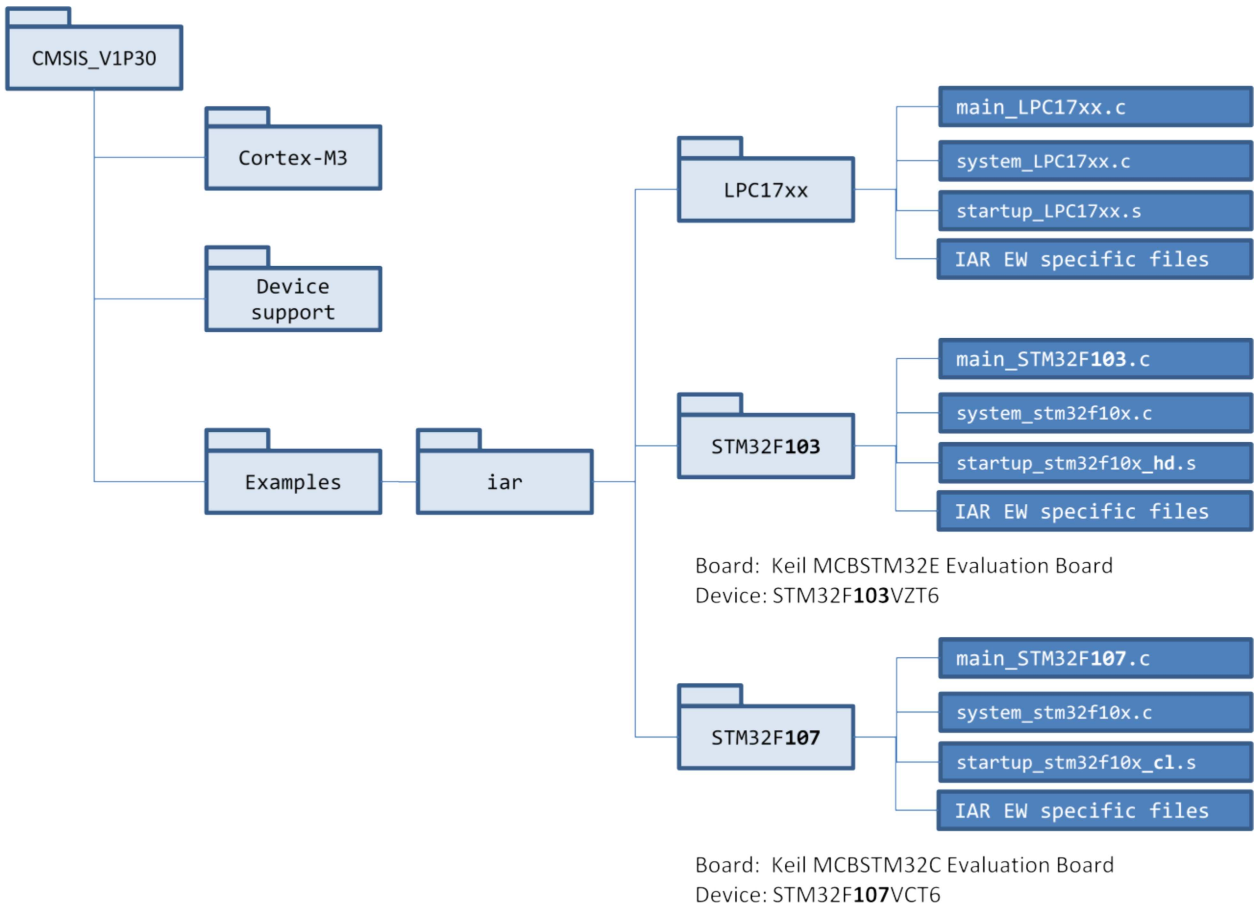

FIGURE 10 IAR EXAMPLES NXP & ST

Finally, looking at Figure 11, there are examples for ARM, IAR and CodeSourcey for the LPC17xx and both STM32F10x's, whereas the Ride7 environment only has examples for the STM32 devices.

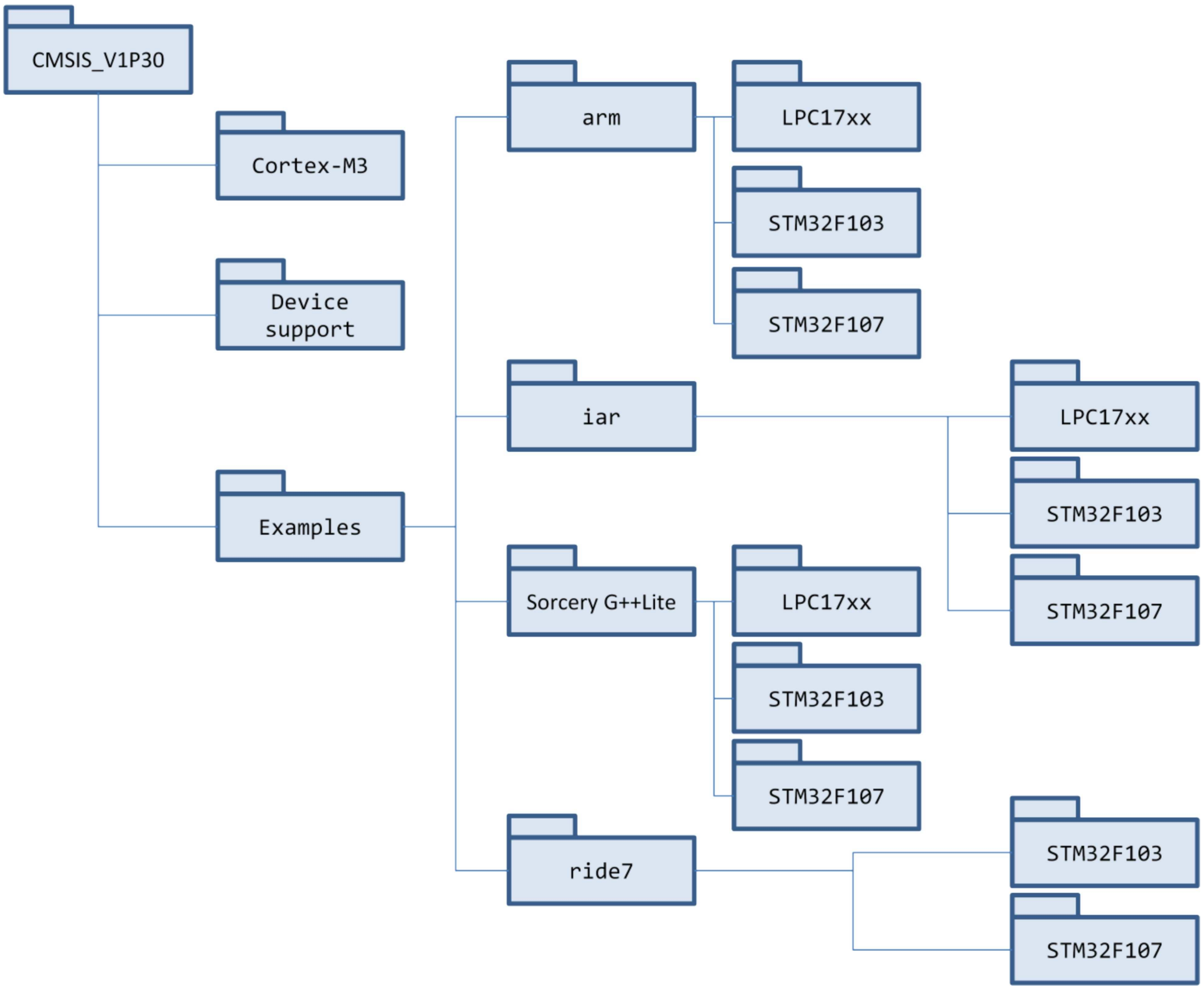

FIGURE 11 DEVICE EXAMPLES

OVERALL THE VENDOR/DEVICE SUPPORT

Overall the Vendor Support in CMSIS v1.30 is summarized in Table 4 Device Vendor Support.

TABLE 4 DEVICE VENDOR SUPPORT

	Atmel	**Energy Micro**	**NXP**	**ST**	**TI**	**Toshiba**
	AT91SAM3U	EFM32	LPC13xx LPC17xx	STM32F103 STM32F107	LM3S	TMPM330
ARM	☑	☑	☑	☑	☑	☑
IAR	☑	☑	☑	☑	☑	☑
Ride7				☑		
GCC	☑		☑	☑	☑	☑

BEYOND THE CORE

Everything we've discussed so far relates to how different compilers and microcontrollers support the Cortex-M3 core and CMSIS. A further area of CMSIS, which currently appears to be less well supported, is the Middleware Access Layer (MAL). The intent of the MAL is that peripheral-devices common across many microcontrollers, but not actually part of the Cortex-M3 core, can also be abstracted to a common interface.

The code for the MAL was supplied as part of the CMSIS V1.20 download (14), but is not included in V1.30.

Initially the MAL defined the following:

- CMSIS UART Device
- CMSIS SPI Device
- CMSIS Ethernet Device

All devices have a similar structure. For example the CMSIS UART Device attempts to provide a standard interface to a USART. The intent is that the file structure for these middleware devices will follow a similar structure to the core, e.g. a set of common files

- `UartDev.h` - Global defines and structure definitions for general UART Device interface.
- `UartDev_Device.h` - Device dependent definitions of the device-specific UART Device Driver.
- `UartDev_Device.c` - Device specific UART Device Driver

In `UartDev.h` a common structure is defined based on function pointers (similar to a C++ v-table). A particular implementation will populate these with device-specific definitions. As of CMSIS V1.20, there are only examples for the STM32F10x (i.e. `UartDev_STM32.h` and .c).

```
/*----------------------------------------------------------------------------
  UART Device
  IO Block Structure
 *----------------------------------------------------------------------------*/
typedef struct {

   /* changed by the user application before call to Init. */
   UartDev_CFG   Cfg;
   /* Initialized by UART driver. */
   int (*Init)    (void);                                   /*!< Initialize */
   int (*UnInit)  (void);                                   /*!< unInitialize */
   int (*BufTx)   (void *pData, int* pSize, unsigned int flags); /*!< Transmit */
   int (*BufRx)   (void *pData, int* pSize, unsigned int flags); /*!< Receive */
   int (*BufFlush)(void);                                   /*!< Flush buffer */
} UartDev_IOB;
```

IN SUMMARY

Adoption of CMSIS has the potential for huge benefits to many different interested parties:

- The compiler vendors can work with the silicon vendors to quickly support new devices
- Application engineers will have a selection of toolsets available for their microcontroller of choice
- RTOS vendors can port their RTOS to new devices quickly
- Common middleware support means higher-level middleware should need little or no porting (e.g. lwIP is a light-weight implementation of the TCP/IP protocol suite)
- And of course ARM establish themselves as the *de facto* standard for embedded 32-bit microcontroller cores (and make a tidy fee in licensing cores)

It's hard to see why CMSIS isn't going to be widely adopted and accepted with open arms (no pun intended).

All that said there are some concerns; the key ones being:

- Who is going to police the CMSIS implementations and ensure quality?
  - Surely it's got to be ARM? But are they then going to commit dedicated resources to CMSIS?
- Who is going to develop Device Support and Examples for new devices?
  - It's expected to be the silicon vendors, but how well do they know the compiler optimizations and what resources will they throw at it?

Why these concerns? While examining the CMSIS code base a few, mainly minor, items just got me pondering.

Take for example inlining. The inline keyword suggests to the compiler that it compiles a C function inline, if it is sensible to do so. However GCC inline does not force the compiler to actually inline the body of a function defined with `__INLINE` attribute.

The semantics of `__forceinline` are exactly the same as those of the C++ inline keyword. The compiler attempts to inline a function qualified as `__forceinline`, regardless of its characteristics. However, the compiler does not inline a function if doing so causes problems.

Conversely the `__get_xxx, __set_xxx, __LDREX, __STREX` groups of C-wrapped assembly functions are implemented as regular functions as opposed to static inline functions.

Coding style wise, `core_cm3.c` does not including `core_cm3.h`. Not a big deal here, but certainly not good practice.

In the SMT32 tree, there are common files having same version number but actually with minor differences (albeit only in the comments). This really does beg the question regarding configuration control.

Finally you've got to question the use of features that are not part of C such as anonymous unions. From a pragmatic viewpoint I understand the rationale for this, but to make a statement about MISRA-C compliance, goes against the grain a bit.

Finally, I would expect the Cortex-M0 audience (and a lot of the M3 audience) to see CMSIS as too complicated and heavyweight (as in this paper it's taken nearly 6000 words to try and give a sensible overview!).

In summary CMSIS has real promise, but there is still work to be done.

BIBLIOGRAPHY

1. Cortex-M Series. [Online] http://www.arm.com/products/processors/cortex-m/index.php.

2. CMSIS Version 1.30. [Online] http://www.onarm.com/download/download395.asp.

3. MISRA Home Page. [Online] http://www.misra.org.uk/.

4. **British Standards Institute.** *The C Standard.* s.l. : John Wiley & Sons Ltd., 2003. ISBN 0-470-84573-2.

5. CamelCase. *From Wikipedia, the free encyclopedia.* [Online] http://en.wikipedia.org/wiki/CamelCase.

6. LPC17xx series. [Online] http://ics.nxp.com/products/lpc1000/lpc17xx/.

7. [Online] http://ics.nxp.com/products/lpc1000/datasheet/lpc1764.lpc1765.lpc1766.lpc1767.lpc1768.lpc1769.pdf.

8. IAR Embedded Workbench® for ARM. [Online] http://www.iar.com/website1/1.0.1.0/68/1/.

9. Application Binary Interface (ABI) for the ARM Architecture . [Online] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html.

10. Sourcery G++ Lite Edition. [Online] http://www.codesourcery.com/sgpp/lite_edition.html.

11. STM32F10x Product Lines. [Online] http://www.st.com/mcu/stonline/img/STM32F10x_value_productline.jpg.

12. Ride7. [Online] http://www.raisonance.com/~ride7__microcontrollers__tool~tool__T018:4cw36y8a5c39.html.

13. STM32 Primer. [Online] http://www.stm32circle.com/hom/index.php.

14. CMSIS Version 1.20. [Online] http://www.onarm.com/download/download389.asp.

15. CMSIS Peripheral Access Layer for STM32F10x V3.1.0. [Online] http://www.onarm.com/download/download390.asp.

TRAINING IN REAL-TIME

EMBEDDED DEVELOPMENT

Feabhas Ltd
5 Lowesden Works
Lambourn Woodlands, Hungerford
Berkshire RG17 7RY, UK

Tel: +44(0)1488 73050
Fax: +44(0)1488 73051

Email: info@feabhas.com
Web: www.feabhas.com