

CMSIS: a truly device-independent framework for ARM users?

The Cortex Microcontroller Software Interface Standard (CMSIS), a vendor-independent Hardware Abstraction Layer (HAL) developed by ARM®, aims to help developers to share software resources across multiple product variants based on Cortex™-Mx MCUs, including devices from different manufacturers.

In this article, Jose Perez, Field Applications Engineer, Future Electronics (Spain), examines how CMSIS influences software portability between two ARM Cortex-M3 MCUs: the STM32F10x from STMicroelectronics, and the LPC17xx from NXP Semiconductors.

Portability strategy

ARM's strategy to support software portability and re-use contains both hardware and software elements. On the hardware side, the Cortex-Mx architecture integrates more functionality into the core itself than in previous ARM-7 architectures. For instance, Cortex-Mx devices from different manufacturers will not only share the same CPU, they will also have the same Nested Vectored Interrupt Controller (NVIC), the same core timer (SysTick), and the same logic module for debugging. This guarantees that any firmware will have the same functionality across all devices. Figure 1 shows the basic architecture of CMSIS.

Despite this, the functionality must still be implemented in code that is common across all IC vendors' devices and CMSIS addresses this requirement. CMSIS is also conceived to allow the developer to migrate easily from one supplier's software tool to that of another supplier. For those using a Real-Time OS (RTOS), the kernel features a device-independent interface, including a debug channel.

READ THIS TO FIND OUT ABOUT:

- Designing with ARM® Cortex™ MCUs from multiple vendors
- Using the Cortex Microcontroller Software-Interface Standard (CMSIS)
- Overcoming limitations of CMSIS

Getting started with CMSIS

CMSIS files can be downloaded from the IC manufacturer's website, and are usually bundled with the manufacturer's peripheral library in the same zipped file. The file packages examined in this article were STM32F10x_StdPeriph_Lib/V 3.2.0 and Lpc17xx.cmsis.driver.library.zip/ V 1.3. Both are based on CMSIS v1.3.

Once unzipped to a hard disk, the two file sets adopt different directory structures. Comparing them is not difficult, however, because the CMSIS files are grouped together in a folder, \\Core\CM3. The remaining folders provided by NXP and STM, which include the peripheral-driver library, project templates, examples and help files, are similar to each other.

Most developers start a new software project by using either the project template or an example project. In fact, ARM supplies a template for each toolchain it supports; each silicon vendor then adapts it to include the interrupt vectors for all device-specific interrupt handlers. This file is named startup_device.s. In the case of the IAR toolchain, it is responsible for:

- Setting the vector-table entries with the exceptions and interrupts that the Interrupt Service Routine (ISR) addresses
- Defining each interrupt handler as a weak function for a dummy handler
- Initialising the stack pointer
- Calling the SystemInit() function and jumping to __iar__program_start

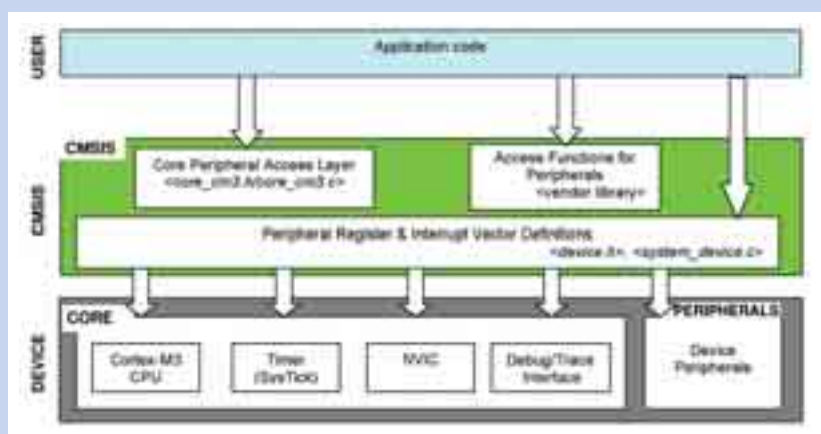


Fig. 1: CMSIS used in an application with no OS

pendent code-generation

Typically, in this last function, variables are either cleared or initialised to values from ROM, as needed, and the Main() function is called, starting the application.

As part of the SystemInit() function, defined in file system_device.c, the oscillator system is initialised and the SystemCoreClock variable is updated.

To summarise, the startup_device.s and system_device.c files supplied by the vendor give the user-defined interrupt vectors and interrupt handlers, initialise the oscillator/PLL and set the core frequency.

Because ARM has set the templates for these files, the functionality of both files and firmware is almost the same for the STM32F10x and the LPC17xx. There is only one small difference: in the NXP device, SystemInit() also initialises the Flash Accelerator. This arises simply because this module is not available in the STM part.

Another important file that the application programmer must include in the C source code is device.h (lpc17xx.h or stm32f10x.h). This file is part of the Device Peripheral-Access Layer and is provided by the silicon vendor; it contains the interrupt numbers for all exceptions and interrupts, and provides definitions for the device peripherals' data structures and address mapping.

Strengths and limitations

A section of this file, named Configuration of Cortex-M3 processor and Core Peripherals, contains the first hint of a limit to the ability of CMSIS to impose a uniform development interface. Differences between the devices now become apparent, arising from their use of different versions of the Cortex-M3 core: the STM device uses rev. 1 while NXP's uses rev. 2, which includes a Memory-Protection Unit (MPU). The other difference is that the STM part uses 4 bits for interrupt-priority levels whilst the NXP part uses 5 bits.

For developers migrating from 8-bit designs using no OS, these differences will barely affect portability: applications with no OS will not usually use an MPU, and four levels of priority will generally be enough.

In other respects, the CMSIS framework ensures uniformity between different devices at the level of core configuration. core_cm3.h is the header file for the Core Peripheral Access Layer. It defines all the structures and symbols for the CMSIS core, including the CMSIS version number, Cortex-M3 core registers and bitfields, and core peripheral mapping.

With the core_cm3.c file, some functions for accessing core registers and for generating specific Cortex-M3 instructions are provided: This is achieved efficiently in the form of static inline functions. It is worth stressing that CMSIS gives the programmer a full set of interrupt-related functions to enable/disable and to manage priorities and flags. The same file provides control of the core SysTick timer and the Instrumentation Trace Macrocell.

Design of core functions, then, enjoys a great deal of uniformity across all devices. However, CMSIS struggles to bridge differences between different IC manufacturers in relation to the treatment of device peripherals. The file, device.h, reveals these differences.

There are two possible approaches to writing code for peripherals: through direct access to the peripheral's register or by using the silicon vendor's peripheral-driver library. In the first case, the programmer can use the peripheral definitions available in device.h; in the second, the

peripheral driver's Application Programming Interface (API) can be used.

These libraries contain macros, data types, structure types and functions that exploit the full capability of the peripheral. They also make this functionality available without any requirement on the developer's part to have in-depth knowledge of the device. Associated with the libraries is a full set of examples. Together, the libraries and examples significantly accelerate project development.

In the case of the NXP and STM libraries, CMSIS compliance ensures a uniform approach to accessing peripherals. This means that implementations of the same action in equivalent peripherals

on devices from STM and NXP appear similar (see Figure 2). In many cases, however, different implementations are required because the LPC17xx and STM32F10x peripherals, such as ADCs, timers, and even GPIOs, differ substantially.

Where this is the case, the best approach is to avoid attempting to

migrate existing code from one device to the other and, instead, to implement a completely new driver for the new device. The vendor's library will again provide valuable help.

CMSIS and software portability

CMSIS is most useful in porting software from one ARM Cortex-Mx device to another when the application is using core resources. The CMSIS definitions of interrupt-management methods and system initialisation are of great help.

In general, the coding rules, conventions and Doxygen style of comments applied by CMSIS help the developer to understand code written for different devices.

The standard technique for defining peripheral management in the peripheral-driver library is, however, only useful in the minority of cases where the same kind of peripheral has similar functionality.

An additional CMSIS layer, the Middleware Layer, is currently under discussion. This new feature could add a higher level of hardware abstraction for the serial peripherals (UART, SPI), bringing a new level of portability.

It should also be said that, in general, both STM and NXP provide excellent implementations of CMSIS; the examples and help files provided enable developers to become more productive.

It is inevitable that different peripheral implementations will emerge from different IC manufacturers and these differences are bound to be reflected in driver designs. However, for the core elements defined by ARM, the CMSIS provides an effective and comprehensive development framework that facilitates software migration.

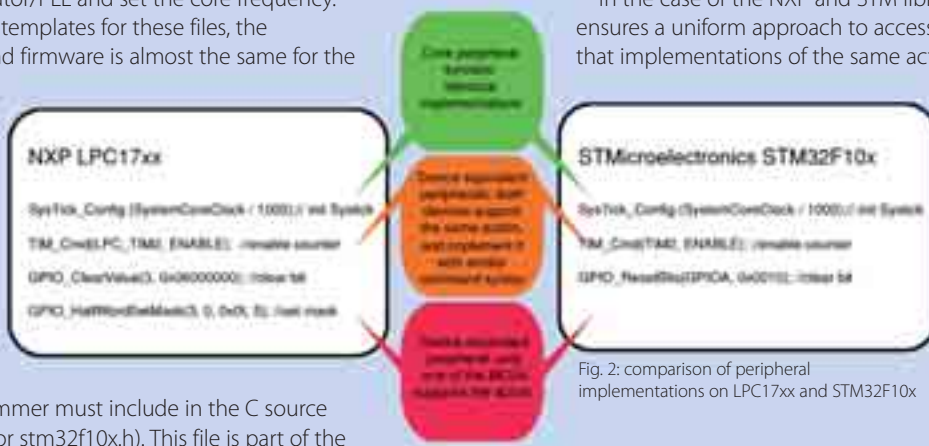


Fig. 2: comparison of peripheral implementations on LPC17xx and STM32F10x

 For parts and pricing go to: my-ftm.com/100734 