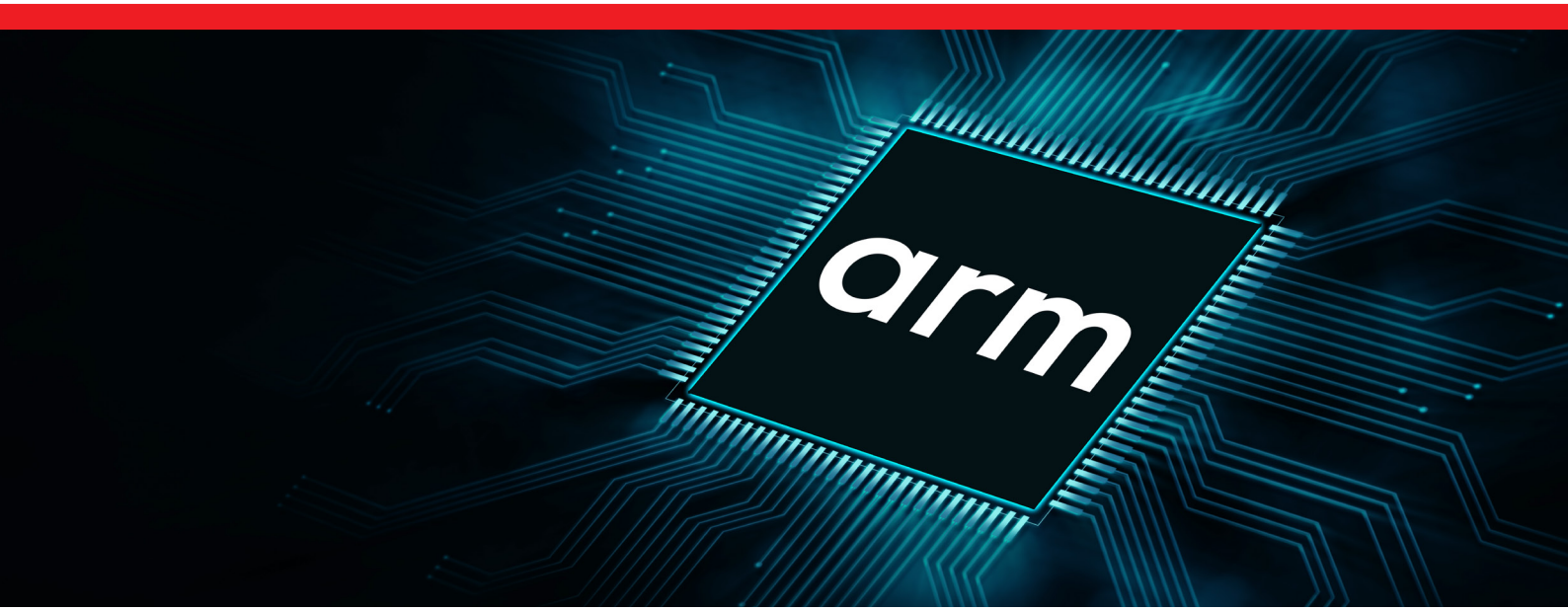
hitex
EMBEDDED TOOLS & SOLUTIONS




# Software Architecture for Arm Cortex-M Microcontrollers:
## Doing it right first time to improve developer productivity

Written by Trevor Martin, Arm Technical Specialist, Hitex (UK) Ltd

hitex
EMBEDDED TOOLS & SOLUTIONS

# Table of Contents

# List of Figures

# List of Tables

## Introduction

In this paper we will look at how we can develop a standard software architecture for small microcontrollers that makes best use of both the software standards developed by Arm and their partners and a small footprint Real Time Operating System (RTOS). The key aim of such an architecture will be to improve developer productivity by providing the following benefits:

**Reuse**

One of the major objectives of this architecture is "do once, use often". This applies not just to source code, but all of a project's collateral including test frameworks, documentation and hardware drivers.

**Testing**

The structure of the architecture is intended to make software testing an integral part of software development.

**Early software development**

A standard driver layer allows the creation of stub functions which promote development on evaluation boards and within software simulators. This allows a software team to proceed before the hardware is available.

**Improved workflow**

A well-structured software architecture will improve collaboration between hardware and software teams and improve workflow within software teams. The proposed architecture can be used within any existing workflow such as Agile or Waterfall.

**Improved design methodology**

By having a standard architectural framework it becomes easier to develop a systematic process to decompose a requirements document into a software design.

## CMSIS

When the first Cortex-M microcontrollers became widely available in 2006, Arm started to create the Cortex Microcontroller Software Interface Standard: CMSIS. This has since grown into a set of interlocking standards that support firmware development across a vast number of microcontrollers from many different vendors.

The CMSIS project provides a range of software standards that define how the 'C' language is used to develop portable microcontroller firmware. The key CMSIS specifications used within our software architecture are defined below:

| Specification | Description |
|---|---|
| CMSIS Core | A set of low-level functions and macros used to access the Cortex-M processor registers. |
| CMSIS RTOS2 | A standard API for small a footprint Real Time Operating System. |
| CMSIS Driver | A standard API for common microcontroller peripherals. |
| CMSIS Pack | Defines a container for reusable software components. |
| CMSIS Build | Provides a standard project and build system for Cortex-M development tools. |
| CMSIS Zone | A high level mark up language and utility that is used to define complex memory maps for multiprocessor projects. |

*Table 1: CMSIS specifications*

## General Purpose MCU Architecture

A proposed general-purpose software architecture that uses a layered model is shown below. This architecture can be used on any Cortex-M microcontroller and can also be used with the Arm Platform Security Architecture (PSA).

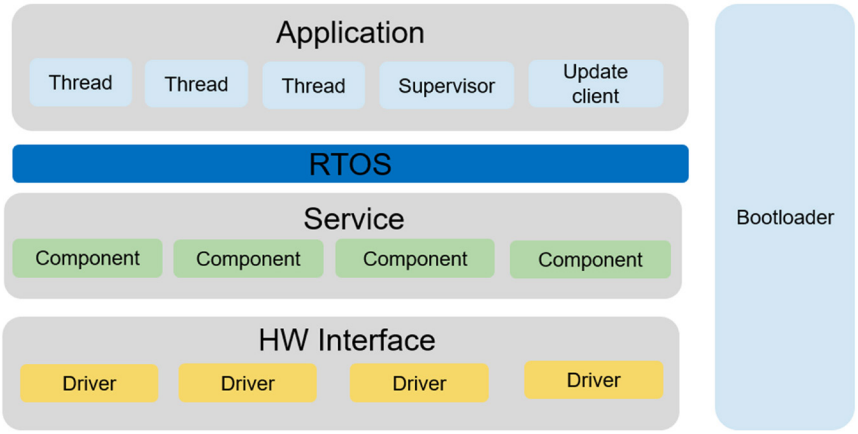



*Figure 1: General purpose MCU Architecture*

This architecture consists of four layers, or three plus one since the RTOS layer can be considered as standard code that does not need any development effort. Our development effort will be focused on creating the remaining three layers. An important feature of this architecture is that each layer except the HW interface layer does not touch any microcontroller registers. Only the driver code and RTOS code will access hardware registers within the microcontroller. This ensures that code in the application and service layers are device independent. This is critical for both software testing and code reuse. A definition of each layer is shown below:

| Software layer | Description |
|---|---|
| Application | The 'business logic' of the design |
| RTOS | Scheduler, memory allocation and inter-process communication |
| Service | A set of software components each providing a service to the application layer |
| Hardware interface | Device independent low-level drivers each with a standard API |

*Table 2: MCU architecture layer definitions*

**Application Layer**

The application layer consists of threads for each 'unit of concurrency' within the design. Ideally the threads will consist of a high-level logic which makes calls to functions within the RTOS and service layers. The application layer will not access the driver layer or hardware registers directly.

Within the application layer we can create some standard threads which can be reused in every project some suggestions are shown below.

**Supervisor thread**

The supervisor thread starts the RTOS and creates the initial threads, during runtime it manages system peripherals such as watchdogs and power management.

**Update Client**
The update client downloads and stores new image updates and will trigger the bootloader to upgrade the system.

**RTOS Layer**
The use of an RTOS provides scheduling, memory management and commonly used primitives (delays, signals, message queues etc). The RTOS layer is recommended to use the standardised CMSIS-RTOS2 API. This separates our application code from a specific vendors RTOS. For example, this allows us to select between a commercial grade RTOS and a safety certified RTOS depending on the project requirements.

**Service Layer**
The bulk of the firmware exists within the service layer and will be designed as reusable software components. Each software component contains a set of associated functions which provide a service to the application layer through a well-defined interface. An example of typical services would be a GUI component, motor control component, communications component. A component may be a simple library of functions or it may create its own threads to provide a more active functionality. The basic structure of a component is shown below:

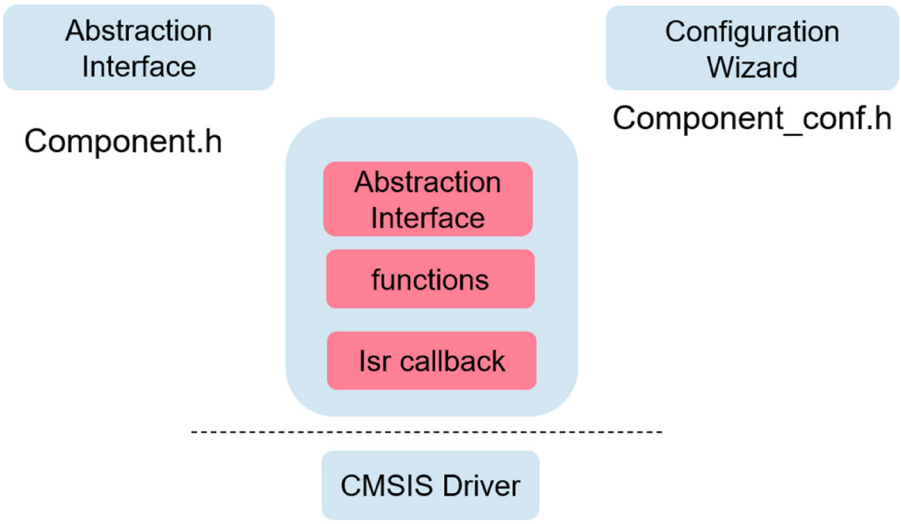



*Figure 2: Basic structure of a component*

There are two key features that a component must meet. First, all of the component source code must be located in a set of dedicated modules and no code from other parts of the project may be included. This ensures that the component is self-contained and easy to reuse. Secondly, like the application layer it is important that no part of the component code accesses any hardware registers, but instead relies on the drivers to access the microcontroller peripherals. This keeps the component device independent. In the case of peripheral interrupts, the driver layer is used to configure and enable peripheral interrupt sources while the executable code for an interrupt service routine is provided through a call-back routine located within the software component. This again isolates the component from specific device hardware.

The component exports its API through a header file. This file only contains function calls and component data is only exposed through ‘helper’ functions. By fully encapsulating data within a component, we can eliminate many common software bugs without going to the complexity of an object orientated language such as C++.

The component is configured through a separate header file which contains all the necessary #define declarations. The CMSIS standard specifies a set of mark-up tags which can be used to create a graphical configuration wizard within the header file. Creating such a wizard doesn't take much effort and when the component is reused it becomes much more intuitive for developers on future projects.

**Driver Layer**

The CMSIS-Driver specification defines a standard API for a wide range of common communication peripherals for microcontrollers (USART, SPI, I2C etc). Silicon vendors will often provide these drivers as part of their standard support. The CMSIS driver API provides a common interface which can be used by our components irrespective of the underlying microcontroller.

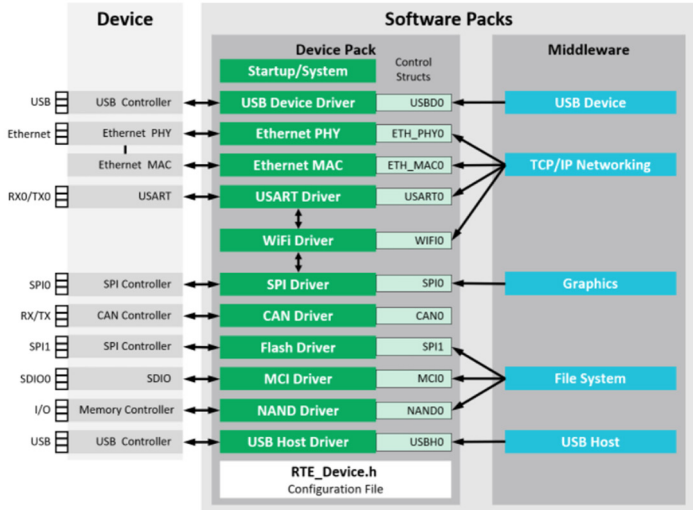



*Figure 3: Diagram of CMSIS Driver Layer*

One problem with this plan is that the range of CMSIS drivers is limited to peripherals that are used by middleware stacks. Typically these are communication interfaces such as Ethernet, USB and SPI. In order to create a general-purpose architecture, it is necessary to extend the range of CMSIS drivers with custom drivers for additional commonly used peripherals (ADC, Timer etc). These additional driver specifications would be defined in-house to provide commonly required feature sets and would follow the existing CMSIS Driver conventions. Once the new peripheral specification is defined a driver template can be created for future driver development. Once this initial start-up work is done the overhead to develop a driver is minimal. The work is in defining the driver profile and creating the base templates, but this is only done once.

**Bootloader**

A further standard component of many designs is a second stage bootloader which is used to update application images.  Here we can adopt a further open source project called MCU Boot. The MCU Boot bootloader is designed to manage multiple executable images which can be signed and encrypted prior to installation. A port of the MCU Boot firmware is now available as part of the ARM PSA trusted firmware. This branch of the MCU Boot firmware has be modified to use CMSIS Flash drivers to make it easily portable between microcontrollers.

## Development

The code base for a typical microcontroller project is often relatively small and it is common to develop the full application in a single project environment. By using a layered architecture with the bulk of our code located in self-contained software components in the service layer we can take a multi project approach to development and place software testing at the heart of our development effort.

In this system elements in each layer can be developed as a standalone project which includes a test framework. This gives us a hierarchy of projects which can be used to prove each software component before integration into the final design as shown below:

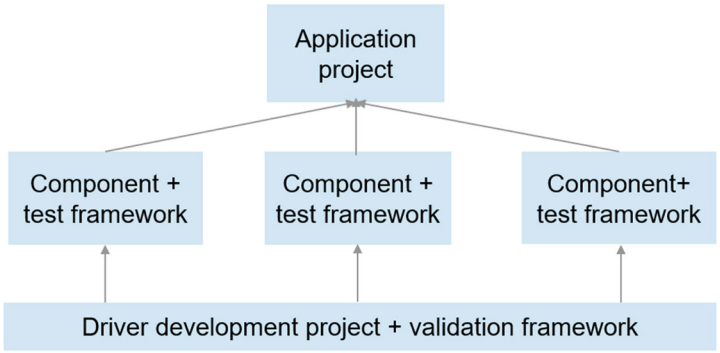



*Figure 4: Multi-project approach framework*

There are a number of open source test frameworks available. Unity is a commonly used framework that is written in 'C' and has been explicitly designed to run within constrained microcontrollers. The Cortex-M3, -M4 and -M7 provide a debug channel called the Instrumentation Trace (ITM) which provides a serial channel to a console window in the debugger for test results. This allows our test framework to report results without the need to use a microcontroller peripheral such as a USART.

The basic approach for each layer is outlined below:

**Component layer**

Decoupling the software components from the hardware and application layers allows each component to be developed and tested in its own software project. If the hardware and CMSIS drivers are available, we can develop a fully functioning component. If we are early in the development cycle the driver layer can be stubbed to provide dummy data and as the driver has a common interface any work here can be reused on later projects.

Either approach allows us to build a test suite as the component is developed and even adopt a full Test Driven Development (TDD) process if you are disciplined enough. This coding and testing approach has the advantage of not only detecting bugs early in the project, but it also has an unexpected side effect. When you are writing test cases alongside developing code you tend to constantly review and improve the existing code. Trust me, this approach is highly recommended!

**Application Layer**

We can use similar testing techniques for the application layer. Here we can create each of our threads and build up its application code by first stubbing out calls to the components and enclosing each thread in a test framework. The full version of each component can then be integrated as they become available.

### Hardware Layer

Part of the CMSIS Driver specification includes validation test suite which can be used to prove the driver software and the microcontroller hardware. This is ready to go and can be used on any microcontroller and hardware. The test suite is easy to expand to support additional custom drivers and new tests. Any extensions to the test suite would be done once and can be reused on any new project. Once we have a fully populated test suite any CMSIS Drivers for new microcontrollers can be developed within the test framework.

## Reuse

For all reusable software components, it is worth spending some additional effort to increase their ease of use by using features within the CMSIS Pack standard. Two useful features here are configuration wizards and software packs.

### Configuration wizards

It is possible to create configuration wizards by placing tags in comments within header and source files. This is quick and easy to do while creating visually intuitive code.

```
//   <o>Select USART Peripheral
//            <0=> USART 0
//            <1=> USART 1
//            <2=> USART 2
//            <3=> USART 3
//            <4=> USART 4
#ifndef GPS_USART_PERIF
#define GPS_USART_PERIF   0
#endif
```


*Figure 5: Configuration wizard example*

### Software Packs

The CMSIS Software Pack specification allows you bundle software components and their collateral into reusable software packs. A software pack is a compressed file with an additional XML file that describes the contents. A range of in-house software packs can be stored on a local server providing version and configuration control. Once installed into a supporting toolchain each component may be added to a project through a platform manager. When installed the pack XML file provides information about dependencies which can be used by the toolchain platform manager.

So, for example a networking stack would require a CMSIS Ethernet Driver and an RTOS to be part of the project. Over time the creation of reusable software components and software packs will lead to big gains in developer productivity.

## Conclusion

The full specification for this system is still being developed and will be released early next year but hopefully this article will give you a taste of what's involved. If you are interested in more details as they become available, then please drop me an email on tmartin@hitex.co.uk.

## References

| Reference | URL |
|---|---|
| CMSIS | https://www.keil.com/pack/doc/CMSIS/General/html/index.html |
| MCU Boot | https://juullabs-oss.github.io/mcuboot/ |
| Platform Security Architecture | https://www.arm.com/why-arm/architecture/platform-security-architecture |
| Unity test framework | http://www.throwtheswitch.org/unity |

*Table 3: References*

hitex

EMBEDDED TOOLS & SOLUTIONS

**Hitex (UK) Ltd**

Millburn Hill Road

University of Warwick Science Park

Coventry CV4 7HS

Phone: +44 24 7669 2066

Fax: +44 24 7669 2131

Email: sales@hitex.co.uk