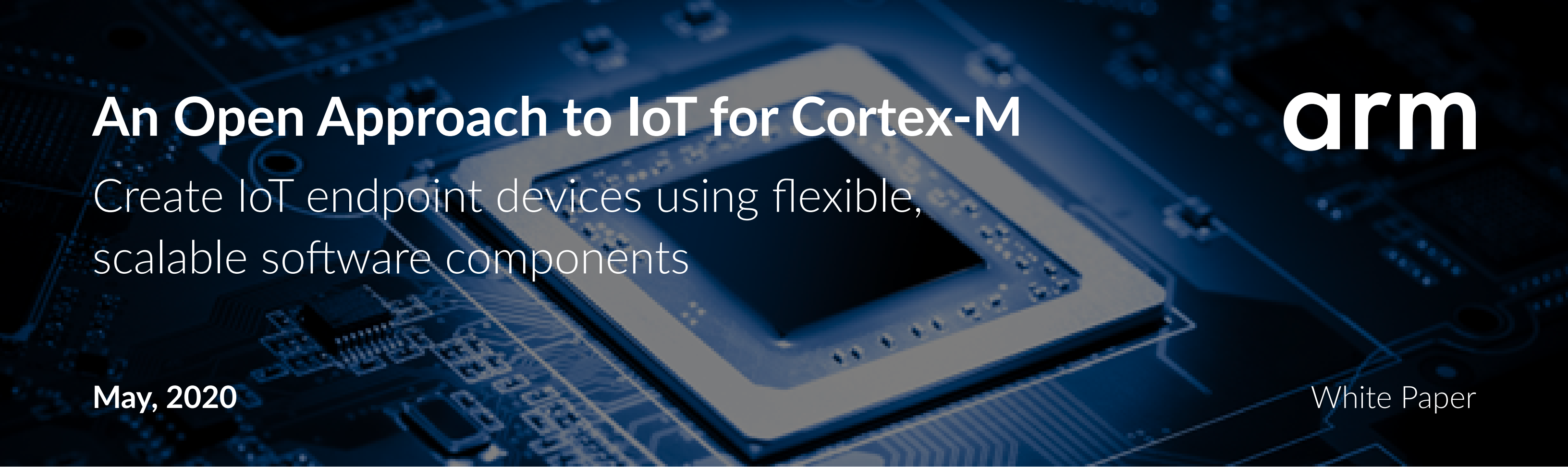


An Open Approach to IoT for Cortex-M

Create IoT endpoint devices using flexible, scalable software components

May, 2020

White Paper

Overview

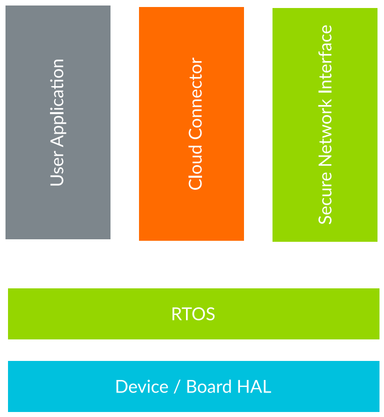
Arm Cortex-M processors have been shipped in more than 47 billion chips for a vast range of applications, from industrial sensors to wearables. This growth has exploded more so in the last few years due to the significant rise in connected products for diverse markets. AWS IoT provides broad and deep functionality, spanning the endpoint to the cloud, so customers can build IoT solutions for virtually any use case across a wide range of devices. With designers of IoT applications under extraordinary pressure to build innovative solutions quickly, affordably, and satisfy many design requirements, how can the IoT continue to scale across a growing number of use cases?
This whitepaper showcases a simple path to developing secure Cortex-M based IoT devices with Arm and AWS, and how together, the collaboration provides choice and scalability for IoT developers.

Introduction

The common market prediction is that by 2035, one trillion connected IoT devices are deployed world-wide. The applications span across all industries, for example: consumer, industrial, metering, medical, agriculture to name just a few. A trillion is a massive number and it breaks down for example to 10,000 different designs that are deploy in 10 million units over a period of 10 years. Products that ship in very high volume are all cost sensitive and therefore many IoT endpoint devices will be based on cost effective microcontrollers that are available already in lots of variants.

Most designs will be started with low-cost evaluation boards that are today common in the microcontroller industry utilizing reference designs that are based on open source software. For rapid IoT device development, scaling of these reference designs to cost optimized and resource constrained high-volume production is critical. An effective, flexible, easy-to-use software development process is paramount, as embedded engineers will need to optimize, extend, and validate complex software stacks that implement the overall device functionality. Fig. 1 is a simplified view of the various software components in an IoT application.

Fig. 1: IoT application components - simplified



In Fig. 1 the components represent:

- Device / Board HAL: hardware abstraction to the processor and the peripherals. It also contains the overall device configuration that is specific to the system design.
- RTOS: real-time operating system that is used for thread scheduling and resource management.
- Secure Network Interface: IP communication stack that implements an encrypted connection to the internet. It may use different physical interfaces, such as wired Ethernet, WiFi, or low-power radios
- Cloud Connector: protocol stack that interfaces to the cloud solution provider. For example, AWS provides an SDK for connecting to AWS IoT that is tailored for embedded systems [7].
- User Application: implements the bespoke functionality of the embedded system.

All these various software components are created and maintained by different vendors with teams on various geographic locations. The challenges for the embedded software engineers are:

- Migrate a software reference design from evaluations kit to bespoke production hardware
- Optimize the resource usage to minimize system cost
- Manage the various software components over the product lifecycle

To simplify the overall development process and increase the re-use of standardized software, Arm has developed the Cortex Microcontroller Software Interface Standard (CMSIS) [1]. The CMSIS is a vendor-independent hardware abstraction layer for Arm Cortex-M and Cortex-A processors and covers generic tool interfaces.

CMSIS-Pack System

One component of CMSIS is the CMSIS-Pack system that provides a delivery mechanism for device support and software components. The CMSIS-Pack system supports today more than 6,000 different microcontrollers and provides ways to manage software components from different sources. The CMSIS-Pack system is already implemented in the IDE (integrated development environment) of several leading development toolkits such as:

- Arm Development Studio
- Arm Keil MDK
- IAR Embedded Workbench for Arm
- CMSIS Eclipse Plug-In (open-source project) as basis for several other implementations.

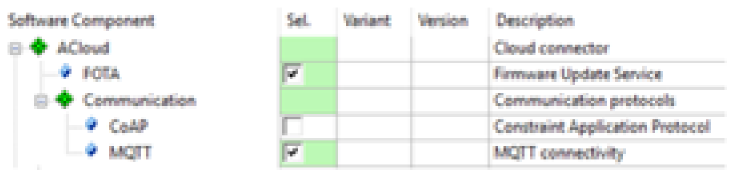
The ability to distribute software components makes CMSIS-Packs attractive for software platform providers. The following will focus on the benefits of CMSIS specifically for software components.

Software Components

The CMSIS-Pack management system, that is integrated in various IDEs, makes it easy to combine software components that are developed independently and even by different vendors [3]. A CMSIS-Pack can include a collection of software components provided as source code or library with header files, configuration files and related documentation.

Software Component Selection in IDEs

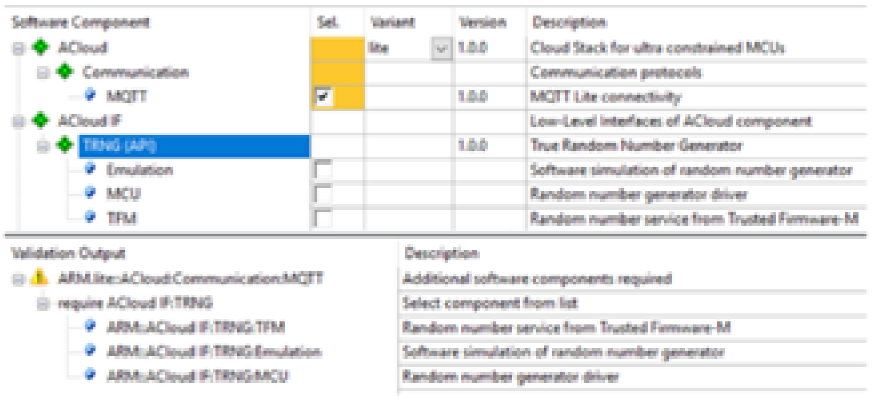
Fig. 2 shows how software components are exposed to the user in the manage Run-Time Environment (RTE) dialog of an IDE. RTE allows to select the software components of your system.



Software Component	Sel.	Variant	Version	Description
ACloud				Cloud connector
FOTA	✓			Firmware Update Service
Communication				Communication protocols
CoAP	☐			Constraint Application Protocol
MQTT	✓			MQTT connectivity

Fig. 2: Software component selection in IDEs

The software components are described in Pack Description (*.pdsc) Format (XML file) that is specified in the CMSIS-Pack documentation [2]. The *.pdsc file refers the files that represent a software component. Depending on the *.pdsc information, this dialog may offer variants of a software component and show version information.



Software Component	Sel.	Variant	Version	Description
ACloud		lite	1.0.0	Cloud Stack for ultra constrained MCUs
Communication				Communication protocols
MQTT	✓		1.0.0	MQTT Lite connectivity
ACloud IF				Low-Level Interfaces of ACloud component
TRNG (API)			1.0.0	True Random Number Generator
Emulation	☐			Software simulation of random number generator
MCU	☐			Random number generator driver
TFM	☐			Random number service from Trusted Firmware-M

Validation Output	Description
ARM.lite:ACloud:Communication:MQTT	Additional software components required
require ACloud IF:TRNG	Select component from list
ARM::ACloud IF:TRNG:TFM	Random number service from Trusted Firmware-M
ARM::ACloud IF:TRNG:Emulation	Software simulation of random number generator
ARM::ACloud IF:TRNG:MCU	Random number generator driver

Fig. 3: Software component dependencies

Software components can have dependencies to other software components. Fig 3. shows the user view of these dependencies where the component MQTT requires a low-level interface to the TRNG (True Random Number Generator) component. In the example above, three different implementations are available for this software component and the user should decide which implementation to use in the actual system. If there is just one software component available, the RTE system can automatically pick this implementation.

Software components and API Interfaces

Fig 4. shows the different parts of a software component which consists of header files, source or library files. Software components also typically provide interfaces that are represented in header files to other parts of the software. There are two different types of interfaces:

- Component API that allows to use the functionality of the software component itself.
- Driver API to interface with a HAL or software components that is called by the software component.

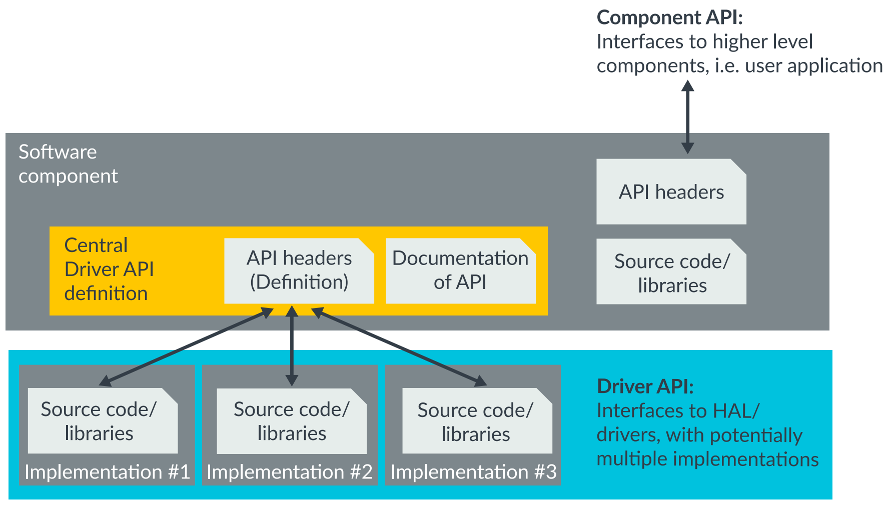


Fig. 4: API definition for software components

A common problem when providing a software component is that driver API headers evolve over time. The apis element in the *.pdsc file allows a software component to define an API to lower level software or hardware drivers. It shares header file and documentation of an API interface across multiple other software components which ensures consistency. This central header file allows to extend an API over time and the version information of the API definition can be used by various implementations to guarantee consistency.

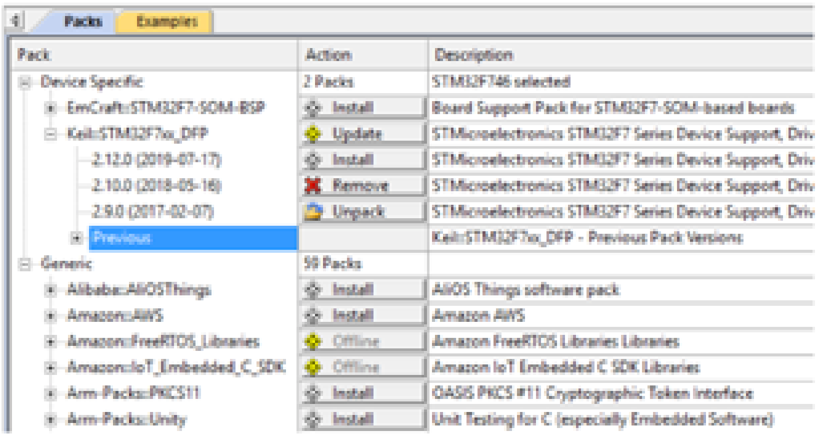
Software Packs

Software packs are a collection of one or more software components and simplify the re-use, management, and distribution. While the structure of the software packs is flexible, it is common practice that related software components are published in a single pack.

There are multiple ways to distribute a software pack that range from local installation to wide-web distribution using a Pack Index Service. A Pack Index Service makes packs available widely to development tools or web portals [8]. This makes it easy for software vendors to provide a new software packs or update existing software packs.

Software packs that are distributed via the Pack Index Service are directly available in the IDE of the development tools (Fig. 5). Embedded developers can therefore pick and choose the relevant components from this software catalogue.

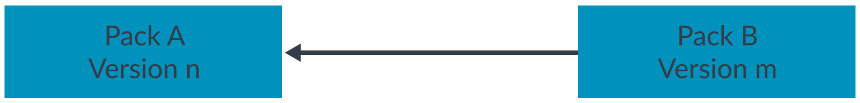
Fig. 5: Software Pack Installer in the IDE



Pack	Action	Description
Device Specific	2 Packs	STM32F746 selected
EmCraft::STM32F7-SOM-BSP	Install	Board Support Pack for STM32F7-SOM-based boards
Keil::STM32F7xx_DFP	Update	STMicroelectronics STM32F7 Series Device Support, Driv
2.12.0 (2019-07-17)	Install	STMicroelectronics STM32F7 Series Device Support, Driv
2.10.0 (2018-05-16)	Remove	STMicroelectronics STM32F7 Series Device Support, Driv
2.9.0 (2017-02-07)	Unpack	STMicroelectronics STM32F7 Series Device Support, Driv
Previous		Keil::STM32F7xx_DFP - Previous Pack Versions
Generic	59 Packs	
Alibaba::AliOSThings	Install	AliOS Things software pack
Amazon::AWS	Install	Amazon AWS
Amazon::FreeRTOS_Libraries	Offline	Amazon FreeRTOS Libraries Libraries
Amazon::IoT_Embedded_C_SDK	Offline	Amazon IoT Embedded C SDK Libraries
Arm-Packs::PKCS11	Install	OASIS PKCS #11 Cryptographic Token Interface
Arm-Packs::Unity	Install	Unit Testing for C (especially Embedded Software)

Software packs can also request the installation of other additional packs (Fig. 6). For example, Pack B can request that a specific version of Pack A is installed. The pack management ensures that this pack is pulled. Software Pack Installer can request therefore additional pack installations.

Fig. 6: Software Packs can request related Software Packs



This concept makes it easy to use other software packs as a basis for more complex software. For example:

- Board support pack requests the installation of related device pack.
- Cloud connector requests the installation of a Crypto service pack.

Example programs that are based on software packs trigger the installation of the related software packs which reduces the overall size of the examples. It makes it also easier to ensure that example projects are using up-to-date software as explained in the following section.

Project Life Cycle Management

CMSIS uses semantic versioning across software packs, software components and related configuration files [9]. A version number contains values for MAJOR.MINOR.PATCH releases (for example version “2.3.12”) which indicates the compatibility of the software:

- A change of MAJOR version number is an incompatible change.
- A change of MINOR indicates added functionality in a backwards compatible manner.
- A change of PATCH are backwards compatible bug fixes.

Semantic versioning also provides ways to indicated pre-releases or the release quality of a software (such as Alpha, Beta, etc.). This is important during the development cycle of a software component itself.

Once the software component is used in an application, the RTE management in the IDE keeps track of the initial pack that was used to install the software. Using the software pack management in the IDE (Fig. 5) makes it easy to verify that your application is based on current releases and gives you direct access to potential updates.

During the software development cycle of a product, this provides flexibility to manage and control the software that is used. For example:

- In the Concept phase of a new project, the features of the most recent software components should be used. The latest versions of the available software packs can be downloaded with the Software Pack Installer (Fig. 5).
- During the Development phase only the latest version of the chosen or installed software packs should be used. This is the default selection of most IDEs.
- Once the software is in the Release phase only specific versions of software components should be used. The RTE management in the IDE allows you to fix the version of a software pack in a project (Fig. 7).

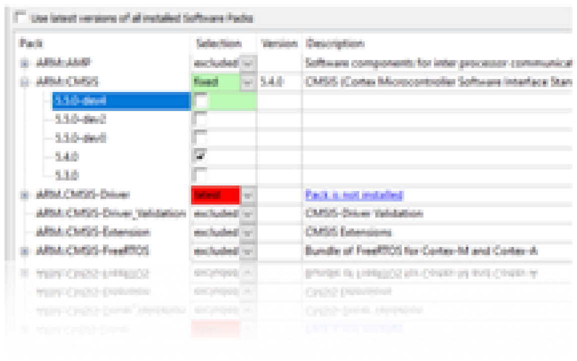


Fig. 7: Select specific pack versions in the IDE

Use latest versions of all installed Software Packs

Pack	Selection	Version	Description
ARM::AMP	excluded		Software components for inter processor communicat
ARM::CMSIS	fixed	5.4.0	CMSIS (Cortex Microcontroller Software Interface Stan
5.5.0-dev4			
5.5.0-dev2			
5.5.0-dev0			
5.4.0	✓		
5.3.0			
ARM::CMSIS-Driver	latest		Pack is not installed
ARM::CMSIS-Driver_Validation	excluded		CMSIS-Driver Validation
ARM::CMSIS-Extension	excluded		CMSIS Extensions
ARM::CMSIS-FreeRTOS	excluded		Bundle of FreeRTOS for Cortex-M and Cortex-A

The semantic versioning used throughout all parts of the software components supports migration to a newer version of a specific software component. Source files are automatically replaced and configuration files that have project specific settings are tracked for version changes. Some IDEs even simplify the migration of configuration settings into newer versions of configuration files.

Security

Security is a mandatory but constantly changing and evolving requirement in connected IoT systems, and many microcontrollers offer specific features. With regulations on the horizon and new threats being identified, IoT devices need a strategy to protect against security threats and software developers need a standardized way to access security features.

The Platform Security Architecture (PSA) offers a framework for securing connected devices [4]. It provides a step-by-step guide to build in the right level of device security, reduce risk around data reliability and allow businesses to innovate on new ideas to reap the benefits of digital transformation.

For Cortex-M based microcontrollers, the Trusted Firmware-M (TF-M) provides a high-quality open source reference implementation of secure software mandated by PSA. TF-M forms the foundations of the Secure Processing Environment (SPE) of microcontrollers providing Secure Boot, Isolation from untrusted Software and set of Secure Services that can be used by Applications (Fig. 8).

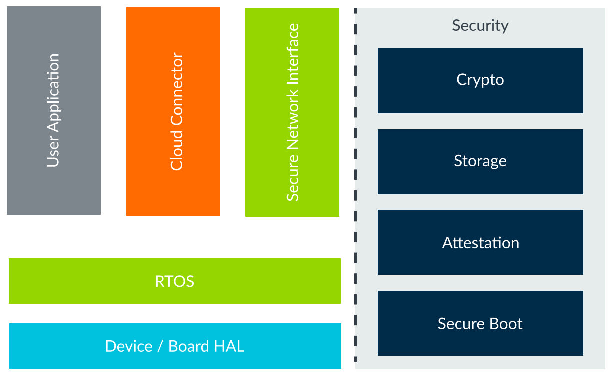


Fig. 8: Security services provided by TF-M

TF-M may be implemented in various ways, for example using an off-chip Secure Element or multi-core processor systems. For cost-sensitive IoT applications, the Cortex-M23 and Cortex-M33 processors implement TrustZone technology for Armv8-M, which is a system-wide approach to security with hardware-enforced isolation built into the CPU.

TrustZone enabled microcontrollers are today available from several silicon vendors. On Cortex-M23 or Cortex-M33 processors, the TF-M can be pre-configured for a specific microcontroller using also the device specific security features. Such a pre-configured version provides a standardized API to this security features and matches at the same time the requirements for many IoT endpoint devices. As TF-M will be also provided as software pack, the creation of IoT devices will be further simplified.

IoT Clients

An IoT client is a software interface which runs in the IoT endpoint device and establishes the connection to a cloud service. It represents the software components that are required for the Cloud Connector in Fig. 1 (or other places in this document). Many cloud service providers offer open-source software that implements an IoT client for an embedded system. Arm adopted these IoT clients and delivers it as software pack for using it in Cortex-M based IoT endpoint devices.

The repositories for these public software packs are available on GitHub (https://github.com/MDK-Packs/) and contain IoT clients for the following cloud service providers:

- Amazon AWS IoT
- Google Cloud IoT
- IBM Watson IoT
- Microsoft Azure IoT Hub
- Paho MQTT (Eclipse) – generic not related

These IoT clients directly connect to the Secure Network Interface that is shown in Fig. 1. As IoT endpoint devices have diverse connectivity requirements, embedded software developers will need to cope with different interface technologies to a network. Fig. 9 shows three different ways to implement the Secure Network Interface.

- WiFi is the most popular interface of IoT endpoint devices. WiFi connectivity for microcontrollers is frequently implemented using WiFi chipsets that connect via SPI or UART to the microcontroller. The CMSIS-Driver framework defines both the WiFi interface itself and the related drivers for UART or SPI. Today, several popular WiFi chipsets are already supported with ready-to-use software packs [12].
- Wired Ethernet interfaces are frequently used in industrial applications. The TCP/IP stack that is part of the Keil MDK Middleware can be used to connect to wired Ethernet.
- A popular open-source TCP/IP stack is LwIP. Also, this software component is available as software pack and interfaces to the CMSIS-Driver for Ethernet MAC/PHY.

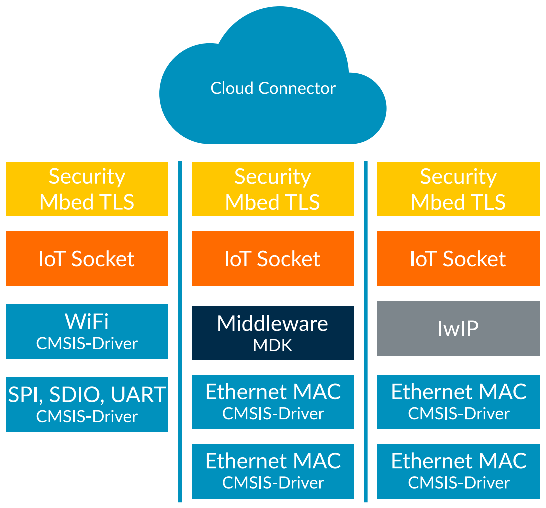


Fig. 9: Various ways to implement a "Secure Network Interface"

CMSIS-Driver

The CMSIS-Driver provide a consistent interface for middleware to physical device hardware. Drivers that are specific to a microcontroller device, are frequently part of the Device Family Pack.

Arm also offers software pack with a set of generic CMSIS-Driver implementations that support for example Ethernet MAC/PHY, Flash, or WiFi chipsets. The source code of these drivers is open-source and available on GitHub [13].

To support the validation of the CMSIS-Driver, Arm offers a validation suite. This suite helps the developers of CMSIS-Drivers to ensure consistency, for example a new driver for a WiFi chipset can be verified. But it helps also the system developer of an IoT endpoint device to verify the configuration and setup of the CMSIS-Driver interface.

An application note explains how to use these IoT Clients in software projects [11]. Using these ready-made software components allows software developers to create flexible IoT endpoint devices based on a wide portfolio of Cortex-M microcontroller devices. Future versions of these IoT Clients will utilize the TF-M that is described in the previous section "Security".

IoT Layer Projects

For scaling example projects on a wide range of evaluation boards, Arm provides a layer support for projects that are created using the CMSIS-Pack system. Layers are a set of pre-configured software components. Fig. 10 shows how these layers can be applied to IoT endpoint device examples:

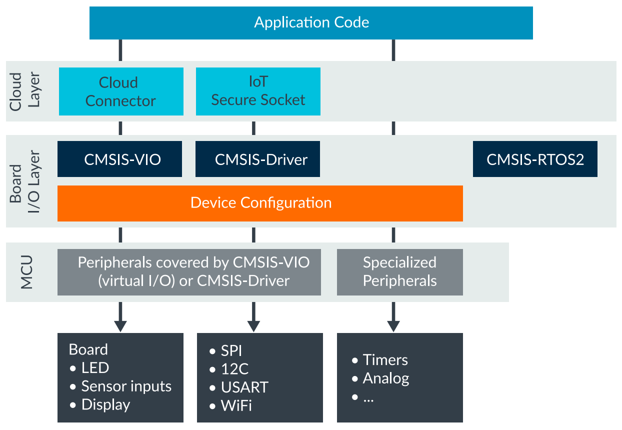


Fig. 10: Project layer configurations for evaluation boards

- I/O Layer represents the software components that connect to the physical peripherals of an evaluation board.
- Cloud Layer is the pre-configured interface to a specific cloud service provider.

Combining various I/O Layers with multiple Cloud Layers and related example code will allow to deploy reference applications to many evaluation boards at scale.

For testing these examples, a Continuous Integration system is required, and this is supported with the new CMSIS-Build and CMSIS-Test component. CMSIS-Build is a MAKE-based system for verifying software against new software pack versions.

CMSIS-Test provides an interface to typical peripherals of evaluation boards, for example LED, sensor inputs, display, or push buttons. However, it also gives access to these peripherals via defined memory locations which may be used in test systems to stimulate the application program.

The CMSIS-Test component also allows to disconnect example or application code from physical board interfaces. This is useful during bring-up where parts of the software are developed using evaluation boards. When such a software project is ported to production hardware that does not provide board interfaces, the physical interface can be disabled. This helps also to migrate example code from evaluation boards to bespoke production hardware and ensure that fundamental functionality works consistently.

Summary

Arm takes a holistic view to the development cycle of IoT endpoint devices using Cortex-M microcontrollers. CMSIS ensures the software productivity that is required to create robust, cost-effective applications. Many ready-to-use software building blocks are provided to utilize the CMSIS-Pack concept.

PSA offers the framework for securing connected devices from ground-up. The TF-M open-source software project provides the security services that are required in IoT endpoint devices. TrustZone enabled Cortex-M microcontrollers deliver the processing power for cost-sensitive mass production.

The development process described in this whitepaper is supported by various software tools ranging from open source to commercial professional editions. Engineers can choose the tooling for stringent testing that is mandatory in the embedded industry.

References

[1] Arm, “CMSIS Cortex Microcontroller Software Interface Standard”

www.arm.com/cmsis

[2] Arm, “CMSIS” documentation

www.arm-software.github.io/CMSIS_5/General/html/index.html

[3] Arm, C. Seidl, “What are CMSIS Software Components”

www.community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/what-are-cmsis-software-components

[4] Arm, “TrustZone for Cortex-M”

www.developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m

[5] Arm, “Platform Security Architecture”

www.arm.com/psa

[6] Trusted Firmware-M (TF-M)

www.trustedfirmware.org/

[7] AWS “SDK for connecting to AWS IoT from a device using embedded C”

www.github.com/aws/aws-iot-device-sdk-embedded-C

[8] Arm, “Pack Index Service for Keil MDK”

www.keil.com/pack

[9] Semantic Versioning 2.0.0

www.semver.org/

[10] Arm, “IoT Clients”

www.keil.com/iot

[11] Arm, “Application Note 312: Connecting to the Cloud”

www.keil.com/appnotes/docs/apnt_312.asp

[12] Arm, “CMSIS-Driver for WiFi”

www.www2.keil.com/mdk5/cmsis/driver/

[13] Arm, “Generic CMSIS-Driver for Cortex-M microcontrollers”

www.github.com/arm-software/CMSIS-Driver