SECURING MODERN-DAY DEVICES WITH EMBEDDED VIRTUALIZATION AND ARM TRUSTZONE TECHNOLOGY

FELIX BAUM
VIRTUALIZATION SOLUTIONS PRODUCT MANAGER

Mentor Graphics®

E M B E D D E D   S O F T W A R E

W H I T E P A P E R

www.mentor.com

INTRODUCTION

Before spending too much time describing the technical aspects of how to secure an embedded device with virtualization and ARM® TrustZone® technology, let's take a step back and review some of the different types of embedded devices that exist today.

What does it even mean to secure an embedded device? It all depends on the type of device you want to secure. You must always take into consideration the environment in which the device will be used, potential security claims made on the device, and in many cases, it's essential to examine the processes used to develop the device.

A BASIC LOOK AT DEVICE TYPES

Let's first take a look at a simple thermostat (Figure 1). This single-purpose, standalone device has no Internet connection and limited user interaction. Although mostly mechanical, one can potentially envision this type of thermostat with a small LCD screen and a couple of buttons to control operation. The user turns it on and off. Sets the temperature for it to maintain. And that's about it. When securing this type of device, we should probably worry more about physical abuse – someone literally taking a hammer and smashing it – rather than any type of security breach. If this type of device functioned more than just a mechanical unit, we might have to worry about malicious attempts to replace the EEPROM inside. But the ways one can attack this particular device, the "area of attack," are quite limited because of its simplicity.

Figure 1: *Basic standalone device.*

If we consider a slightly more sophisticated thermostat (Figure 2), the device still has limited capability, but it has an embedded processor that allows the user to program it for more advanced operational use. It might even be networked to allow remote access to log in and check home comfort levels. Units like these do not allow remote control of the parameters and offer a limited, well-defined user interface (UI). As you can see from the UI, this device is a bit more complex, thus the area of attack is larger but still well contained and easily managed. Securing this type of device is not as simple as the standalone unit, but still easy to do.

Figure 2: *Networked device with processor and simplified UI.*

Keeping with the theme of thermostats for the home, Figure 3 represents a "thermostat on steroids," a complete home automation unit. This unit has a sophisticated graphical UI and consolidates many of its features. It is fully networked allowing multiple users to access its touch-screen LCD panel. Users can also log in remotely. It might even be sophisticated enough to retrieve the latest forecast from the local weather service, or find the best possible electrical rates from the local utility company. This particular device is managed – it needs to be configured, monitored, and updated. It's not as simple to secure this type of device compared to the first two examples. One has to address a plethora of threats, not to mention the exposure and area of attack is much larger.

Figure 3: *Fully networked, automated device.*

Now that we have a better understanding of the effort needed to secure an embedded device, let's tackle the question that keeps management awake at night: How much security is enough security? The basic rule is to consider optimal security at the most affordable cost as it aligns with the value or importance of the data you're trying to secure. If the value placed on the loss of data and/or device is $1, there is little need to spend $10 to make the device more secure. Software developers also must understand the holistic approach to security. It's critical that developers realize that security should be addressed throughout all stages of embedded device development (Figure 4). Even when it comes to destruction or disposal of a device.

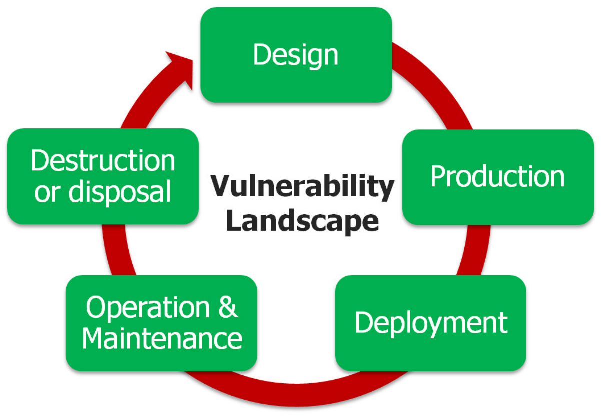

Figure 4: *Security needs to be addressed in all stages of device development.*

PROTECTING DATA IN ALL STAGES

When addressing how to secure an embedded device, it's important to first take into account the surface area vulnerable to attacks, the "area of attack" which has already been discussed. Second, it's critical to design and develop a device that is robust and secure by layering various secure capabilities. Finally, it's important to understand most of the threats today target data not for the sake of data, but for the ability to manipulate the data.

A common scenario might be an attack on an algorithm that affects the operation of the very system it depends on for operation. Examples might include banking information in point-of-sale devices, patient information in medical and IT infrastructure devices, or even the parameters that govern how a device functions within the automobile. When it comes to protecting data, one needs to make sure to protect data in all stages: **data at rest**, **data in use**, and of course, **data in transit**.

Data at rest is best described as when the device is powered down. Key points to consider in this stage include:

- Where is the bootable image stored?
- Are there anti-tampering methods used to inform the device if it's being tampered with? Are there means to prevent it from booting into a vulnerable state?
- Have the executables been encrypted, or could anyone who gains access remove EEPROM, dump the memory, or attempt to reverse engineer the application?
- Has the data been stored in a secure database?
- Will the device use trusted boot process to come up from the cold boot?

The concept of **data at use** applies when a device is booted and operating normally and data is being generated and processed. Some of the questions to ask include:

- Have obfuscation methodologies been used for sensitive data?

- Is the device executing in a validated state? Has a chain of trust been established?
- How frequent is the random generation algorithms being used?
- Does the software rely on hardware capabilities to enforce protection such as MMU-based protection methods or/and ARM TrustZone
- Regarding devices with multiple users are user privileges being enforced and are secure file systems that support the concept of multiple users with various privileges being deployed?

Finally, there is **data in transit**, which is used to describe data leaving the device. A good design should address these concerns:

- How is data being protected if it is hijacked?
- Are encryption or tunneling protocols being used?
- Have firewalls been deployed and what are the strategies for denial-of-service attacks?

The considerations above are just some of the points to keep in mind when designing a robust and secure embedded system.

BUILDING SECURITY INTO YOUR DEVICE

The design of any security architecture and determining what state the data and information is in conventionally relies on two basic concepts: first; the principle of least privilege to the data, and second; the partitioning of the system into protected compartments in order to isolate the information.

At the end of the day, when protecting data at various stages two considerations become clear:

- To make an **embedded system secure**, you need to have a specialized, secure, and protected memory location in which to store the keys for the secure boot, passwords for the banking and social media, encryption algorithms for secure communications, and DRM for the audio and video content.

- To make an **embedded system robust,** you need to control the level of access to the critical functions and capabilities, engineer a system via “secure by design” methodology.

Although various architectures take different approaches when dealing with security, due to the overwhelming trend in the embedded industry to adopt ARM-based devices, we will discuss how ARM TrustZone technology helps protect against security threats. ARM offers a compartment security model that provides increasing levels of security through a combination of reducing the attack surface and increasing isolation.

A system designer that selects from the ARM® Cortex®-A15 class of devices will typically use a combination of the following to provide an appropriate level of protection of both user and system assets:

NORMAL WORLD (WITH USER MODE AND SYSTEM MODE)

Running processes and/or applications are isolated from one another by the operating system and the Memory Management Unit (MMU). Each executing process has its own addressable memory isolated from other processes, along with a set of capabilities and permissions that are administered by the operating system kernel executing with system level privilege. This is the normal operating state for application software, and together with the operating system, it is often referred to as the “normal world,” in contrast to the “trusted world.”

TRUSTED WORLD

TrustZone security extensions allow the system to be physically partitioned into the secure and non-secure components. This provides further isolation of assets and can be used to ensure that the software operating within the normal operating system cannot directly access secure memory or secure peripherals.

HYPERVISOR MODE

The hypervisor allows multiple instances of the same or different operating systems to execute on the same processor as a virtual machine. Each virtual machine can be isolated, and through use of a system MMU, other bus masters can also be virtualized. This separation can be used to protect and secure resources and assets in one virtual machine from other virtual machines.

ARM TRUSTZONE TECHNOLOGY

ARM TrustZone architecture provides a solution that is able to *carve out* or segregate a hardware subset of the full system-on-chip (SoC). It does this by defining processors, peripherals, memory addresses, and even areas of L2 cache to run as "secure" or "non-secure" hardware. A SoC that utilizes ARM TrustZone technology has the ability to dynamically, with only a few clock cycles delay, expose the full SoC to secure software, or to expose a subset of that SoC to normal software (Figure 5).

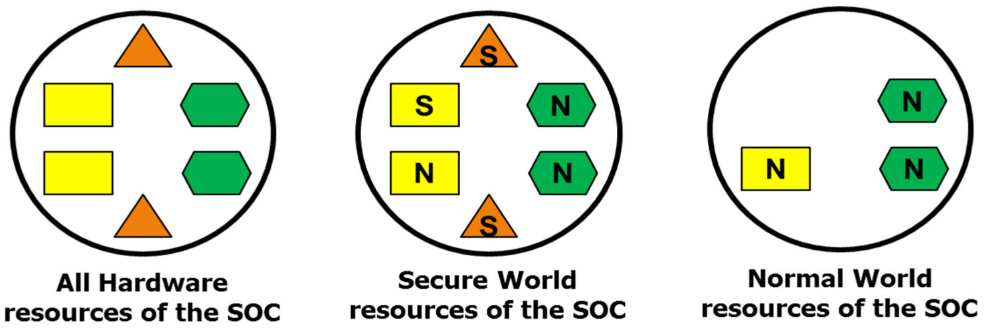
S
S
N
N
N
S
All Hardware resources of the SOC
Secure World resources of the SOC
N
N
N
Normal World resources of the SOC

Figure 5: *ARM's TrustZone technology allows developers to designate functions within a SoC as "Secure World" or "Normal World."*

The normal world (or non-secure world) created and enforced by TrustZone is typically a defined hardware subset of the SoC. TrustZone ensures that a non-secure processor can access only non-secure resources and receive only non-secure interrupts. For example, a normal world hardware subset might include the UART, Ethernet, and USB interface, but exclude CAN controller access. The CAN might instead be dedicated to the secure world where a separate RTOS or application runs for the sole purpose of managing CAN traffic, independent of the normal world software stack.

Software that runs in the normal world is assumed to be flawed from a safety and security perspective. That is, normal world software is expected to contain bugs, exploits, hacks, faults, or irregularities that could expose sensitive information or functions. It is these assumptions that drive the value of TrustZone's ability to isolate the adjacent processing area (secure world) where sensitive data storage or functions are managed from the perceived flawed normal world software.

Unlike the hardware subset in which normal world software runs, software running within the secure world has complete access to all of the SoC hardware. Thus, from the perspective of the secure software's execution, the system looks and acts nearly identical to what would be seen on a processor that does not have TrustZone. This means that secure software has access to all resources associated with both the secure and normal worlds.

ARM TrustZone architecture contributes to the overall system security by preventing normal world software from accessing the secure world resources. It is important to understand that TrustZone does little to improve the safety or security of the software that runs in the secure world except to prevent unwanted secure world access by normal world software. Therefore, it is the developer who determines which software is "trusted" typically through rigorous development processes, testing, and certification.

Trusted software is the term chosen to represent software that runs within the secure world. Because a trusted system can only be created when the software is known to be error-free and without exploits (e.g. verified as trusted software) and because smaller works of code are significantly easier to certify for safety and security needs, trusted software will typically include only the minimum of functionality and device interfaces. If the trusted software doesn't include a TCP/IP stack, or a USB device, then these functional areas are implicitly secured in the absence of potentially exploitable software.

Due to this assumption around the fundamental integrity of the trusted software, ARM does not intend TrustZone to support a general purpose operating system (GPOS) within the secure world. The challenge of certifying any standard operating system (100K, or millions of lines of code) running as trusted software would simply be too great.

A Trusted Execution Environment (TEE) refers to software stack running within the secure world and the communications which allow that secure software to interact with the normal world software. TEE software (Figure 6) typically consists of a small microkernel and applications, and APIs which allow the secure software to communicate with the larger, user-centric software (e.g. Android, etc.). GlobalPlatform is a cross industry association that publishes specifications to facilitate the secure and interoperable deployment and management of embedded applications. One of these specifications defines a TEE offering what some might call "typical RTOS" APIs and functionality as well as additional capabilities and APIs that are well suited to the TEE use cases.

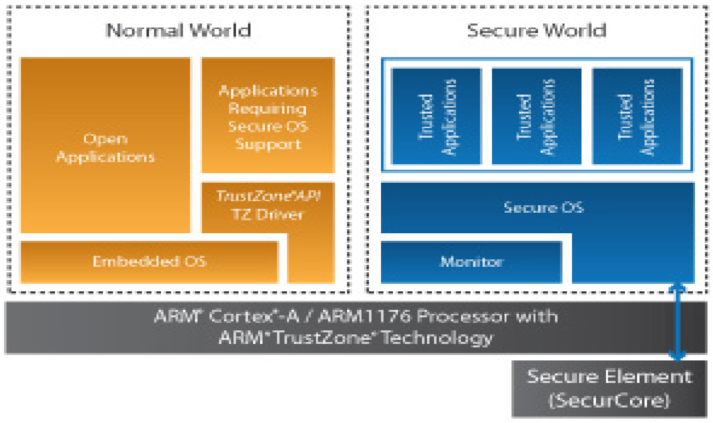

Figure 6: *Normal World and Secure World communications via an ARM Cortex device and ARM TrustZone technology.*

SECURING SoCs IN MULTICORE ARCHITECTURES

A single ARM-based core can execute normal world context or secure world context. But what happens in popular SoCs that are deployed with multiple cores? As seen in Figure 7, each one of these cores can potentially transition from executing normal world context to the secure world. As such, one can potentially run into situations where more than one core is accessing the same secure application, not only extending the surface of the attack, but potentially exposing the code to the nasty timing issues that are difficult to debug. It is best for developers to configure their device the way described in the Figure 8, where only one core is allowed to execute the secure world content.

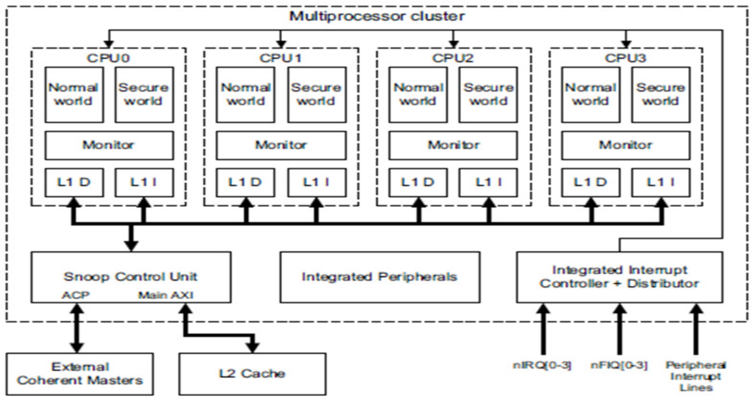

Figure 7: *Managing a multicore SoC.*

In this design, when an application running on any one core needs to launch a secure application, it would have to reach out to core 0, where the transition to the secure world happens. This would make multicore designs simpler and more robust.

Running one application in the normal world and hiding secure keys and algorithms in the secure world does make a lot of sense, but unfortunately, it's not very practical. A recent trend has developed in which silicon manufacturers are shipping more of the multicore parts. In many designs, more than one OS is used and this is where hypervisor mode and virtualization extensions in the SoC become useful. A more complete architecture would be built on top of the hypervisor which would incorporate support for ARM TrustZone technology. Designers would be able to partition application and peripherals between virtual machines while at the same time secure keys and proprietary algorithms inside the secure world.

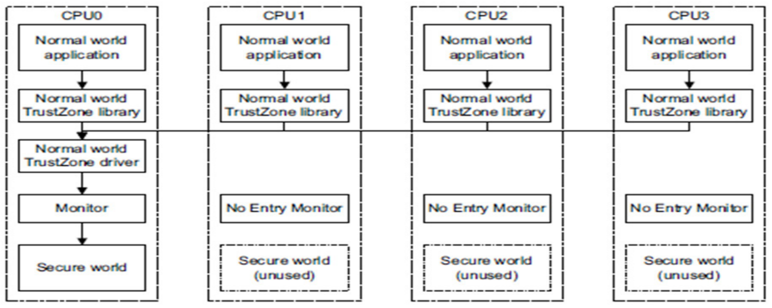

Figure 8: *Using one designated core (CPU0) to execute secure world content.*

To fit the real-time and performance demands of the embedded market, there are certain requirements embedded virtualization places on a hypervisor design. These include:

Type 1 (bare metal) hypervisor supporting hardware virtualization extensions with a minimal footprint and available in source code: Well, this one is obvious – the smaller the code, the less performance degradation one would expect when moving from running natively to virtualized. In addition, the better support for virtualization is present in the SoC, assuming the hypervisor supports them, the less code one has to write to support the same features in the software and the faster things run (aka hardware offload). The last point here is most embedded designs are custom. No two designs are alike even if they use the same underlying SoC, board, and guest OSes. The ability to tinker with source files to optimize the hypervisor performance and behavior to better fit a particular SoC/board/design is very important!

Support for TrustZone with robust and validated codebases: This one is rather straight forward as well. Since this paper is about security and reliability – what better way to ensure security and reliability if not by partitioning and separation of the code and data? Since the hypervisor code is at the foundation of the design, it has to be well tested and in some instances, even certified to an industry specific standard. This again, is one of the reasons why having a small code base is very important – less code to test equals cheaper validation and certification costs.

Multicore and multi-OS enabled with flexible scheduling: As previously mentioned, designers who are looking at multicore SoCs plan to deploy multiple guest operating systems. In some instances, they would like to have one guest OS run on two or more cores, and in others situations they would prefer to have two guest OSes share a single core. To support these various configurations flexible scheduling is a must. Note of caution: A flexible scheduler does not mean that one has to deploy a full-blown RTOS at the heart of the hypervisor to achieve this capability. In fact, as mentioned above, the smaller and simpler the scheduler, the less problematic and more robust it will be.

Extensive and flexible device model: This always seems to happen. Once you get things separated and running nicely in various virtual machines, your manager comes along and tells you that you need to make sure that these virtual machines can talk to each other and exchange data securely. Even worse is that right after that, he comes back demanding that you support sharing of the peripherals between various guest operating systems – Ethernet, serial, and graphics are just a few resources that are being brought up all the time. As such, a device model needs to support:

- Direct mapping of the peripherals where the guest OS owns the device. With this option, guest operating environments enjoy fastest, almost native performance.
- Shared mapping, where one virtual machine owns the device and shares it with other guest OSes. While hypervisor codebase stays small, guest OSes can all be reachable via one networking interface.
- Virtualized mapping, where hypervisor controls the device operation and driver stub and each guest OS performs as a second level driver. This option grows the footprint of the hypervisor but provides data assurance.
- Emulated model, where hypervisor owns one kind of device and emulates a different kind of device for the guest OSes. A driver within hypervisor controls the device operation and driver stub and each virtual machine performs as a second level driver.

Putting it all together, a design deployed for example on an ARM Cortex-A15 device would look like the diagram in Figure 9. In normal world we have the hypervisor executing with two virtual machines. Two instances of Linux® running virtual machines within the normal world space are set up to have kernel and drivers to execute in the kernel mode context, while user applications are mapped to the user mode. Trusted Execution Environment (TEE) and secure applications are mapped to the secure world space. There are a few TI OMAP5- and Jacinto6-based designs set up in this fashion.

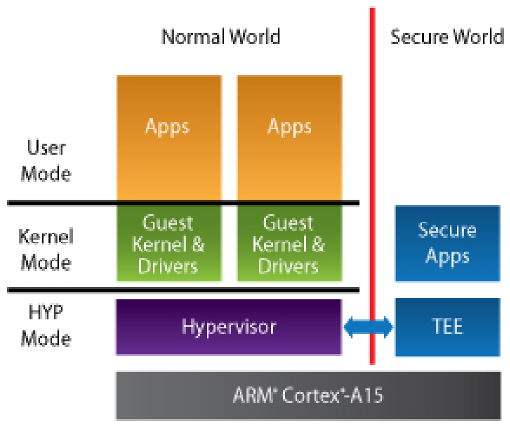

***Figure 9:** Incorporating hypervisor into ARM's Normal and Secure worlds.*

SECURE BOOT EXAMPLE

To provide a better understanding of how you can utilize this technology, let's look at the secure boot capability that many software designers are asked to implement these days. While the integrity of a software load may be tested using simple checksums, a checksum does not provide a way to detect the legitimacy of the software or its origin. While most designers would like to make sure the devices they design boot using authentic and authorized software, some of the designs I have engaged with require the additional ability to control what software may be loaded onto fielded devices. This may be necessary as an added safety precaution, such as in a situation where running unvalidated software might pose a serious danger to life or limb, or as a part of a revenue enforcement strategy, where a single piece of hardware may be upgraded to support additional features via software and the device vendor wishes to prevent the end-user from running illicitly modified software loads that enable features that haven't been legitimately paid for. In conjunction with other software, it may also be used for enforcing digital rights management.

Before we boot the device, we need to understand how we can make our software trusted, and one of the ways to accomplish this is to sign your code. Code signing is a process where an operating system can ensure that only trusted applications are loaded and run. This is done by requiring that the code modules include a digital signature from a trusted source. A digital signature is typically implemented using two cryptographic mechanisms: a hash algorithm and an asymmetrical encryption algorithm. Unlike a symmetrical algorithm, where the same key is used to encrypt and decrypt data, an asymmetrical algorithm uses one key for encryption and a different key for decryption. The encryption and decryption keys are mathematically related, but in a way that makes it difficult to determine one key knowing the other. These are often referred to as public/private key cryptosystems, as the encryption key is usually kept secret and only the decryption key is published. Common examples include RSA, DSA, and elliptic curve.

To digitally sign data or an application, a user first chooses a private key and then uses the RSA algorithm to generate a public key, thereby creating a public/private key pair. The private key is kept strictly confidential while the public key is openly published. The user then generates a hash of data or file to be signed, using the SHA256 algorithm. This yields a 32-byte block of hash information (often called the message digest). The user then encrypts the 32-byte hash with the private key. This encrypted hash becomes the signature, which is distributed along with the data.

When hardware is coming out of reset and being powered on, at each stage of the boot process, we need to answer the same two questions before proceeding to the next stage:

- **Have the data or application come from an authorized entity?**
- **Have the data or application been tampered with?**

Only if the answer to the first question is YES, and NO to the second question should we load this information into memory and execute the code. When the OS or the recipient of the data/application wants to validate the signature, the first step is to obtain a copy of the provider's public key, which was previously published. This can then be used to decrypt the hash that accompanied the data. The recipient can then independently calculate its own version of the SHA256 hash of the data and compare the result to the decrypted hash.

If the two hashes match, then this means two things: 1) the data has not been altered since it was signed; and 2) the data originated from the actual owner of the public key. Thus, in addition to verifying the data's integrity, we can also confirm that it came from a trusted source. As you can see we were able to kill two birds with one stone!

The code signing scheme requires that the master certificate and algorithms used be already present in the system and when using ARM Cortex A devices, it is imperative to utilize ARM TrustZone to protect them. It's important that the bootloader, the first secure OS stage, and master certificate storage be tamper-resistant. If only the master certificate is tamper-proof, but say, the bootrom storage is not, and if there's no prior validation of the bootrom image code, then an attacker could modify the bootrom image to disable the code signing feature entirely, in which case the security of the master certificate storage is irrelevant.

Using the keys and algorithms stored in the secure world, with hardware authenticating the bootloader, the bootloader validating the operating system, and the operating system verifying the application code and data we are able to establish a chain of trust rooted in the hardware. And even after all of the software is up and running and your device is operating properly, one might decide to download a new file (video or application) and not only authenticate this new file by validating its signature, but the other way around, the file itself might demand for the attestation of the underlying software stack as a way to prove that it has not been tampered with since the system was booted.

CONCLUSION

This paper demonstrates how ARM's TrustZone technology, together with a hypervisor offering embedded virtualization, can provide a strong, robust, and secure base for SoC designs that simply cannot be matched by a PC-based design. It shows how security can be used meaningfully for business and consumer applications.

Finally, this paper described how the combination of the TrustZone architecture, the security recommendations for SoC designs provided in TrustZone, and how a GlobalPlatform Trusted Execution Environment (TEE) can serve as a secure foundation that can protect against the main threat facing networked embedded devices when connected to the Internet of Things (IoT) world: software attacks.

Visit Mentor Embedded security for more information on how Mentor is enabling secure embedded systems, and for details on the Mentor Embedded Type 1 Hypervisor, visit the Mentor product page.

Author's biography

Felix Baum is working in the Product Management team of the Mentor Graphics Embedded Systems Division, overseeing the virtualization and multi-OS and multicore technologies. Felix has spent nearly 20 years in the embedded industry, both as an embedded developer and as a manager. During the last few years he led product marketing and product management efforts for various real-time operating system technologies and silicon architectures. Prior to that as a field applications engineer in the greater Los Angeles area, he consulted with customers on the development of highly optimized devices for a broad range of industries, including Aerospace, Networking, Industrial, Medical, Automotive and Consumer. Felix started his career at NASA's Jet Propulsion Laboratory at the California Institute of Technology, designing flight software for various spacecraft and managing a launch campaign for the GRACE mission. Felix holds a master's degree in Computer Science from the California State University at Northridge and a Master of Business Administration from the University of California at Los Angeles.

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

For the latest product information, call us or visit: **www.mentor.com**

©2015 Mentor Graphics Corporation, all rights reserved. This document contains information that is proprietary to Mentor Graphics Corporation and may be duplicated in whole or in part by the original recipient for internal business purposes only, provided that this entire notice appears in all copies. In accepting this document, the recipient agrees to make every reasonable effort to prevent unauthorized use of this information. All trademarks mentioned in this document are the trademarks of their respective owners.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Garden
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

MGC 01-15 TECH12360-w