WHITEPAPER

Twilio Microvisor—Architecture and Design Considerations for Modern IoT Infrastructure

CONTENTS

Introduction

The Internet of Things is at a critical juncture. Repeating past mistakes—specifically those related to security and maintenance—will slowly but surely erode confidence in IoT, and its ability to contribute to addressing the myriad of problems that both businesses and the environment face. It's time to change the way we work on embedded systems.

Based on a decade of hands-on experience with the development and maintenance of real world IoT solutions, this white paper explores the unique issues and challenges that connecting devices to the Internet brings. The paper is organized into two parts.

Part I addresses a broader audience, such as IoT product/project managers and CTOs. It lays out a typical device-side IoT architecture and describes the traditional approach of implementation. It details the associated challenges and develops an argument for a different approach, now made possible through new hardware advancements.

Part II addresses the experienced embedded engineer and explains Twilio's thinking with regard to how the above-mentioned challenges can be effectively addressed with a new architecture.

Authors: Joe Birr-Pixton, Toby Duckworth, Hugo Fiennes, Peter Hartley, Phil Michaelson-Yeates

Part I: The Challenges of Connecting Devices

Key Considerations for building an IoT device

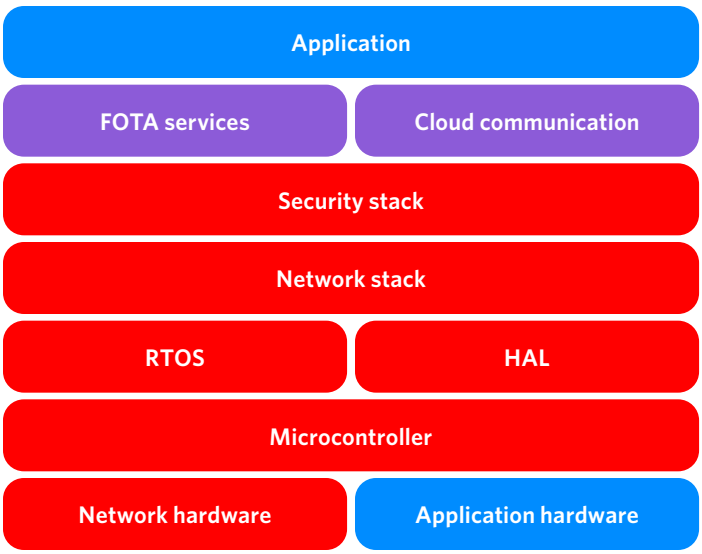
Connected devices vs. unconnected devices

Microcontrollers have been used in products for many decades, and have revolutionized product feature sets, reliability and performance over time. Moore's law has brought 16- and 32-bit processing to even the smallest and cheapest consumer products, and the availability of this memory and CPU power has enabled the use of real time operating systems (RTOS) where previously developers had to write "bare metal" code.

However, the transition from unconnected to connected products—in the context of IoT—has uncovered fundamental issues with how software is built for microcontrollers.

Connected device architecture

For IoT devices built around microcontrollers, a typical high level system architecture might look something like the diagram below. On the hardware side, there's a microcontroller connected to both networking hardware (Cellular/Wi-Fi/Ethernet) and to the application hardware—the sensors and actuators used by the IoT application.



In order to manage resources and tasks, an off-the-shelf RTOS is typically used. There are many choices here such as FreeRTOS, NuttX, ThreadX, and from a high level they all perform the same tasks—allocation of both memory and processor resources to different tasks within the system. To help decouple higher software layers from the specific hardware involved, there's usually also a Hardware Abstraction Layer (HAL) which may be built into—or sit alongside—the RTOS, taking care of the actual hardware accesses to perform I/O.

Connected devices also need a network stack, typically providing TCP/IP networking. The bottom of the stack talks to the network hardware to exchange packets, and the top of the stack provides stream and datagram APIs. On top of this is layered the security stack, to provide authentication and encryption services used by both cloud communications and FOTA (Firmware Over-The-Air update) services.

At the very top, there's the application, implementing the specific functionality of the device at hand. This talks to the application hardware and the system services and additionally takes care of cloud communication.

Usually, the stack has been integrated by the device maker:

- Blue parts in the diagram shown above indicates those that are completely unique to the application
- Red indicates parts that are more often than not open-source or vendor-provided codebases (such as FreeRTOS, lwip, mBedTLS)
- Purple indicates areas which may be based on open-source or vendor code but are often heavily customized for the application. For example, the cloud communication code may be an open-source client for MQTT (a widely used messaging protocol), but with modifications to use customer TLS certificates.

Integration & maintenance challenges

Some pre-integration often exists—for example, Arm provide packaged releases with Mbed OS, network stack and security stack, and Infineon/Cypress provide FreeRTOS, lwIP and Mbed TLS as part of their WICED platform (Wireless Internet Connectivity for Embedded Devices, a platform to enable Wi-Fi and Bluetooth connectivity in system design). Yet the design decisions made by

these integrators do not always line up well with the application requirements, resulting in heavy developer customization. That, in turn, comes with the additional complexity of having to merge new releases from the supplier with the existing code base.

Merging changes from suppliers is a requirement to maintain system stability and security over the long term—and in IoT deployments that can mean a decade or more—, especially as these packages usually include code that is directly network-facing, which is easiest for an attacker to target. While some vendors provide long term support branches ("LTS"), which retain API compatibility for essential security updates, the definition of "Long Term" is often not compatible with a product's lifecycle. For example, Mbed TLS has LTS releases which offer security updates without API changes for up to 3 years. But beyond that, the developer would need to integrate a possibly radically different API to maintain a secure product—or heavily compromise a product's security by continuing to rely on out-of-date code.

As always, the more software you're writing or integrating, the more maintenance you will have to perform on this code over the product's entire lifecycle. **Whereas an unconnected product might comprise 90% application code and 10% third-party code (and ongoing maintenance isn't required as physical access would be required for any attack), connected products are often 20% application code and 80% third-party code, all of which has to be maintained to protect the user and manufacturer's reputation.**

Security design

Besides maintenance, there's a very real problem with both design and implementation of security components. As with any specialist field, there's a lot of expertise required to make the correct trade-offs and design decisions when building a connected product—and people with the appropriate skills are rare and hence expensive to hire.

When areas of the product are being architected from scratch—especially parts which may not be serviced adequately by well-supported open source software—the risks associated with a subtly-flawed design decision could be significant.

Value and cost predictability

As can be seen in the architecture diagram, there's a huge amount of software required to build a secure connected product—and most of it does not depend on the application itself. **Not only is the time and money spent on integrating and maintaining external components a huge burden to a product's lifetime costs, it is also essentially invisible to the end user, and doesn't differentiate the product in the market.**

Millions of engineer-hours have gone into reinventing the "connectivity wheel" for every single IoT product that has ever shipped. Complexity, budgets, schedules and lack of relevant domain knowledge has also meant that many of these products suffer from latent security issues just waiting to ruin someone's day.

Solving maintenance issues in IoT

As noted, one of the major challenges with solving the maintenance issue in an MCU design is the close integration between the RTOS and the application. Larger systems such as desktop computers and mobile phones have always had an OS/application split, with the platform supplier, e.g. Microsoft, maintaining the operating system & network stack and providing updates over time to keep it secure.

So, could these problems be addressed with a similar OS/application split applied to embedded systems? There are three issues that crop up:

- Who is responsible for maintaining and updating the operating system, and how can they ensure that updates do not have a detrimental effect on product operation?
- How much extra cost and complexity does changing OS (or the way the OS is integrated) bring to development?
- What's the impact on the Bill of Material (BOM) cost to provide this split?

Responsibility for updates

When compared to a desktop or mobile application, embedded IoT applications are vastly different. Most desktop applications—and almost all mobile applications—are human-centric, providing a

service or function to the user via processing and connectivity provided by the host device. As such, performance and consistency are appropriate for humans; the user interface might change, a screen might take a couple of seconds longer to appear, or functionality may be degraded if connectivity is not available—but humans quickly adapt.

In comparison, an embedded IoT application is I/O-centric and may have non-negotiable performance targets—whether these are for response time, functionality in the event of degraded communication, or power consumption. These targets depend on the specific use case of the device.

This different set of developer expectations, coupled with the reality that updates have to be deployed to unattended devices that may be physically entirely inaccessible for their lifetime, result in a very different burden on the shoulders of whoever maintains the devices.

Essentially, the developer needs to have confidence that no third party updates will ever break the deployed application. There are two ways the maintainer can help relieve developer concerns:

1. **Comprehensive testing.** A cursory smoke check or ad-hoc manual quality assurance is not going to uncover the insidious issues that cause problems with embedded systems. All relevant guaranteed behavior and performance has to be tested continuously (so regressions can be addressed well before any release) and in an automated fashion (so that testing is always performed in a consistent manner).

2. **Minimized functionality.** Almost as important as testing is minimization of the maintained footprint. The less functionality that is delegated to the third party, the less functionality that could change behavior in the event of an update. In the real world, device performance can vary based on factors out of anyone's control—RF (Radio Frequency) propagation or network routing, for example. And when diagnosing such issues, being able to remove third-party updates from the list of possible culprits is very helpful.

Just as hardware developers are intimately aware of DFM (Design for Manufacture, a set of practices that help products move smoothly from prototype to production with high yield and minimal field failures), software developers are aware of DFT (Design for Testability).

In the world of long-lived IoT products, consistent testing over long periods of time is essential. This means that testing must be automated vs. manual, as people working at an organization will change over time. As such, at a minimum, OS and networking code must be defended with a full suite of automated tests: from build-time unit testing to system testing on target hardware, to regular fuzz testing of external interfaces in order to uncover unintended behaviors. This level and duration of DFT and test automation is obviously expensive.

Complexity of development and the impact on cost and BOM

Just as developers get comfortable with a particular Instruction Set Architecture (ISA), they also get expertise in an operating system architecture, a set of development tools, and development & debug workflows. Changing any of these components—even for tangible long-term gains—is painful in the short term and can introduce uncertainty in project schedules.

In an ideal world, a developer would be able to continue to use their preferred tools and RTOS while still having someone else provide the essential maintenance for long-term support.

One advantage of linking the operating system with the application is that it becomes easy to only pull in OS code that the application actually makes use of. This reduces the footprint of the OS and hence reduces the overall memory usage of the product.

Adding any functionality to an embedded system—even reliable FOTA—does increase the hardware BOM cost, mainly related to flash and RAM usage. Unfortunately, there's no real way around this, but the upsides are significant and the incremental costs are generally small, especially when compared to the cost of application development.

Maintained microvisor vs. maintained operating system

It's clear from the sections above that attempting to provide a single maintained RTOS for a wide variety of applications is only going to be successful for a subset of developers—those who are already familiar with the chosen OS, and those who do not rely on custom modifications to that OS.

If, however, we approach the problem from a different angle and instead look at what services we are trying to provide to the embedded developer, a new solution appears: a hypervisor that runs alongside the developer's RTOS and application code and insulates it from common attack vectors. Let's call it a *microvisor.*

The areas which are security-related and hence require long-term maintenance are:

- **Secure boot**, as attackers may target the boot process, and countermeasures may need to be deployed. A breach here could expose keys, application code, and more.
- **Network stack**, as this can be attacked via the local network (and in some cases, via the internet).
- **Security stack**, as crypto algorithms will need to evolve over time as protocol bugs are discovered and algorithmic weaknesses are discovered.
- **FOTA & connectivity services**, which are built on top of the aforementioned layers and hence will also evolve over time.
- **Network drivers**, as issues can appear over time with wireless networks evolving (compatibility updates, fixes for security issues in wireless module firmware, etc.)

With appropriate hardware support, such a microvisor can be built—one which claims the necessary peripherals for network support at boot time, and establishes an application-independent connection to the cloud service that provides FOTA updates. But aside from that, it stays largely out of the way of the developer's application and choice of RTOS.

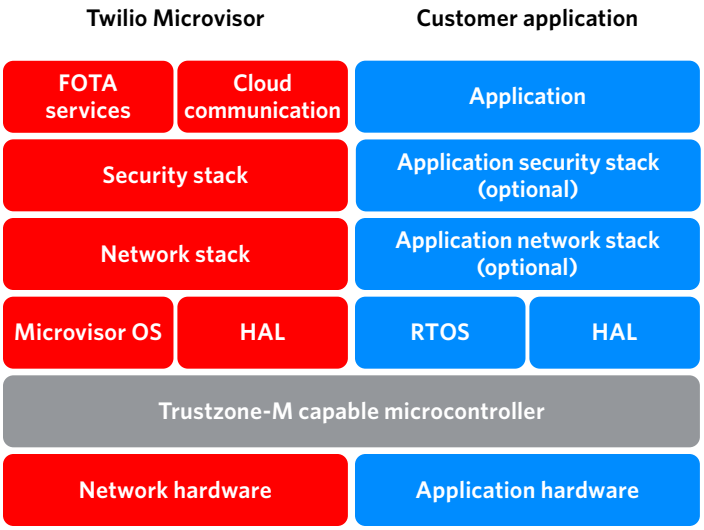
The microvisor can then protect itself from attack, whether it be via hardware tampering or a network interface. It can also protect the developer's application from a large variety of attacks.

Part II: Implementing a New IoT Architecture with Twilio Microvisor

Overview

We are now taking a deep dive into what a microvisor architecture looks like; in particular, how Twilio is approaching design with Twilio Microvisor.

Revisiting our device architecture diagram, a microvisor-based system looks like this:



We now have a very clear division between domains in the microcontroller; the red parts on the left are provided and maintained by Twilio, and deal with security and network services.

On the right, the blue parts all come from the developer—including their choice of RTOS. For maximum compatibility with existing code, the application can include its own network and security stack, allowing third party libraries such as those provided by Amazon to connect to AWS IoT to be used without modification.

This architecture is enabled by Arm's TrustZone® technology, which—at a silicon level—enables process and resource isolation. The processor core itself adds banked resources for "secure" and "non-secure" modes, and this separation extends through the bus fabrics to individual peripherals and memory areas which can be configured to only be accessible from a specific zone.

Additionally, the silicon supported by the Twilio Microvisor includes other specialist security features such as DPA (Differential Power Analysis) resistant cryptography accelerators, tamper detection, and so on.

This architecture allows the developer's RTOS and application to run mostly unchanged, able to talk to non-networking hardware without any interference, and essentially run as if they were on a normal MCU. There is no degradation in performance as the microvisor is only involved in boot- and network-related operations.

The secure "bubble" that the application runs within also largely removes the requirement for RTOS maintenance—the developer can rely on the microvisor to protect the boot process, mitigate at least first level network attacks, and provide a secure and fail-safe way to update RTOS, application and application keys when required. Fewer updates here mean less ongoing QA for devices that have shipped and lower product maintenance costs.

Microvisor peripheral usage

At boot, the microvisor assumes control of peripherals required for its operation. These fall into four categories:

Microvisor system

As the microvisor starts first, it is responsible for initializing the oscillators and clock tree. A single timer peripheral is used for microvisor purposes.

Off-die storage

The on-die flash is valuable due to its performance and security characteristics, hence it is not used for storage of already encrypted data such as staged OTA upgrades. Secondary storage, in the form of an external QSPI flash, is used for these purposes—and as such the QSPI device is managed by the microvisor.

Excess space in the QSPI device is available for application use via microvisor APIs.

Networking peripherals

As the microvisor manages the connection to the internet, the communications peripherals necessary for this are claimed by the microvisor at startup.

For Wi-Fi, BLE and Ethernet—at least on the initial silicon—that means the SDIO peripheral, and for cellular this means either USB host or UART, or both in some cases. Required control and clock lines are also claimed.

Diagnostics

A tricolor LED is used to indicate system status.

All other peripherals in the MCU are available for uncontended application use, just as they would be in a non-microvisor system—i.e. they are found at the locations documented in the MCU's datasheet and are connected to the application NVIC and DMA controller as expected.

Memory

The majority of on-die memory, both flash and RAM, is available to the application. Because the initial part that the Twilio microvisor ships on has not yet been announced, we can only give further details under a 3-way NDA with the silicon vendor.

Networking

The microvisor takes responsibility for the cloud-facing network interfaces. In order to provide FOTA services, it needs to be able to establish a secure outbound connection to the companion Twilio service.

Included in this responsibility is maintenance of the firmware that runs within the external controllers/modems, as this firmware has been targeted by hackers many times in recent years and so applying updates to these subsystems is an essential part of maintaining the device security boundary.

With a TCP stack already being attached to the interface, a traditional OS-like approach would provide a socket API to the application, allowing sharing of the stack overhead between the microvisor and application zones; there are downsides to this approach though:

- **Compatibility**. Existing applications and example code could need extensive modifications to talk to the microvisor socket API. Unlike a linux application, embedded applications are often written to work with non-BSD type socket interfaces with different threading models.

- **Memory usage**. Because the network stack is running in the microvisor zone, RAM allocations for packet buffers and so on would also need to come from the microvisor zone—likely limiting application performance for protocols like TCP where the send buffer size is directly related to upstream throughput.

As such, the microvisor provides a virtual network interface which accepts (and delivers) raw packets as if it were a physical interface. This is a much easier layer at which to connect to any network stack, as all stacks are designed to be portable across network types.

The downside of this approach is that in many cases there will be two TCP stacks running within the microcontroller—one for the microvisor and one for the application—though the microvisor stack is by default optimized for space vs. performance.

Applications that are happy relying on the microvisor and cloud service to provide the secure cloud tunnel can instead just exchange messages using microvisor APIs, with the microvisor assuming responsibility for safe and secure transfer of data to and from the cloud.

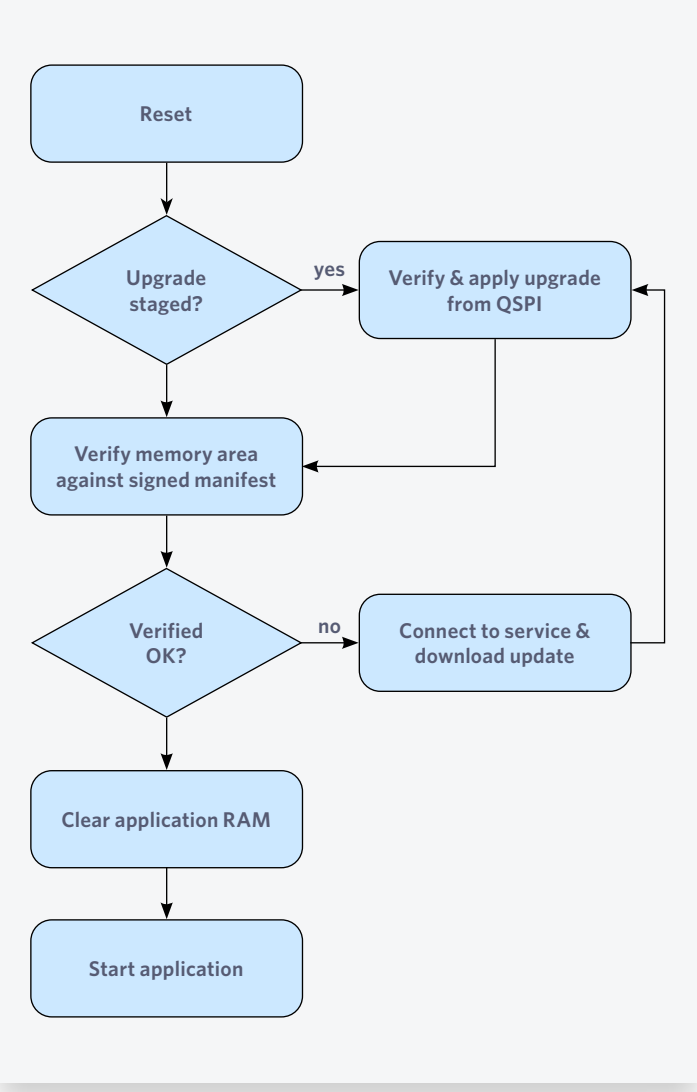
Some applications also have non-cloud facing network interfaces—for example, a ZigBee or LoRA gateway product. These network interfaces are totally managed by the application code, without any microvisor involvement.

First boot

The Twilio Microvisor installation process permanently configures the MCU to boot in secure TrustZone mode with the entrypoint being in secure on-die flash.

Startup flow

A simplified version of the startup flow is illustrated below; in the event that no verified code is available to run, the device will wait for the service to provide a valid image. For more detail on upgrade signing, see FOTA upgrades below.



Application of staged updates

At cold boot, or after an upgrade has been downloaded, the microvisor's secure boot stub will check for valid upgrade data in external QSPI and decrypt and apply this upgrade package—which can include updates to the upgrade code—to the internal flash areas. This process applies to both microvisor updates and application updates; after successfully writing this upgrade data, the external QSPI image is invalidated, ensuring that pending upgrades are applied fully even in the event of a power loss or unexpected reset.

Microvisor image verification

The next step in the boot process is a cryptographic verification step for the microvisor, whereby the internal microvisor flash content is hashed and checked against a signed manifest, also stored internally. If this step fails, the device has likely been tampered with (or has suffered a hardware failure) and will not boot. Devices can be recovered from this state by loading the QSPI with a correctly signed and encrypted upgrade image which will refresh the internal flash state.

Successful verification results in the microvisor being entered by the boot stub.

Interrupt handling

There are two NVICs in the microcontroller, one being dedicated to the microvisor and one to the application. Because the majority of the MCU peripherals are assigned to the application, the microvisor interrupts are all related to either the upstream communications peripherals or to the microvisor scheduler.

Due to the dedicated NVIC, the application & application RTOS do not require any modification to work with the microvisor—an interrupt raised by an application peripheral will be decoded, prioritized and executed just as it would be in a system without a microvisor present. Unless the developer calls into the microvisor from their interrupt handler, no microvisor code is run in an application interrupt path.

Interrupt priorities

In order for the microvisor to be able to regain control over an errant application, some microvisor interrupts must take priority over application ones. Without this, a buggy application could preempt the microvisor continuously and cause a device to become unrecoverable.

This has important implications for application interrupt handling though—as the majority of peripherals are on the application side, the majority of interrupt handlers are also on the application side. Many handlers will be timing critical—for example, a UART interrupt handler must read data from the UART holding register before the next byte arrives to prevent data loss.

In order to prioritize application interrupts appropriately, high priority interrupts are wrapped by the microvisor which then dispatches them from microvisor mode. This guarantees that they can interrupt network processing and other management tasks, while still ensuring the microvisor can regain control of execution if required.

IRQs that are natively handled by the application NVIC cannot interrupt microvisor code and as such may see longer latencies in circumstances where the microvisor is busy.

In either case, the ISR code is identical to such code running in a non-microvisor context.

Exception handling

Reporting/crash logging

Uncaught exceptions result in the microvisor collecting information about the exception (register snapshot, stack dump, optional memory dump) and optionally persisting this information until it can be reported to the service.

This allows crash reports to be collected en-masse from an installed base by an application developer and used to improve software quality.

Non-banked exceptions

Some processor exceptions are not banked between microvisor and application zones by the Cortex-M33 core, such as Hardfault and Busfault. As such, these faults are always handled by the microvisor.

The SecureFault exception, triggered when application code attempts to access microvisor memory or peripherals, is caught by the microvisor and reported via the standard crash reporting mechanism. The application code is restarted as if it had been cold-booted.

Banked exceptions

These exceptions—MemManage, UsageFault, SVcall, SysTick—are delivered as would be expected on a non-microvisor system.

The banking—and banked MPU—ensure that the application RTOS can use memory protection to catch errant behavior without involving the microvisor, ensuring application-consistent behavior of the system when such exceptions are encountered.

Watchdog

The microvisor can provide a watchdog for application code with a configurable period. If the watchdog is not refreshed in the appropriate interval, the application will be reset as if a hardware watchdog had fired.

FOTA upgrades

A key feature of the microvisor is the FOTA (firmware over-the-air) upgrade system. This system allows the whole application—or just parts of it—to be updated on any number of devices at a time convenient to the application.

Application manifests

An application package contains a manifest, which defines what code/data should be stored in each defined memory area—either within on-die flash or in QSPI flash. The service is aware of the device's current operational manifest, and so when a new

package is queued for FOTA, only areas which do not match will be deployed to the device.

The manifest also allows a device to—regardless of connectivity—cryptographically verify the application is complete before execution.

Customers can deploy public keys to the device & service at time of manufacture and sign any or all parts of the manifest with the corresponding private keys. This process allows the service to verify that a manifest should be accepted by a device—rejecting deploys by a bad actor with service access early in the process.

In the event of the service being severely compromised, the device itself verifies the signatures after staging and will reject incorrectly signed upgrades without disturbing the currently valid and running code.

Upgrade staging

The microvisor takes care of moving data required to apply an update from the cloud to a staging area in device QSPI storage, where it remains encrypted. Once the device has all the changed parts of the application stored safely in the staging area, it applies the upgrade in a fail-safe and restartable way, to ensure the upgrade appears atomic from the application's point of view.

Upgrade notification

The application can be notified during the upgrade staging process, as it may want to indicate progress to an end user. Once the staging process is complete, the application is notified and can pick a convenient time to perform the upgrade.

If required, the customer can force the upgrade to be applied at any time after staging; this may be used if the old code is behaving badly, for example.

Power management

As noted earlier, the microvisor looks after the oscillators and clock tree so that it can perform its management tasks. It also manages the various sleep modes that the system can enter to save power.

Deep sleep

The lowest power mode available is deep sleep, in which only the RTC and backup RAM is powered, with the processor and all other peripherals powered down. The first silicon draws less than half a microamp when in deep sleep mode.

Exit from deep sleep can be triggered by an RTC timer or a wakeup pin, and looks to the application like a cold boot. The reason for application boot is available via a "wake reason" microvisor API.

The application enters deep sleep by configuring wakeup pins as necessary and then calling a microvisor API specifying a wake time.

Shallow sleep

In this mode, RAM and GPIO states are preserved, and some peripherals may remain clocked. Typically this mode is entered using the ARM WFI instruction after configuring the MCU state, but in the microvisor system the application calls a microvisor API, which registers the application's desire and synchronizes sleep entry with other tasks that may be in process.

It's the application's responsibility, as with a non-microvisor architecture, to ensure that wake sources are configured appropriately before calling the sleep API.

Exiting this mode is typically via an interrupt either from a timer or a peripheral.

System services

The microvisor provides additional services to the application, which help with rapid application development and debugging.

Logging

Serial logging is often used during embedded development to provide insight into program flow and state. Often this is regularly flushed to determine where the application hit a serious issue to aid debug. This approach is obviously still possible with a microvisor application but doesn't work remotely or at scale and uses up a UART on the device.

With the microvisor, a unidirectional logging stream can be written to by the application. When the device has a connection to the service, this stream is packetized, encrypted, and securely delivered to the service where it can be viewed in the Twilio Console or consumed by a customer's application via service APIs, resulting in a much more scalable and secure logging solution.

Console

Here, a bidirectional stream is made available to the application by the microvisor, and securely surfaced via service APIs. This allows an application to provide a traditional console interface or REPL for interactive control with minimal overhead.

Debug

To maintain the system's security boundary, the initial loading of the microvisor code into the MCU at production time irrevocably disables the hardware JTAG/SWD interface on the silicon.

However, as the microvisor has higher privilege than the application on the MCU, it provides equivalent application debug functionality to developers over the secure service connection. This connection is routed through the service to a local gdbserver stub on the developer's machine, allowing any GDB compatible debug tool to read & write memory, set breakpoints, single step code, and so on no matter where the device is in the world and how it is connected.

Note that a developer can mark areas in the manifest as non-debuggable, which will disable these debug features within the microvisor, helping protect against any possible cloud-based attack.

Conclusion

In the end, any architecture is a result of innumerable individual design choices, guided by the learnings and experiences of the team that constructed it. In embedded systems particularly—where processor cycles and microamps are often critical—there is no such thing as a free lunch, but we believe that the trade-offs we have made are clearly justified in pursuit of long-lived security and stability that can be conferred on any number of products built on this base.

twilio

The team welcomes comments and discussion on our design; you can contact us at microvisor@twilio.com. Thank you!