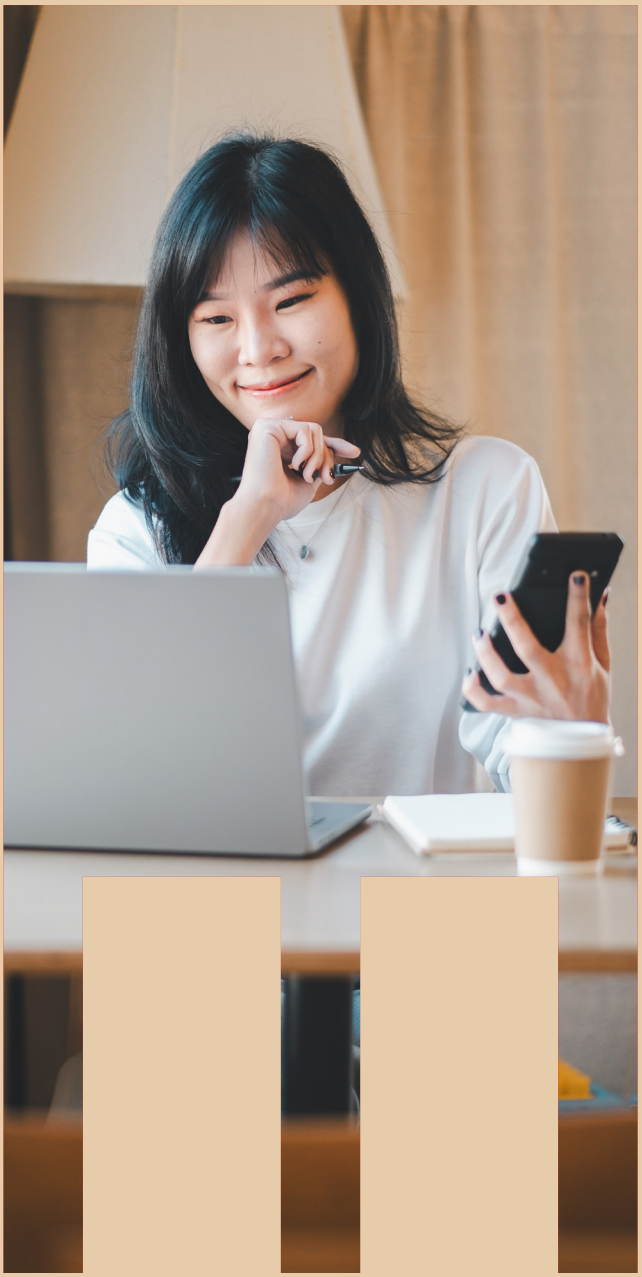
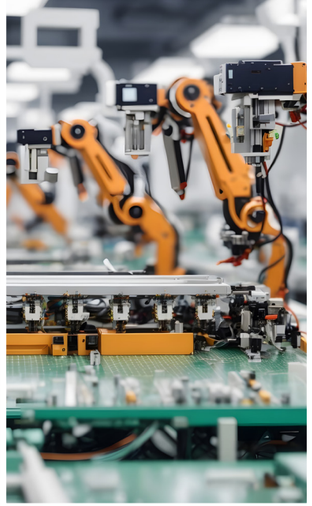
We get technical

- Real-Time Operating Systems (RTOS) and their applications
- How to select and use the right ESP32 Wi-Fi/ Bluetooth module
- IoT security fundamentals: connecting securely to IoT Cloud services
- Application layer protocol options for M2M and IoT functionality



DigiKey



contents

3 Real-Time Operating Systems (RTOS) and their applications

5 Use a cellular and GPS SiP to implement asset tracking for agriculture and smart cities

10 How to select and use the right ESP32 Wi-Fi/Bluetooth module

13 IoT security fundamentals: connecting securely to IoT Cloud services

17 **Special feature: retroelectro**
The ALOHA System: Task II

21 Application layer protocol options for M2M and IoT functionality

24 Deploy a secure Cloud-connected IoT device network complete with Edge computing capabilities

28 Use rugged multiband antennas to solve the mobile connectivity challenge

31 Getting started with Zephyr: a developer's guide to your first project

DigiKey

Editor's note

In the dynamic world of technology, wireless communication stands as a cornerstone of modern innovation, reshaping how we interact with our surroundings and with each other.

Wireless technology, at its core, is the transmission of information over a distance without the need for wires, cables, or any other physical connection. This seemingly simple concept has revolutionized communication, enabling not just cell phone calls and text messages, but also the rapid expansion of the Internet, media streaming, and much more. The impact of wireless technology extends beyond personal communication, deeply embedding itself in various sectors such as healthcare, automotive, and the burgeoning field of the Internet of Things (IoT).

The future trajectory of wireless technology is towards faster, more reliable, and more pervasive networks. The ongoing rollout of 5G technology promises not only higher data rates and reduced latency but also the ability to connect a massive number of devices simultaneously. This leap forward presents engineers with both opportunities and challenges, requiring innovative solutions in network infrastructure, signal processing, and application development.

Wireless technology is a dynamic and ever-evolving field that demands a multidisciplinary approach from engineers. Its profound impact on various sectors makes it an exciting area for engineering exploration and innovation. As we move forward, the role of engineers will be pivotal in shaping the future of wireless communication, ensuring its growth, efficiency, and security.

For more information,
please check out our website at
www.digikey.com/automation.

Real-Time Operating Systems (RTOS) and their applications

Written by:
Lim Jia Zhi, Senior Embedded Software Engineer at DigiKey

A Real-Time Operating System (RTOS) is a lightweight OS used to ease multitasking and task integration in resource and time constrained designs, which is normally the case in embedded systems. Besides, the term 'real-time' indicates predictability/determinism in execution time rather than raw speed, hence an RTOS can usually be proven to satisfy hard real-time requirements due to its determinism.

Key concepts of RTOS are:

Task

Tasks (could also be called processes/threads) are independent functions running in infinite loops, usually each responsible for one feature. Tasks are running independently in their own time (temporal isolation) and memory stack (spatial isolation). Spatial isolation between tasks can be guaranteed with the use of a hardware memory protection unit (MPU), which restricts accessible memory region and triggers fault exceptions on access violation. Normally, internal peripherals are memory-mapped, so an MPU can be used to restrict access to peripherals as well.
Tasks can be in different states:

- Blocked – task is waiting for an event (e.g., delay timeout, availability of data/resources)
- Ready – task is ready to run on CPU but not running because CPU is in use by another task
- Running – task is assigned to be running on CPU

Scheduler

Schedulers in RTOS control which task to run on the CPU, and different scheduling algorithms are available. Normally they are:

- Pre-emptive – task execution can be interrupted if another task with higher priority is ready
- Co-operative – task switch will only happen if the current running task yields itself

Pre-emptive scheduling allows higher priority tasks to interrupt a lower task in order to fulfil real-time constraints, but it comes in the cost of more overhead in context switching.

Inter-task communication (ITC)

Multiple tasks will normally need to share information or events with each other. The simplest way to share is to directly read/write shared global variables in RAM, but this is undesirable due to risk of data corruption caused by a race condition. A better way is to read/write file-scoped static variables accessible by setter and getter functions, and race conditions can be prevented by disabling interrupts or using a mutual exclusion object (mutex) inside the setter/getter function. The cleaner way is using thread-safe RTOS objects like message queue to pass information between tasks.

Besides sharing of information, RTOS objects are also able to synchronize task execution because tasks can be blocked to wait for availability of RTOS objects. Most RTOS have objects such as:

- Message queue
  - First-in-first-out (FIFO) queue to pass data
  - Data can be sent by copy or by reference (pointer)
  - Used to send data between tasks or between interrupt and task
- Semaphore
  - Can be treated as a reference counter to record availability of a particular resource
  - Can be a binary or counting semaphore
  - Used to guard usage of resources or synchronize task execution
- Mutex
  - Similar to binary semaphore, generally used to guard usage of a single resource (MUTual EXclusion)
  - FreeRTOS mutex comes with a priority inheritance mechanism to avoid priority inversion (condition when high priority task ends up waiting for lower priority task) problem
- Mailbox
  - Simple storage location to share a single variable
  - Can be considered as a single element queue
- Event Group
  - Group of conditions (availability of semaphore, queue, event flag, etc.)
  - Task can be blocked and can wait for a specific

combination condition to be fulfilled
- Available in Zephyr as a Polling API, in FreeRTOS as QueueSets

System tick

RTOS need a time base to measure time, normally in the form of a system tick counter variable incremented in a periodic hardware timer interrupt. With system tick, an application can maintain more than time-based services (task executing interval, wait timeout, time slicing) using just a single hardware timer. However, a higher tick rate will only increase the RTOS time base resolution, it will not make the software run faster.

Why use RTOS?

Organization

Applications can always be written in a bare metal way, but as the code complexity increases, having some kind of structure will help in managing different parts of the application, keeping them separated. Moreover, with a structured way of development and familiar design language, a new team member can understand the code and start contributing faster. RFCOM Technologies has developed applications using different microcontrollers like Texas Instruments' Hercules, Renesas' RL78 and RX, and STMicroelectronics' STM32 on a different RTOS. Similar design patterns allow us to develop applications on different microcontrollers and even a different RTOS.

Modularity

Divide and conquer. By separating features in different tasks, new features can be added easily without breaking other features, provided that the new feature does not overload shared resources like the CPU and peripherals. Development without RTOS will normally be in a big infinite loop where all features are part of the loop. A change to any feature within the loop will have an impact on other features, making the software hard to modify and maintain.

Communication stacks and drivers

Many extra drivers or stacks like TCP/IP, USB, BLE stacks, and graphics libraries are developed/ported for/to existing RTOSs. An application developer can focus on an application layer of the software and reduce time to market significantly.

Tips

Static allocation

Use of static allocation of memory for RTOS objects means reserving memory stack in RAM for each RTOS object during compile time. An example of a static allocation function in freeRTOS is xTaskCreateStatic(). This ensures that a RTOS object can be created successfully, saving the hassle of handling a possible failed allocation and making the application more deterministic.

In terms of deciding stack size needed for a task, the task can be run with a bigger (more than enough) stack size and then the stack usage can be checked in runtime to determine the high-water mark. There is a static stack analysis tool available as well.

Operating system abstraction layer (OSAL) and meaningful abstraction

Just like Hardware Abstraction Layer (HAL), use of the RTOS abstraction layer allows application software to migrate easily to other RTOSs. Features of RTOSs are quite similar, so creating OSAL should not be too complicated. For example:

Using the freeRTOS API directly:

```
if( xSemaphoreTake( spiMutex, ( TickType_t ) 10 ) == pdTRUE ) { // dosomething }
```

Wrapping the RTOS API into OSAL:

```
if( osalSemTake( spiMutex, 10 ) == true) { //dosomething }
```

Using the abstraction layer on inter-task communication to make code more readable and minimize the scope of an RTOS object:

```
if( isSpiReadyWithinMs( 10 ) ) { // doSomething }
```

Additionally, the abstraction also allows a programmer to change the RTOS object used underneath (e.g., from mutex to counting semaphore) if there is more than one SPI module available. OSAL and other abstraction layers help in software testing as well by simplifying mock function insertion during unit testing.

Tick interval selection

Ideally, a lower tick rate is better because of less overhead. To select a suitable tick rate, the developer can list down timing constraints of modules in an application (repeating interval, timeout duration, etc.). If there are some outlier modules needing a small interval, having a dedicated timer interrupt can be considered for the outlier modules rather than increasing RTOS tick rate. If the high frequency function is very short (e.g., write to register to turn an LED on/off), it can be done inside an Interrupt Service Routine (ISR), otherwise deferred interrupt handling can be used. Deferred interrupt handling is a technique of deferring interrupt computation into an RTOS task, the ISR will only generate an event through the RTOS object, then the RTOS task will be unblocked by the event and do the computation.

Tick suppression for low power application

Tickless idle disables tick interrupt when the system is going idle for a longer time. One significant way for embedded firmware to reduce power consumption is to put the system in low power mode for as long as possible. Tickless idle is implemented by disabling the periodic tick interrupt and then setting up a countdown timer to interrupt when a blocked task is due to execute. If there is no task waiting for a timeout, the tick interrupt can be disabled indefinitely until another interrupt occurs (e.g., button pressed). For example, in the case of a Bluetooth Low Energy (BLE) beacon, the MCU can be put into deep sleep between the advertising interval. As shown in Figure 1, the beacon is put into deep sleep mode for most of the time, consuming power in tens of µA.

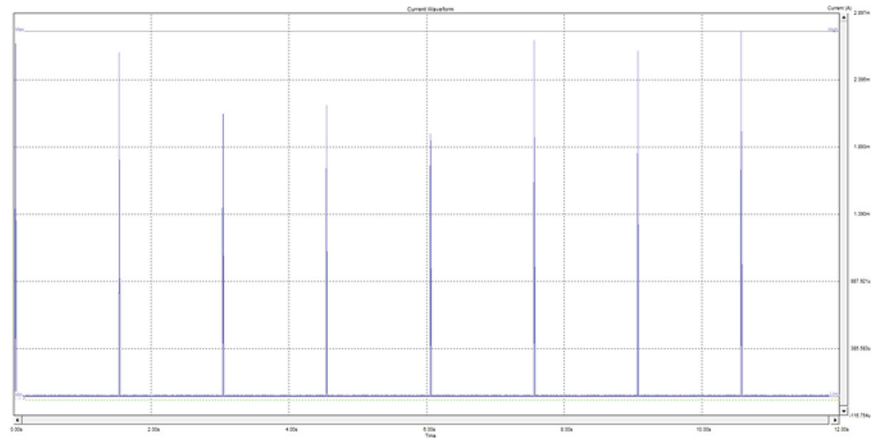
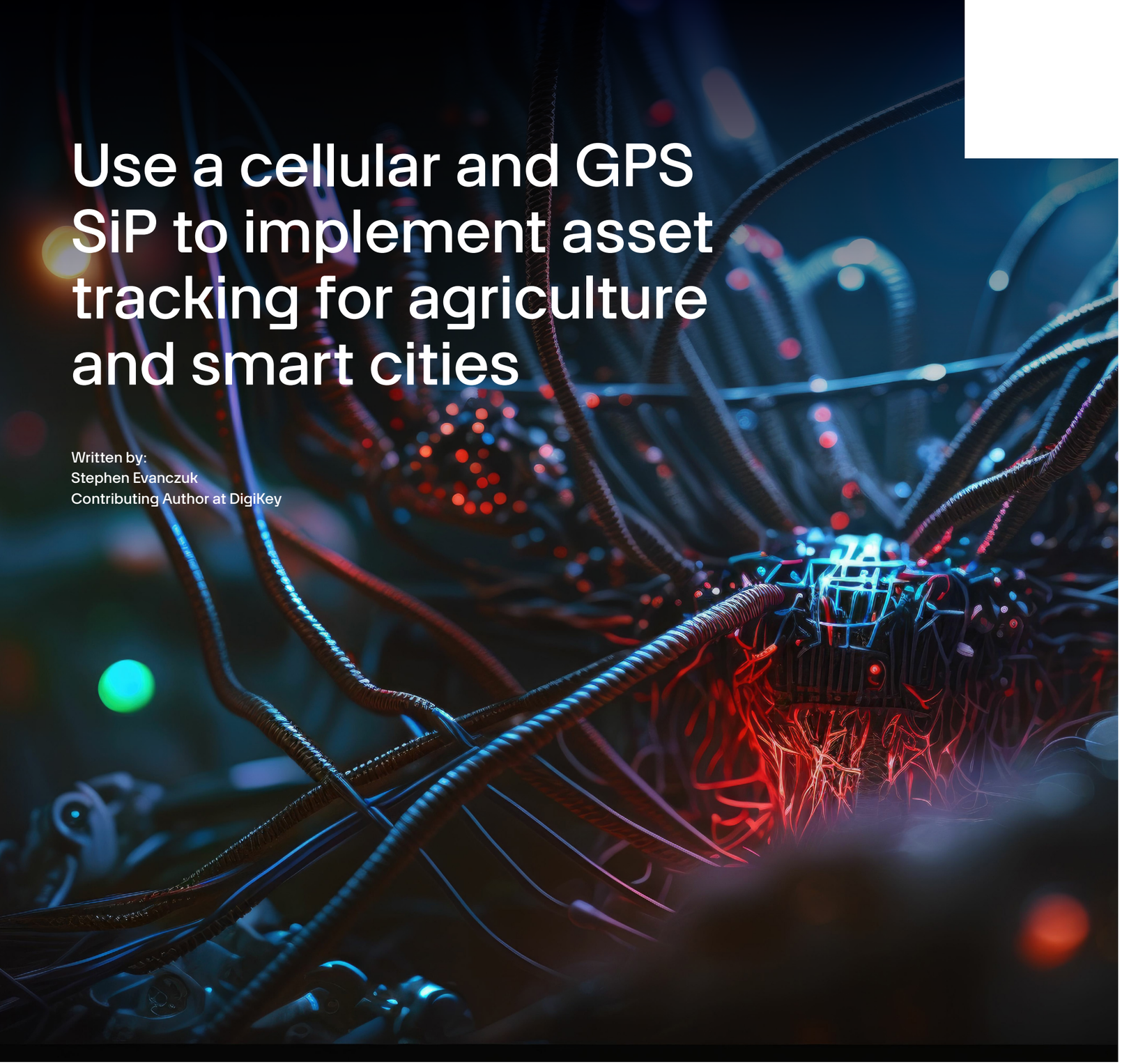


Figure 1: Current consumption of a BLE beacon Credit: RFCOM

Conclusion

An RTOS provides features like scheduler, tasks, and inter-task communication RTOS objects, as well as communication stacks and drivers. It allows developers to focus on the application layer of the embedded software, and design multitasking software with ease and speed. However, just like any other tools, it has to be used properly in order to bring out more value. To create safe, secure, and efficient embedded software, developers should know when to use RTOS features and also how to configure RTOS.

Use a cellular and GPS SiP to implement asset tracking for agriculture and smart cities



Written by:
Stephen Evanczuk
Contributing Author at DigiKey

Developers of Internet of Things (IoT) and asset tracking devices and systems for industry, agriculture, and smart cities need a way to communicate over long distances at minimal power for extended periods of time. Wireless technologies such as RFID tags, Bluetooth, and Wi-Fi are already widely used for asset tracking solutions, but they have limited range and consume too much power. What's required is a combination of GPS and an adaptation of infrastructure such as cellular networks that are already widely deployed and are designed for communications at longer ranges than available with Wi-Fi or Bluetooth.

LTE-based cellular networks were originally designed for wide bandwidth wireless connectivity for mobile products and devices. IoT applications, on the other hand, can get by using lower power, narrowband cellular technologies such as long-term evolution for machines (LTE-M) and Narrowband IoT (NB-IoT). Still, RF/wireless design is difficult, and developers lacking extensive experience, particularly with respect to cellular, face great difficulty implementing a functioning design that optimizes wireless performance and power consumption, while also meeting international regulatory guidelines for both cellular and GPS location services, as well as specific carrier requirements.

This article describes the trends and design requirements of asset tracking. It then introduces a GPS and cellular narrowband system-in-package (SiP) solution from Nordic Semiconductor and shows how it can greatly simplify the implementation of GPS-enabled cellular devices for asset tracking and other agriculture and smart city IoT applications.

Why asset tracking is increasingly important

The ability to ship products efficiently is vital to commerce: Amazon alone shipped an estimated five billion packages in 2019, spending almost $38 billion in shipping costs – a 37% increase over 2018. For any shipping company, delays, damage, and theft place a significant strain on manufacturers, distributors, and customers. For Amazon, nearly a quarter of those shipped packages were returned, 21% because the customer received a damaged package.

Amazon is by no means alone in allocating a significant portion of their budget to shipping. According to the 2020 State of Logistics report from the Council of Supply Chain Management Professionals (CSCMP), companies spent nearly $1.7 trillion on shipping costs in 2019 – an expenditure that accounts for 7.6% of the US gross national product (GDP). At these levels, the ability to track packages, identify delays and instances of

damage can provide significant benefit to suppliers and purchasers to correct shipment problems.

Besides following packages through the supply chain, most enterprises need improved methods for tracking their own assets and locating misplaced items. Yet, half of all businesses still manually log assets, and of those, many rely on employees to search through warehouses, plants, and physical locations to find missing assets.

Comparing connectivity technologies for asset tracking

Although a number of solutions have emerged to help automate asset tracking, the underlying technologies have limited coverage area, are expensive per unit cost, or have high power requirements. The latter is critical as asset tracking and remote IoT devices are battery-powered devices.

Conventional tracking methods based on passive radio frequency identification (RFID) cannot provide live data in transit and require packages to pass through some physical checkpoint to detect the RFID tag attached to a package. Battery-powered active RFID tags are able to provide real-time location data but require additional infrastructure and remain limited in coverage.

Compared to RFID tags, Bluetooth low energy (BLE) and Wi-Fi offer progressively greater range within a coverage area equipped with fixed locators for each technology. Building on a rich ecosystem of devices and software, BLE and Wi-Fi are already applied in location-based applications such as COVID-19 contact tracing and conventional real-time location services (RTLS), respectively. With the availability of direction-finding features in Bluetooth 5.1, the location of a tag can be accurately calculated based on angle-of-arrival (AoA) and angle-of-departure (AoD) data (Figure 1).

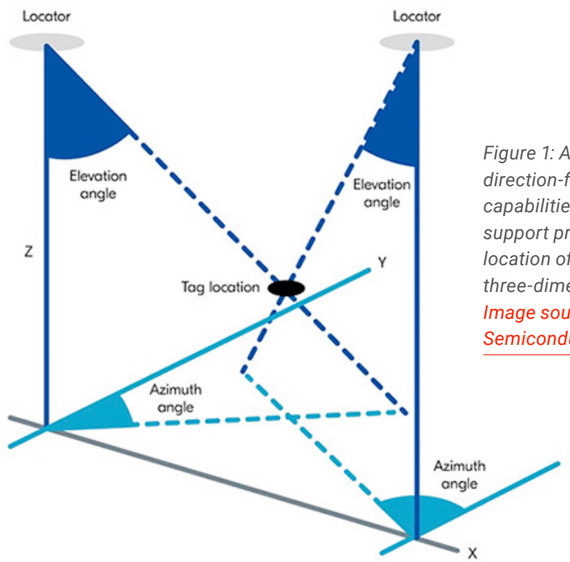


Figure 1: Advanced direction-finding capabilities in Bluetooth support precision location of a tag in three-dimensional space. Image source: Nordic Semiconductor

While BLE applications remain limited to short-range applications, Wi-Fi's greater range can make it effective for use in asset tracking applications within a warehouse or enterprise campus. Yet, Wi-Fi RTLS tags are typically expensive devices with power requirements that make batteries impractical, thereby limiting its use to tracking larger, expensive assets. At the same time, large-scale deployments using either of these technologies can suffer from increasing noise in their reception bandwidth, leading to lost or corrupted packets and degradation of location detection capabilities.

Despite their potential use for tracking assets locally, neither RFID, BLE, nor Wi-Fi can provide the range of coverage needed to easily track an asset once it leaves the warehouse or enterprise campus. The ability to track a package or piece of equipment regionally or even globally depends on the availability of a wireless technology able to achieve both extended reach and low power operation.

Alternatives based on low-power ultra-wideband (UWB) technologies can achieve significant range, but network coverage remains limited. In fact, few alternatives can provide the kind of global coverage already available with low-power wide-area network (LPWAN) cellular solutions based on LPWAN technology standards defined by 3rd Generation Partnership Project (3GPP) – the international consortium that defines mobile communications standards.

Achieving global reach with cellular connectivity

Among 3GPP standards, those based on LTE-M and NB-IoT technologies are designed specifically to provide a relatively lightweight cellular protocol well matched to IoT requirements for data rate, bandwidth, and power consumption.

Defined in 3GPP Release 13, LTE Cat M1 is an LTE-M standard that supports 1 megabit per second (Mbit/s) for both downlink and uplink transfers with 10 to 15 millisecond (ms) latency and 1.4 megahertz (MHz) bandwidth. Also defined in 3GPP Release 13, Cat-NB1 is an NB-IoT standard that offers 26 kilobits per second (Kbits/s) downlink and 66 Kbits/s uplink with 1.6 to 10 s latency and 180 kilohertz (kHz) bandwidth. Defined in 3GPP Release 14, another NB-IoT standard, Cat-NB2 offers higher date rates at 127 Kbits/s downlink and 159 Kbits/s uplink.

Although the specific characteristics of these two broad classes of LPWAN technology lie well beyond the scope of this brief article, both can serve effectively for typical asset tracking applications. Combined with sensors and global positioning satellite (GPS) capabilities in compact packages, asset tracking solutions based on LTE-M or NB-IoT based cellular LPWANs can support the kind of capabilities required for asset management and end-to-end logistics.

Given LPWAN's potential for achieving greater efficiency and cost savings, cellular LPWAN continues to play a greater role in logistics. With the availability of the nRF9160 SiP from Nordic Semiconductor, developers can more quickly and easily serve the growing demand for LPWAN-based devices needed for more effective asset tracking or other IoT applications.

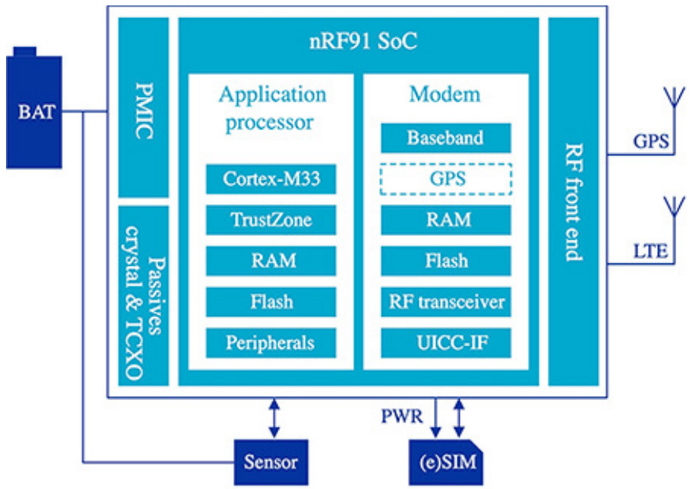
How a SiP device can deliver a drop-in asset tracking solution

Nordic Semiconductor's low-power nRF9160 SiP device combines a Nordic Semiconductor nRF91 system-on-chip (SoC) device with support circuitry to provide a complete LPWAN connectivity solution in a single 10 x 16 x 1.04 millimeter (mm) land grid array (LGA) package. Along with an Arm Cortex-M33-based microcontroller dedicated to application processing, nRF91 SoC variants integrate an LTE-M modem in the NRF9160-SIAA SiP, NB-IoT modem in the NRF9160-SIBA SiP, and both LTE-M and NB-IoT as well as GPS in the NRF9160-SICA SiP. Furthermore, the nRF9160 SiP is pre-certified to meet global, regional and carrier cellular requirements, allowing developers to quickly implement cellular connectivity solutions without the delays typically associated with compliance testing.

All SiP versions combine the microcontroller-based application processor and modem with an extensive set of peripherals, including a 12-bit analog-to-digital converter (ADC) often needed in sensor designs. The SiP further packages the SoC with an RF front-end, power management integrated circuit (PMIC), and additional components to create a drop-in solution for LPWAN connectivity (Figure 2).

Serving as the host processor, the SoC's microcontroller integrates a number of security capabilities designed to meet the growing demand for security in connected devices, including IoT devices and asset tracking systems. Building on the Arm TrustZone architecture, the microcontroller embeds an Arm Cryptocell security block, which combines a public key cryptography accelerator

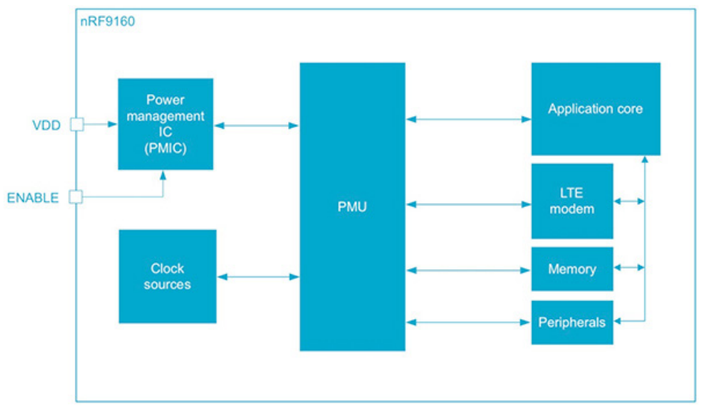
Figure 2: The Nordic Semiconductor nRF9160 SiP combines an SoC with application processor and LTE modem with other components needed to implement a compact low power cellular-based design for asset tracking or other IoT applications. Image source: Nordic Semiconductor



with mechanisms designed to protect sensitive data. In addition, a secure key management unit (KMU) provides secure storage for multiple types of secret data including key pairs, symmetric keys, hashes, and private data. A separate system protection unit (SPU) also provides secure access to memories, peripherals, device pins and other resources.

In operation, the SoC's microcontroller serves as the host, executing application software as well as starting and stopping the modem. Other than responding to start and stop commands from the host, the modem handles its own operations using its substantial complement of integrated blocks including a dedicated processor, RF transceiver, and modem baseband. Running its embedded firmware, the modem fully supports 3GPP LTE release 13 Cat-M1 and Cat-NB1. Release 14 Cat-NB2 is supported in hardware but requires additional firmware to operate.

Figure 3: The nRF9160 SiP includes a PMU that automatically controls clocks and supply regulators to optimize power consumption. Image source: Nordic Semiconductor



How the nRF9160 SiP achieves low power cellular connectivity

The nRF9160 SiP combines its extensive hardware functionality with a full set of power management features. Its included PMIC is supported by a power management unit (PMU) which monitors power usage and automatically starts and stops clocks and supply regulators to achieve the lowest possible power consumption (Figure 3).

Along with a System OFF power mode, which maintains power only to circuits needed to wake the device, the PMU supports a pair of System ON power sub modes. After power-on-reset (POR), the device comes up in the low-power sub mode, which places functional blocks including the application processor, modem, and peripherals in an idle state. In this state, the PMU automatically starts and stops clocks and voltage regulators for different blocks as needed.

Developers can override the default low-power sub mode, switching instead to a constant latency sub mode. In constant latency sub mode, the PMU maintains power to some resources, trading an incremental increase in power consumption for the ability to provide a predictable response latency. Developers can invoke a third power mode using the external enable pin, which powers down the entire system. This capability would typically be used in a system design that uses the nRF9160 SiP as a communications coprocessor controlled by the host system's main processor.

These power optimization features enable the SiP to achieve the kind of low power operation needed to ensure extended battery life in an asset tracking device. For example, with the microcontroller in the idle state and the modem powered down, the SiP consumes only 2.2 microamps (µA) with the real-time counter active. With the microcontroller and modem both off and power maintained only to the general-purpose input output (GPIO)-based wakeup circuitry, the SiP consumes only 1.4 µA.

The SiP continues to achieve low power operation while executing various processing loads. For example, running the CoreMark benchmark with a 64 MHz clock requires only about 2.2 milliamps (mA). Of course, as more peripherals are enabled, power consumption rises accordingly. Still, many sensor-based monitoring applications can often operate effectively at reduced operating rates that help maintain low power operation. For example, current consumption for the integrated differential successive approximation register (SAR) ADC drops from 1288 mA to less than 298 mA when switching from a high accuracy clock to a low accuracy clock for sampling in either scenario at 16 kilosamples per second (Ksamples/s).

The device also uses other power optimization features for its other functional blocks including GPS. In normal operating mode, continuous tracking with GPS consumes about 44.9 mA. By enabling a GPS power saving mode, current consumption for continuous tracking drops to 9.6 mA. By reducing the GPS sampling rate from continuous to every two minutes or so, developers can significantly reduce power. For example, the GPS module consumes only 2.5 mA when performing a single-shot GPS fix every two minutes.

The device's support for other power saving operating modes also extends to the nRF9160 SiP's modem. With this device, developers can enable modem features supporting special cellular protocols designed specifically to reduce power in battery-powered connected devices.

Utilizing low power cellular protocols

As with any wireless device, the largest contributor to power consumption, besides the host processor, is typically the radio subsystem. Conventional cellular radio subsystems take advantage of power saving protocols built into the cellular standard. Smartphones and other mobile devices typically use a capability called discontinuous reception (DRX), which allows the device to turn off its radio receiver for a period of time supported by the carrier network.

Similarly, the extended discontinuous reception (eDRX) protocol lets low power devices such as battery-operated asset trackers or other IoT devices specify how long they plan to sleep before checking back in with the network. By enabling eDRX operation, an LTE-M device can sleep up to about 43 minutes while an NB-IoT device can sleep up to about 174 minutes, dramatically extending battery life (Figure 4).

Another cellular operating mode, called power save mode (PSM), enables devices to remain registered with the cellular network even while they are in sleep mode and unreachable by the network. Normally, if a cellular network is unable to reach a device within some period of time, it will terminate the connection with the device and require the device to

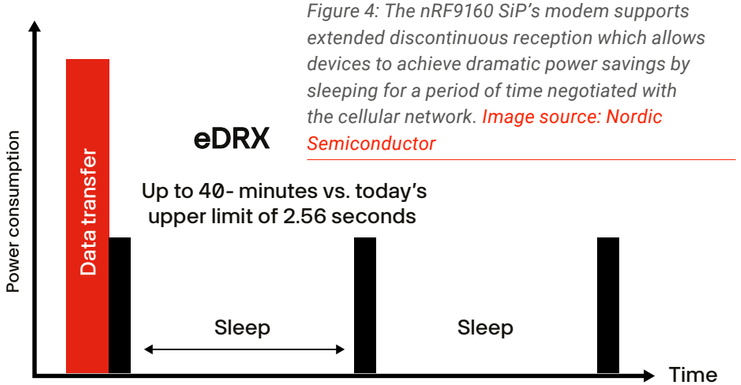


Figure 4: The nRF9160 SiP's modem supports extended discontinuous reception which allows devices to achieve dramatic power savings by sleeping for a period of time negotiated with the cellular network. Image source: Nordic Semiconductor

execute a reattachment procedure that consumes an incremental amount of power. During long-term operation of a battery-powered device, this repeated small consumption of power can exhaust or significantly reduce battery charge.

A device enables PSM by providing the network with a set of timer values that indicate when it will periodically become available and how long it will remain reachable before returning to sleep mode (Figure 5).

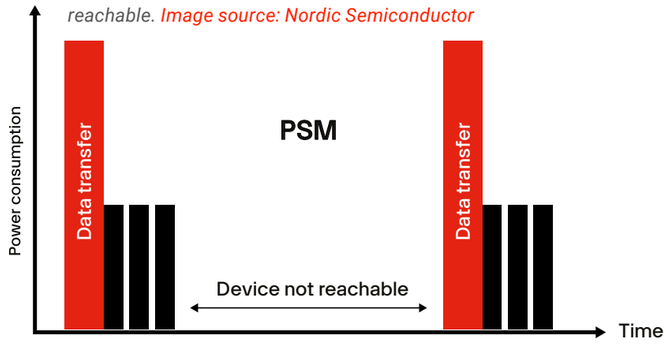
Because of the PSM negotiation, the carrier network does not detach the device. In fact, the device can wake at any time and resume communications. The benefit is that it uses its low power sleep mode when it has nothing to communicate without losing its ability to wake as needed and instantly communicate.

The nRF9160 SiP supports both eDRX and PSM, enabling the device to maintain operation with minimal power consumption. When in its unreachable stage with PSM, the device consumes only 2.7 µA. eDRX uses only slightly more current, consuming 18 µA in Cat-M1 operation or 37 µA in Cat-NB1 operation while using cycles of 82.91 seconds.

Developing low power asset tracking solutions

Implementing the hardware design for an asset tracking device based on the nRF9160 SiP requires few additional parts beyond decoupling components, antennas, and those needed for separate matching networks for GPS and LTE antennas (Figure 6).

Figure 5: The cellular PSM protocol allows devices to take advantage of low power sleep modes without incurring the power costs of reattachment by negotiating specific periods when they are not reachable. Image source: Nordic Semiconductor



Developers can easily combine the nRF9160 SiP with a Bluetooth device, such as Nordic Semiconductor's NRF52840 Bluetooth wireless microcontroller and sensors, to implement a sophisticated sensor-based GPS enabled cellular asset tracker that provides users with access to data through their smartphones and other Bluetooth enabled mobile devices.

Nordic Semiconductor further helps developers quickly begin evaluating cellular-based designs through a pair of development kits. For rapid prototyping of sensor-based asset tracking applications, the Nordic Semiconductor NRF6943 THINGY:91 cellular IoT development kit provides a complete battery-powered sensor system that pairs the nRF9160 SiP with an NRF52840 Bluetooth device, multiple sensors, basic user interface components, a 1400 milliamp-hour (mAh) rechargeable battery, and a SIM card to allow out-of-the-box cellular connectivity (Figure 7).

For custom development, the Nordic Semiconductor NRF9160-DK kit serves as an immediate development platform and reference for new designs. Although it does not include sensors like the THINGY:91, the NRF9160-DK kit combines an nRF9160 SiP with an NRF52840 Bluetooth device and includes a SIM card along with multiple connectors including a SEGGER J-Link debugger interface (Figure 8).

For software development of an asset tracking application, Nordic includes a complete nRF9160 asset tracking application with its nRF Connect software development kit (SDK). The SDK combines Nordic's nrfxlib software library for its SoCs, a Nordic fork of the Zephyr Project real-time operating system (RTOS) for resource constrained devices, and a Nordic fork of the MCUboot project secure bootloader.

The THINGY:91 and NRF9160-DK kits come preloaded with the asset tracking application designed to connect with Nordic's own nRF Cloud IoT platform. Using the preconfigured settings with either kit, developers can immediately begin evaluating cellular-based asset tracking and prototyping their own applications.

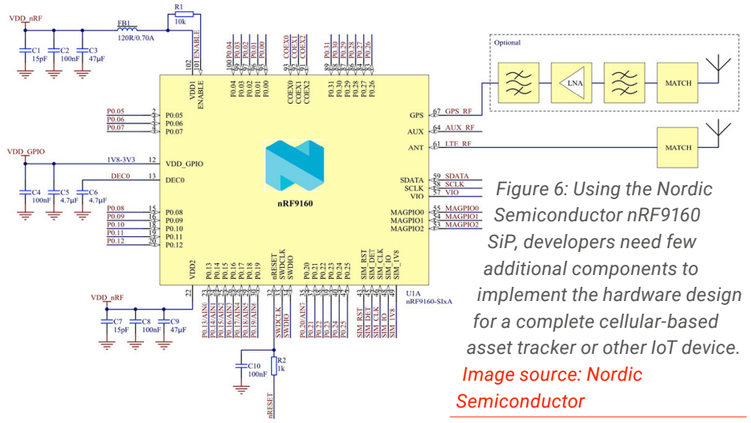
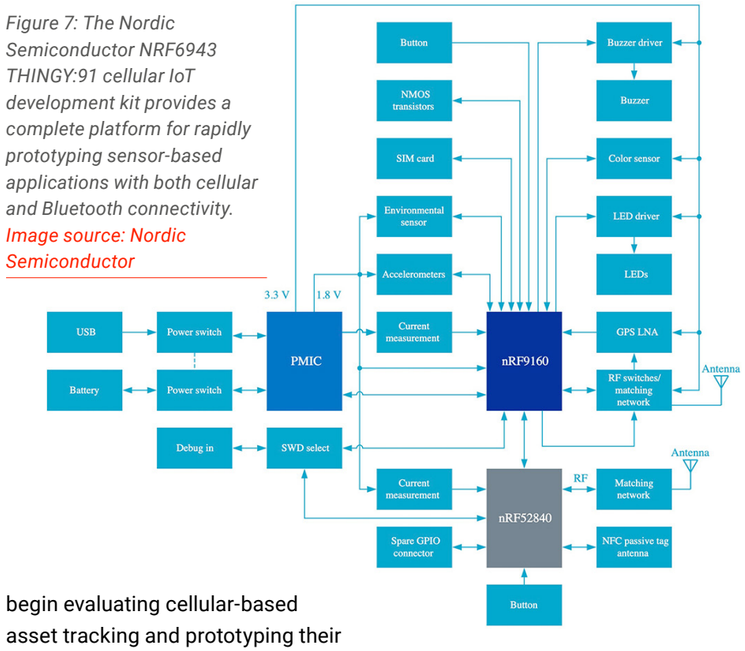


Figure 6: Using the Nordic Semiconductor nRF9160 SiP, developers need few additional components to implement the hardware design for a complete cellular-based asset tracker or other IoT device. Image source: Nordic Semiconductor

Figure 7: The Nordic Semiconductor NRF6943 THINGY:91 cellular IoT development kit provides a complete platform for rapidly prototyping sensor-based applications with both cellular and Bluetooth connectivity. Image source: Nordic Semiconductor



Along with the preloaded firmware, Nordic provides complete source code for the asset tracking application. By examining this code, developers can gain a deeper understanding of the NRF9160 SiP's capabilities, and its use in supporting GPS localization and LTE-M/NB-IoT connectivity in an asset tracking application.

The main routine in this sample software illustrates basic design patterns for implementing a custom asset tracking application. When started, the main routine invokes a series of initialization routines. Among those routines,

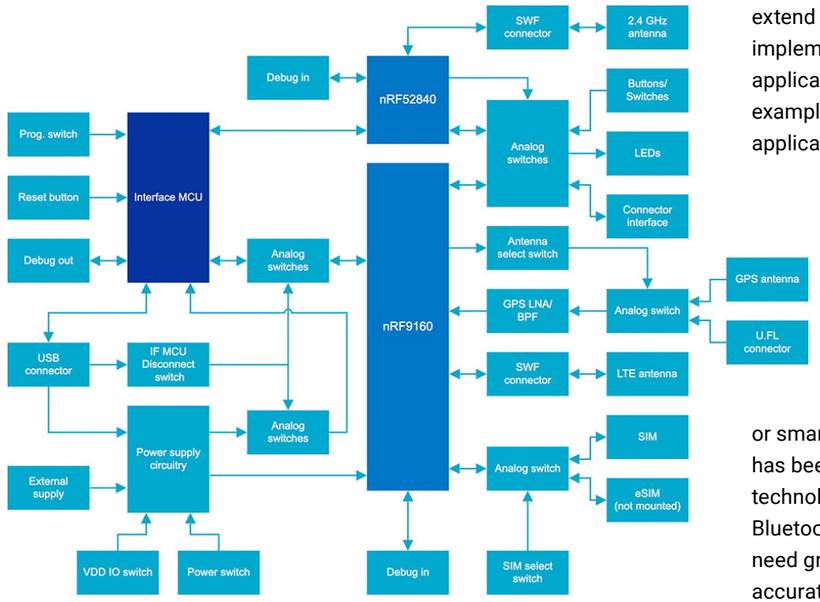


Figure 8: The Nordic Semiconductor NRF9160-DK kit offers a comprehensive development platform for implementation of custom cellular-based applications for asset tracking and other IoT solutions. Image source: Nordic Semiconductor

one initialization routine configures the modem and establishes the LTE connection by sending a series of attention (AT) strings to define connection parameters and invoke the modem's built-in functionality to connect to the carrier network. Another initialization routine, work_init, initializes a set of Zephyr RTOS work queues including those for sensor, GPS, and development board buttons (Listing 1).

During this initialization phase, the functions associated with each work queue initialization invocation perform their own specific initialization tasks, including those required to perform any required updates. For example, the sensors_start_work_fn function called by work_init sets up a polling mechanism that can periodically invoke a function, env_data_send, that sends sensor data to the Cloud (Listing 2).

When running the asset tracker sample application on the Nordic Semiconductor NRF6943 THINGY:91 cellular IoT development kit, the application sends actual data from the THINGY:91's onboard sensors. When running on the Nordic Semiconductor NRF9160-DK development kit, it sends simulated data using a sensor simulator routine included in the SDK. Developers can easily extend this software package to implement their own asset tracking applications or use its code examples to implement their own application architecture.

Conclusion

Using conventional methods, the ability to track valuable packages or locate high value assets across agricultural or smart city environments has been limited to wireless technologies such as RFID tags, Bluetooth, and Wi-Fi. Designers need greater range and more accurate location information over longer periods of time. Low-power LTE cellular standards like LTE-M or NB-IoT combined with GPS can meet these requirements, but implementation can be challenging due to the difficulty and nuances of RF design.

As shown, a Nordic Semiconductor SiP provides a near drop-in solution for long-range, low power asset tracking. Using this pre-certified SiP and its development kits, developers can quickly evaluate cellular connectivity, prototype cellular-based GPS enabled asset tracking applications, and build custom asset tracking devices that take full advantage of the extended range and low power requirements of LTE-M and NB-IoT cellular connectivity.

```
static void env_data_send(void)

{

[code deleted]

    if (env_sensors_get_temperature(&env_data) == 0) {

        if (cloud_is_send_allowed(CLOUD_CHANNEL_
TEMP, env_data.value) &&

            cloud_encode_env_sensors_data(&env_data,
&msg) == 0) {

                err = cloud_send(cloud_backend, &msg);

                cloud_release_data(&msg);

                if (err) {

                    goto error;

                }

            }

    }



    if (env_sensors_get_humidity(&env_data) == 0) {

        if (cloud_is_send_allowed(CLOUD_CHANNEL_
HUMID,

                        env_data.value) &&

            cloud_encode_env_sensors_data(&env_data,
&msg) == 0) {

                err = cloud_send(cloud_backend, &msg);

                cloud_release_data(&msg);

                if (err) {

                    goto error;

                }

            }

    }

[code deleted]
```

Listing 1: The Nordic asset tracker sample application builds on Zephyr RTOS utilities for queue management to create a series of queues with associated callback routines for handling various tasks such as sensor data acquisition and transmission to the Cloud. Code source: Nordic Semiconductor

Listing 2: The Nordic asset tracker sample application demonstrates the basic design pattern for transmitting data including sensor data as shown in this code snippet. Code source: Nordic Semiconductor

```
static void work_init(void)

{

    k_work_init(&sensors_start_work, sensors_start_
work_fn);

    k_work_init(&send_gps_data_work, send_gps_
data_work_fn);

    k_work_init(&send_button_data_work, send_
button_data_work_fn);

    k_work_init(&send_modem_at_cmd_work, send_
modem_at_cmd_work_fn);

    k_delayed_work_init(&send_agps_request_work,
send_agps_request);

    k_delayed_work_init(&long_press_button_work,
long_press_handler);

    k_delayed_work_init(&Cloud_reboot_work, Cloud_
reboot_handler);

    k_delayed_work_init(&cycle_Cloud_connection_
work,

                cycle_Cloud_connection);

    k_delayed_work_init(&device_config_work, device_
config_send);

    k_delayed_work_init(&Cloud_connect_work, Cloud_
connect_work_fn);

    k_work_init(&device_status_work, device_status_
send);

    k_work_init(&motion_data_send_work, motion_
data_send);

    k_work_init(&no_sim_go_offline_work, no_sim_go_
offline);

#if CONFIG_MODEM_INFO

    k_delayed_work_init(&rsrp_work, modem_rsrp_
data_send);

#endif /* CONFIG_MODEM_INFO */

}
```

How to select and use the right ESP32 Wi-Fi/ Bluetooth module

Written by:
Jacob Beningo
Contributing Author at DigiKey

Figure 1. The ESP32-WROOM-32D module runs at speeds up to 240 MHz and contains 8 Mbytes of onboard SPI flash. Credit: Espressif Systems

As the industrial automation accelerates, engineers on the factory floor are working to connect systems to an IoT that has in many ways left older factory floors behind. However, for both new and legacy systems, wireless connectivity to the IoT using Wi-Fi or Bluetooth has been made relatively simple using ESP32 modules and kits.

Created and developed by Espressif Systems, ESP32 – a series of low-cost, low-power system-on-a-chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth – is a breakthrough for automation engineers who don't want to get caught up in the nuances of radio frequency (RF) and wireless design. As a low-cost Wi-Fi/Bluetooth combo radio, it has gained popularity not just among hobbyists but also among IoT developers. Its low energy consumption, multiple open-source development environments, and libraries makes it perfectly suited for developers of all sorts.

However, ESP32 comes in so many different modules and development boards that it can be difficult to select the right one.

This article introduces ESP32 solutions and shows how developers can identify the right module and development board to start connecting their application to the IoT.

The ESP32 module

The ESP32 module is an all-in-one, integrated and certified Wi-Fi/ Bluetooth solution that provides not just the wireless radio but also an on-board processor with interfaces to connect with various

Figure 2. The ESP32-WROOM-32U is pin compatible with the WROOM-32D but replaces the latter's on-board trace antenna with an IPEX connector for an external antenna, allowing for optimized RF characteristics. Credit: Espressif Systems

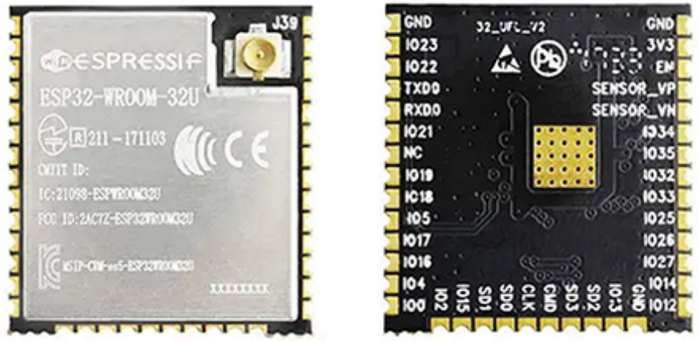


Figure 3. TThe ESP32-DEVKITC-32D-F development board includes breakout headers for connecting to any of the WROOM-32D pins and can be powered through USB for development purposes. Credit: Espressif Systems

peripherals. The processor actually has two processing cores whose operating frequencies can be independently controlled between 80 megahertz (MHz) and 240 MHz. The processor's peripherals make it easy to connect to a range of external interfaces such as:

- SPI
- I2C
- UART
- I2S
- Ethernet
- SD Cards
- Capacitive touch

There are several different ESP32 modules that a developer can select based on their application needs. The first and most popular ESP32 module is the ESP32-WROOM-32D, which runs at up to 240 MHz (Figure 1). The module includes a PC board trace antenna, which simplifies implementation. It also avoids having to add the additional hardware and layout complexity associated with an IPEX connected antenna. However, if the IPEX connector option is selected, there are plenty of good antenna options, such as Inventek Systems' W24P-U.

The module contains 4 megabytes (Mbytes) of flash and has 38 pins that are arranged to minimize the module's size, making it nearly square. In fact, the WROOM-32D is completely pin compatible with the ESP-WROOM-32U (Figure 2). The WROOM-32U replaces the onboard PC board trace antenna with an IPEX connector, based on the Hirose U.FL design. In doing so, the WROOM-32U saves board space and allows developers to connect an external antenna that they can arrange within their product for optimal RF characteristics.

An interesting point about the WROOM-32D modules is that they also come in various flash memory sizes. The modules come in additional memory support variants like the ESP32-WROOM-32D with 8 Mbytes and the ESP-WROOM-32D with 16 Mbytes.

Selecting an ESP32 development board for industrial control

The ESP32 modules are a great choice when designing a board that will be used in production or where they will be put on a board that will be used in 'high' volume.

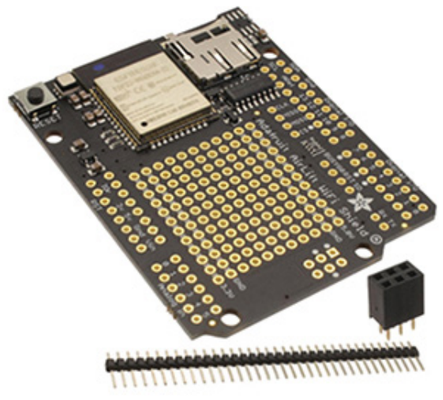


Figure 4. The Adafruit Airlift ESP32 Shield allows designers to prototype their design or build one-off circuits that can be used in industrial automation applications. The Airlift includes prototyping space that can be used for dedicated circuitry. Credit: Espressif Systems

For development of low-volume fixtures on the manufacturing floor, developers can use an ESP32 development board. These boards range from very basic 'getting started' boards to sophisticated boards that include secondary processors and LCDs. There are some that are also well suited for industrial automation applications, assuming simplicity of development is a key requirement.

For instance, there's the ESP32-DEVKITC-32D-F (Figure 3). This is a simple breakout board for the WROOM-32D that has all the power conditioning and programming circuits a designer or developer needs to get started. The board is powered either through an on-board USB micro connector or through the V-IN breakout header. Jumpers or wires can then be used to connect various components to the WROOM-32D.

Another example is the Adafruit Industries Airlift ESP32 Shield. This not only includes the WROOM-32D, but also has additional prototyping space (Figure 4). This prototyping space can be used to add connections to other shields in addition to adding custom circuitry. A developer could use this area to build input and output circuits for low voltage industrial automation applications. There is also an onboard SD card connector that makes developing a data logging application that much easier.

There may be some industrial automation applications where a development board with an additional processor is being used and the ESP32 will just be providing connectivity rather than handling the whole application load. In these applications, the development board or product may have expansion PMOD connectors onboard.

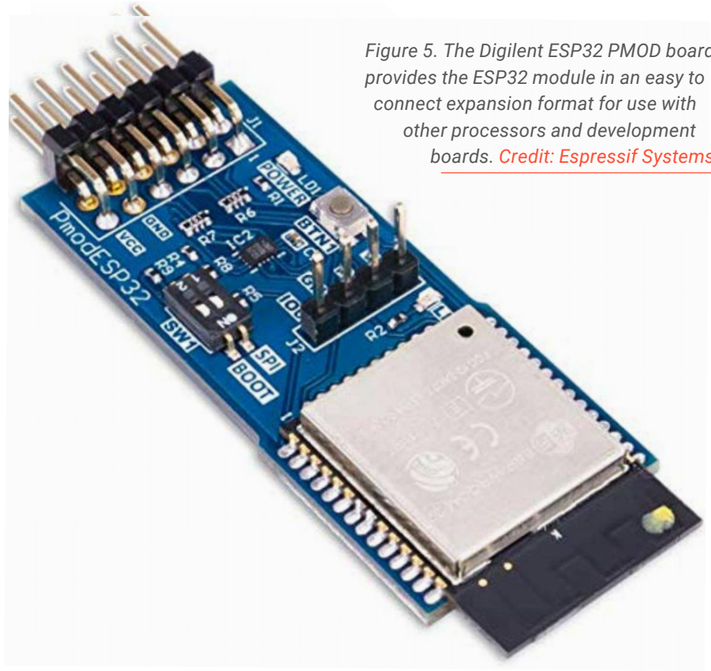


Figure 5. The Digilent ESP32 PMOD board provides the ESP32 module in an easy to connect expansion format for use with other processors and development boards. Credit: Espressif Systems

Rather than custom designing a PMOD board for the ESP32, developers can leverage the Digilent ESP32 PMOD breakout board (Figure 5).

The ESP32 PMOD provides a PMOD standard connector along with the following:

- A power LED indicator
- An on-board user button
- Four pin I/O expansion
- Jumpers for boot configuration

The Espressif Systems ESP-WROVER-KIT provides a full ESP32 development solution with everything designers need to develop an ESP32-based application (Figure 6). For example, the WROVER includes an FT2232HL USB to serial converter from FTDI which makes it easy to program the ESP32 module without the need for custom programming tools. The board also includes an onboard 3.2 inch LCD, a microSD connector, an RGB LED and a camera interface. The development board also as all the I/O lined up and made easily accessible through pin headers.

ESP32 – a series of low-cost, low-power system-on-a-chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth – is a breakthrough for automation engineers who don't want to get caught up in the nuances of radio frequency (RF) and wireless design.

Once a designer has decided which module and development board best fit their application, they need to spend some time looking at the development environment for the ESP32 that best fits their needs.

Selecting an ESP32 development environment

The ESP32 has become so popular that there are several different development environments to choose from to develop and program the device. The most popular development tools include:

- The Espressif IoT Development Framework (IDF)
- Arduino IDE
- MicroPython

The first environment, the Espressif IDF, is a development toolchain for experienced embedded software developers. The toolchain includes several useful pieces such as an IDE to develop the application, a compiler, libraries, and examples. The IDF uses FreeRTOS as the base real-time operating system (RTOS) along with the lwIP TCP/IP stack and TLS 1.2 for Wi-Fi.

For developers who have minimal programming experience, the popular Arduino IDE can also be used to develop an application and deploy it to the ESP32. While the Arduino IDE is a bit slower and clunkier than a professional development environment, it offers a lot of examples and support for the ESP32, which can make development for a newbie much easier.

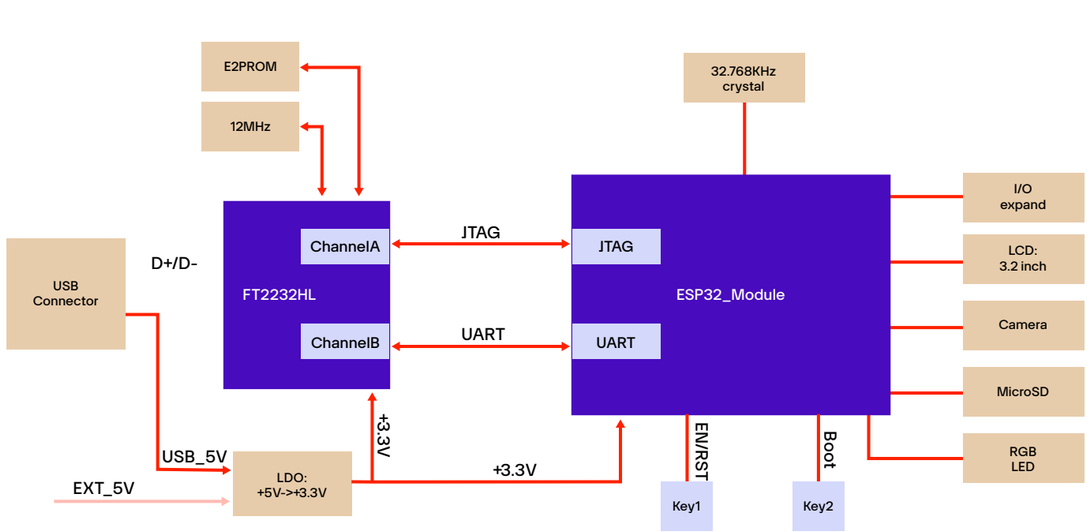


Figure 6. The Espressif ESP-WROVER-KIT board provides industrial automation developers with an ESP32 module that has access to an RGB LED, microSD slot, camera, an LCD, and easily accessible I/O expansion.
Credit: Espressif Systems

Finally, for developers who are interested in developing their application in Python, the ESP32 is supported by the open source MicroPython kernel. Developers can load MicroPython onto the ESP32 and then develop Python scripts for their application. This can make it very easy to update the application on-the-fly in an industrial setting and remove layers of required expertise that normally come with embedded development.

Tips and tricks for working with ESP32

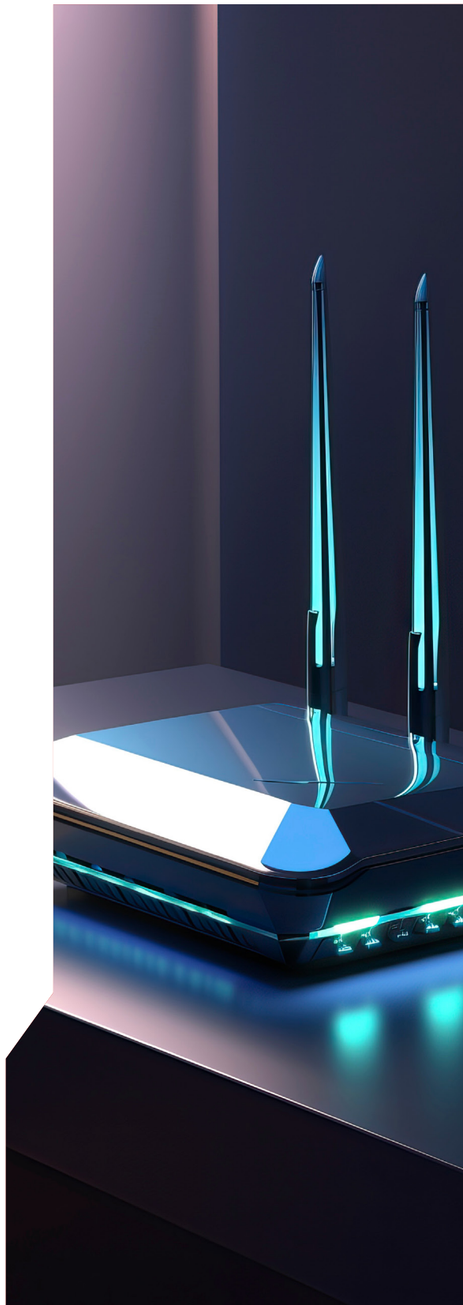
Getting started with ESP32 is not difficult, and a search of the web will provide detailed descriptions of how to set up the various software environments. That said, there are many nuances and decisions required of developers working with ESP32 for the first time. Here are a few 'tips and tricks' for getting started:

- Carefully identify and configure a module's boot pins – MTDI, GPIO0, GPIO2, MTDO and GPIO5 – to load an application from the correct memory source (internal flash, QSPI, Download, Enable/Disable debug messages)
- Set the serial output baud rate to the same baud rate as the ESP32 boot firmware baud rate. This will allow monitoring of the ESP32 boot messages, and the application debug messages, without reconfiguring the baud rate
- Users that don't have embedded programming experience should 'flash' MicroPython onto the ESP32 so that the application code can be written in the easy to learn Python scripting language
- For the application, search the Internet for ESP32 examples and libraries to accelerate application development and integration (there are a lot of great examples already available)
- In the design, make sure that the boot strapping pins are able to be used to boot into the update mode. This will make it very easy to update firmware in the field
- Developers that follow these 'tips and tricks' will find that they can save quite a bit of time and grief when working with ESP32 for the first time.

Conclusion

As shown, ESP32 has several different modules and development boards that developers can leverage to begin designing their industrial IoT application. The advantage of using ESP32 for this purpose is that it simplifies development by removing the need to understand RF circuitry and to certify the wireless receiver. ESP32 is also widely supported, not just by the module manufacturer but also within professional and hobbyist circles. Developers who are not familiar with embedded software can easily use the Arduino IDE or program their wireless application using MicroPython.

All told, ESP32 is an excellent choice for connecting industrial automation equipment quickly and efficiently.



IoT security fundamentals: connecting securely to IoT Cloud services

Contributed By:
Stephan Evanczuk,
Contributing Author at Digikey

Internet of Things (IoT) security depends on multiple layers of protection extending from the IoT device's hardware foundation through its execution environment. Threats remain for any connected device, however, and typical IoT application requirements for Cloud connectivity can leave both IoT device and Cloud services open to new attacks. To mitigate these threats, IoT Cloud providers use specific security protocols and policies which, if misused, can leave IoT applications vulnerable.

Using preconfigured development boards, developers can quickly gain experience with the security methods used by leading IoT Cloud services to authenticate connections and authorize use of IoT devices and Cloud resources.

This article describes the connection requirements of two leading Cloud services, Amazon Web Services (AWS) and Microsoft Azure, and shows how developers can use development kits and associated software from a variety of vendors to quickly connect with these services.

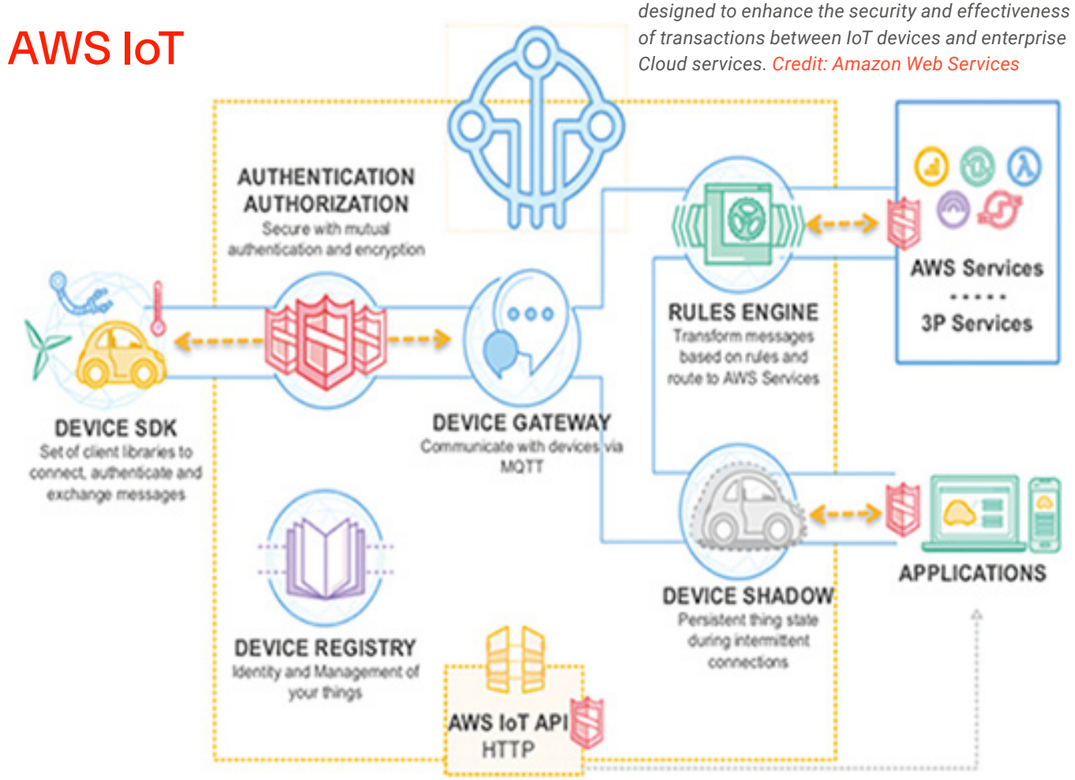


Figure 1. As with other Cloud providers, AWS provides developers with a set of specialized services designed to enhance the security and effectiveness of transactions between IoT devices and enterprise Cloud services. Credit: Amazon Web Services

The role of IoT portals in Cloud services

When an IoT device connects to a resource such as a Cloud service or remote host, it potentially exposes itself – and by extension the entire IoT network – to threats masquerading as legitimate services or servers. Conversely, the Cloud service itself similarly faces the threat of attacks from hackers mimicking IoT device transactions in an attempt to penetrate Cloud defenses. To help ensure protection for both IoT devices and Cloud resources, Cloud services require the use of specific security protocols for mutual authentication of identity for sign in and subsequent authorization to ensure permitted usage of services. These protocols are typically included in a set of services that provide a secure portal between IoT devices and Cloud resources.

As with other available IoT Cloud service platforms, AWS, and Azure each provide a specific entry portal that IoT devices need to use to interact with each provider's full set of Cloud resources such as virtual machines (VMs) and software-as-a-service (SaaS) offerings. Using a functionally similar set of mechanisms and capabilities, Azure IoT Hub and AWS IoT provide this portal for their respective enterprise Cloud offerings.

At a minimum, these and other IoT portals use specific authentication protocols implemented through each provider's software development kit (SDK) to create a secure connection. For AWS,

we get technical

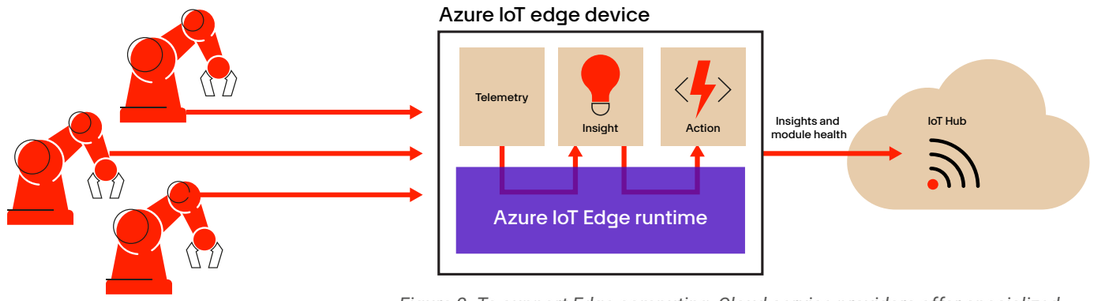


Figure 2. To support Edge computing, Cloud service providers offer specialized services such as Microsoft Azure IoT Edge, which brings some Azure IoT Cloud services closer to the physical devices associated with the IoT application. Credit: Microsoft Azure

IoT devices connect using mutual authentication with a device gateway. The device gateway connects the IoT device with other IoT support services, using information held in a device registry to store a unique device identifier, security credentials and other metadata needed to manage access to AWS services (Figure 1). In Azure, the identity registry serves a similar function.

AWS IoT and Azure IoT each provide a service that maintains state information in a virtual device associated with each physical IoT device. In AWS IoT, device shadows provide this capability for AWS IoT, while device twins provide similar capabilities for Azure IoT.

This notion of a security portal extends to IoT Edge services such as AWS Greengrass or Azure IoT Edge. These Edge service offerings bring some Cloud services and capabilities down to the local network, where Edge systems are placed physically close to IoT devices and systems in large-scale deployments. Developers can use a service such as Azure IoT Edge to implement application business logic or provide other functional capabilities needed to reduce latency or provide services to local operators, such as in industrial automation (Figure 2).

Dealing with requirements for IoT portal connectivity

Whether connecting through an Edge system or directly with the provider's IoT service, IoT devices typically need to satisfy a series of requirements to connect through the provider's IoT portal and use the provider's Cloud services. Although the details differ, IoT devices need to provide at a minimum, some item such as a private key, X.509 certificate, or other security token. The key, certificate, or token provides the IoT portal with attestation or proof of the IoT device's identity during the authentication phase of the device-Cloud connection sequence. In addition, IoT Cloud services typically require a specification of the policies used to define access rights for interactions between IoT devices and Cloud services.

As with other enterprise computing requirements, attestation information for authentication and policy information for access management need to be provided using specific formats and procedures specified by leading IoT Cloud services like AWS IoT and Azure IoT. These services support certificate-based authentication at a minimum, but also support use of other forms of attestation. For example, developers can use token-based authentication methods based on a JSON Web Token (JWT) for AWS IoT, or a shared access signature (SAS) token for Azure IoT.

As mentioned earlier, these services use registries to hold metadata for each IoT device. Along with security and other information, these registries store the access rights policies that need to be defined to connect an IoT device. Although specified in different ways for different Cloud services, these policy definitions describe access rights for different communications channels and entities. For example, a simple AWS IoT access right policy would use a JSON format to indicate that an IoT device with a particular 'thing' name in the AWS IoT device registry could connect and publish messages only on a channel with the same associated thing name (Listing 1).

Listing 1: Developers use a JSON format to describe AWS IoT access rights policies for their IoT devices. Credit: AWS

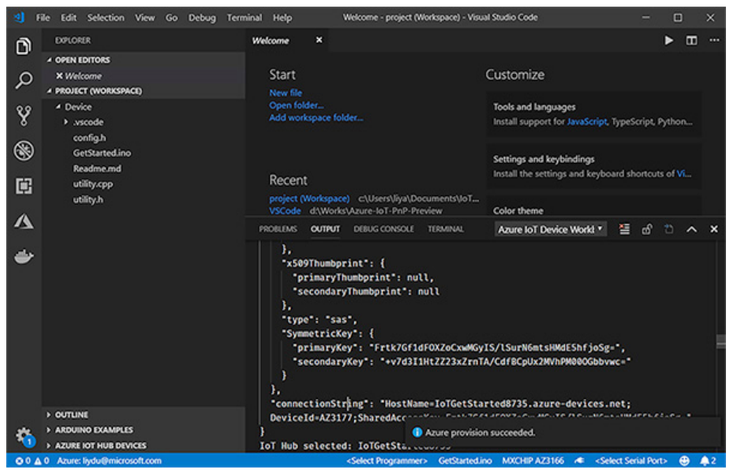
```
{
  "Version": "2012-10-17",
  "Statement": [
   {
    "Effect": "Allow",
    "Action":["iot:Publish"],
    "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${iot:Connection.Thing.ThingName}"]
   },
   {
    "Effect": "Allow",
    "Action": ["iot:Connect"],
    "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"]
   }
  ]
}
```

Cloud-ready development kits

Although Cloud providers offer detailed specifications of those formats and procedures, the providers' support forums are typically filled with questions from developers tripped up by some small but vital detail that prevents authentication and access management. Perhaps even worse, from a security standpoint, unintended misuse of attestation or incomplete definition of access policies can leave IoT devices, networks, and applications open to attack. The availability of off-the-shelf development boards and accompanying software packages allows developers to quickly navigate through these connection procedures, using vendor-provided examples to rapidly connect to IoT Clouds. For example, Espressif Systems' ESP32-Azure IoT Kit or Seeed Technology's AZ3166 IoT Developer Kit include Azure-certified boards designed to connect easily with the Microsoft Cloud.

Microsoft provides complete step-by-step demonstrations, including authentication and access credentials for the supported development kits. With the AZ3166 board, for example, developers press buttons on the board to initiate connection with their local Wi-Fi network. Once connected, they can use the Azure IoT Device Workbench contained within the Azure IoT Tools extension pack for Microsoft Visual Studio Code to develop, debug, and interact with the Azure IoT Hub. Using

Figure 3. Sample code and credentials provided in the Microsoft Azure IoT Device Workbench help developers complete provisioning for connecting the Seeed Technology AZ3166 IoT Developer Kit to Azure IoT Hub. Credit: Microsoft Azure



this toolset and its sample code packages, developers create an object for the IoT device in Azure IoT Hub and use a provided file to provision the associated identity registry with the credentials and other metadata required to connect the IoT board to Azure IoT Hub (Figure 3).

The Azure IoT Device Workbench provides additional support software and metadata that lets developers quickly load the AZ3166 board with sample code and begin transmitting measurements from the board's temperature and humidity sensor to Azure IoT Hub.

The steps involved in creating a representation for the physical IoT device in the IoT Cloud and for provisioning the associated registry are needed just to connect devices with the IoT Cloud. To take advantage of Cloud services, however, the Azure IoT Hub needs an access rights policy. To monitor the device-to-Cloud messages coming from the AZ3166 sensor, developers can simply use the Azure shared access policies screen to select a prebuilt policy designed to quickly enable the required access rights (Figure 4).

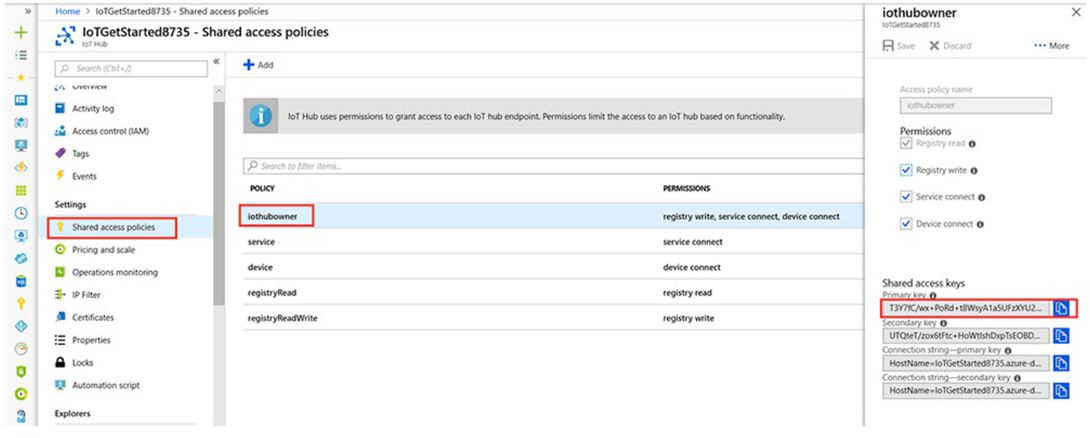
When working with AWS IoT, developers can turn to development kits such as Microchip Technology's AT88CKECC-AWS-XSTK-B Zero Touch Provisioning kit and accompanying software to quickly evaluate Cloud connectivity. This updated version of an earlier Microchip Zero Touch Provisioning kit comes preloaded with authentication credentials. Using additional scripts provided with the kit, developers can rapidly connect the board to AWS IoT without dealing with private keys and certificates (see, 'Take the Zero-Touch Approach to Securely Lock Down an IoT Device').

Other development kits, including Renesas' RTK5RX65N0S01000BE RX65N Cloud Kit and Infineon Technologies' KITXMC48IOTAWSWIFITOBO1 AWS IoT kit, extend support for AWS IoT connectivity with support for rapid development of applications based on Amazon FreeRTOS. AWS provides detailed directions for registering the boards, creating authentication credentials, and loading provided JSON policies needed to connect to AWS IoT and use AWS services.

Simplifying provisioning for large-scale IoT deployments

Development kits such as those described above serve as effective platforms both for rapid prototyping of IoT applications and for exploring IoT Cloud service connection requirements. In practice, however, developers will typically need to turn to more advanced approaches designed to simplify provisioning of IoT devices in real-world applications. Both Azure IoT and AWS IoT support a wide variety of methods that allow more automated provisioning of individual devices or large numbers of IoT devices in a large-scale deployment.

Figure 4. Developers can use prebuilt policies to easily authorize use of Azure Cloud services with sensor data from the Seeed Technology AZ3166 IoT Developer Kit. Credit: Microsoft Azure



service connection requirements. In practice, however, developers will typically need to turn to more advanced approaches designed to simplify provisioning of IoT devices in real-world applications. Both Azure IoT and AWS IoT support a wide variety of methods that allow more automated provisioning of individual devices or large numbers of IoT devices in a large-scale deployment.

With AWS IoT, for example, developers can use a bootstrap method for certificate provisioning. Here, the smart product ships with a bootstrap certificate associated with the minimal access rights needed to request and access a new certificate (Figure 5).

Using the bootstrap certificate, the device connects to the Cloud ('1' in Figure 5), requests ('2') a new certificate, receives ('3') the URL of the certificate generated by an AWS serverless Lambda function, and retrieves ('4') that certificate from an AWS Simple Storage Services (S3) bucket. Using that new certificate, the device then logs back into AWS IoT ('5') to proceed with normal operations.

AWS offers other Cloud services that support dynamic provisioning of authentication tokens using execution resources like AWS Lambda functions. For example, an automotive application might rely on a series of ephemeral connections where use of a token is both more practical and more secure. Here, after an AWS module for IoT authentication and authorisation approves the request for a token, the AWS Security Token Service (STS) generates a token for delivery to the vehicle's systems. Using that token, those systems can access AWS services subject to validation by the AWS Identity and Access Management (IAM) service (Figure 6).

AWS provides a similar capability

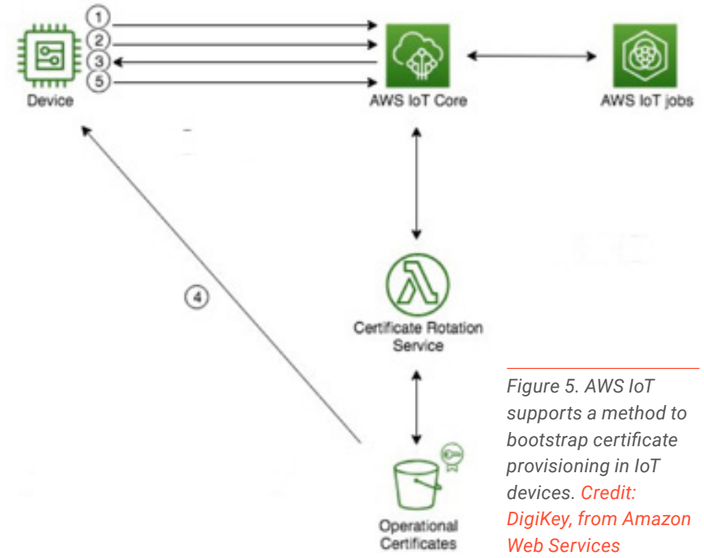


Figure 5. AWS IoT supports a method to bootstrap certificate provisioning in IoT devices. Credit: DigiKey, from Amazon Web Services

Figure 6. Leading Cloud service providers support other forms of attestation for authentication such as this process for dynamic generation of security tokens by AWS Security Token Service (STS). Credit: Amazon Web Services

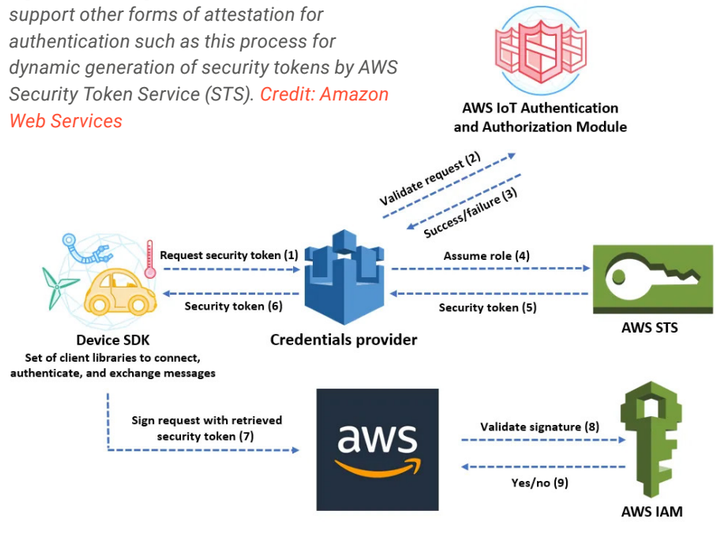


Figure 7: Developers can use Cloud services to implement dynamic assignment of access rights, which is particularly useful for applications with ephemeral connections or short-lived operations. Credit: Amazon Web Services

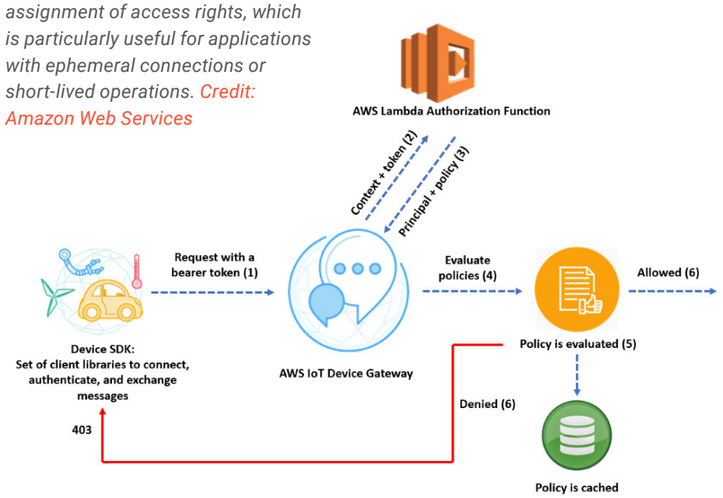
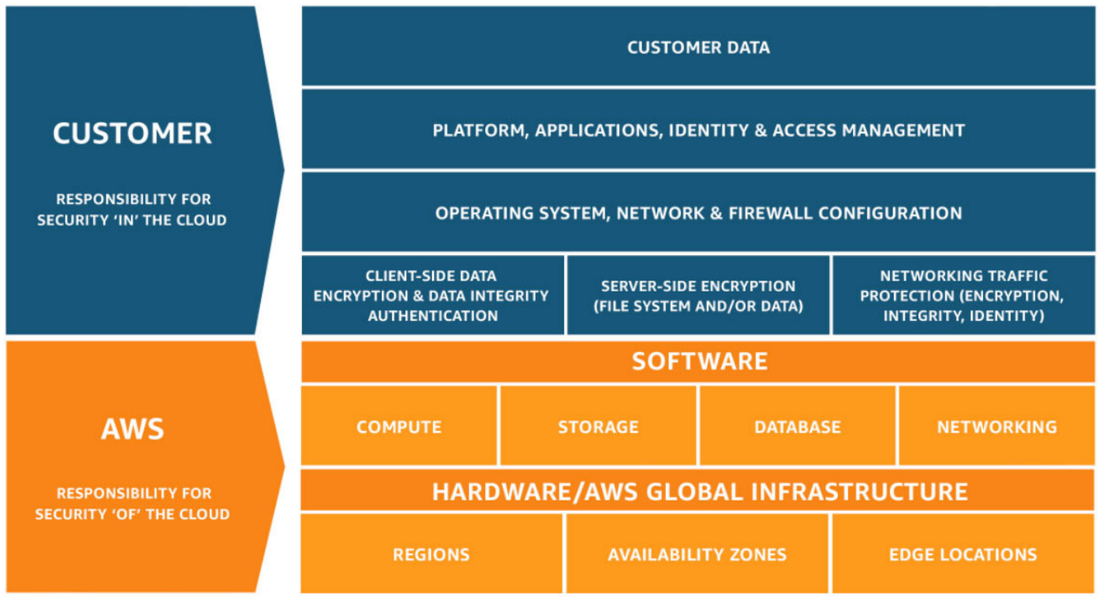


Figure 8. As with other leading Cloud providers, AWS describes the responsibilities that it shares with Cloud users to protect the Cloud infrastructure on one hand and customer applications on the other. Credit: Amazon Web Services



for dynamic assignment of access rights. Here, other AWS Lambda functions would assign a set of policies associated with a valid token (Figure 7).

Other IoT Cloud services allow developers to more efficiently deal with provisioning in large-scale deployments. For example, AWS IoT provides fleet provisioning capabilities, including support for a larger scale deployment of the kind of bootstrap method described earlier. Azure IoT's Device Provisioning Service provides a group enrolment capability that supports provisioning of large numbers of IoT devices that share the same X.509 certificate or SAS token.

Shared responsibility for security

IoT Cloud providers provide a number of effective methods for enhancing end-to-end security for IoT applications. Nevertheless, IoT developers cannot expect that those methods can bear the full weight of security requirements for their particular IoT application. In fact, Cloud service providers carefully outline their specific role and responsibilities in IoT application security with specific models such as AWS' shared responsibility model (Figure 8).

AWS and Microsoft Azure each provide shared responsibilities documents that describe and explain the provider's own role and that of the customer in securing resources, data, and applications. In its documentation, Microsoft also offers an overview of some of the relationships between shared security and compliance requirements. Ultimately, Cloud providers retain responsibility for the security of the Cloud, while customers remain responsible for applications, data, and resources used in the Cloud.

Conclusion

IoT applications depend on layers of security built up from hardware-based mechanisms for cryptography and secure key storage. As with any connected product, security threats can arrive in all manner of interactions when IoT devices connect with Cloud services. To protect themselves and their customers, IoT Cloud providers dictate specific requirements for authentication and access rights management. Although providers offer detailed documentation on those requirements and associated specifications, developers can find that their efforts to implement secure connectivity sometimes leave resources exposed, or conversely, inaccessible. Using development boards and associated software, developers can quickly connect to Cloud services and rapidly prototype IoT applications with end-to-end security.

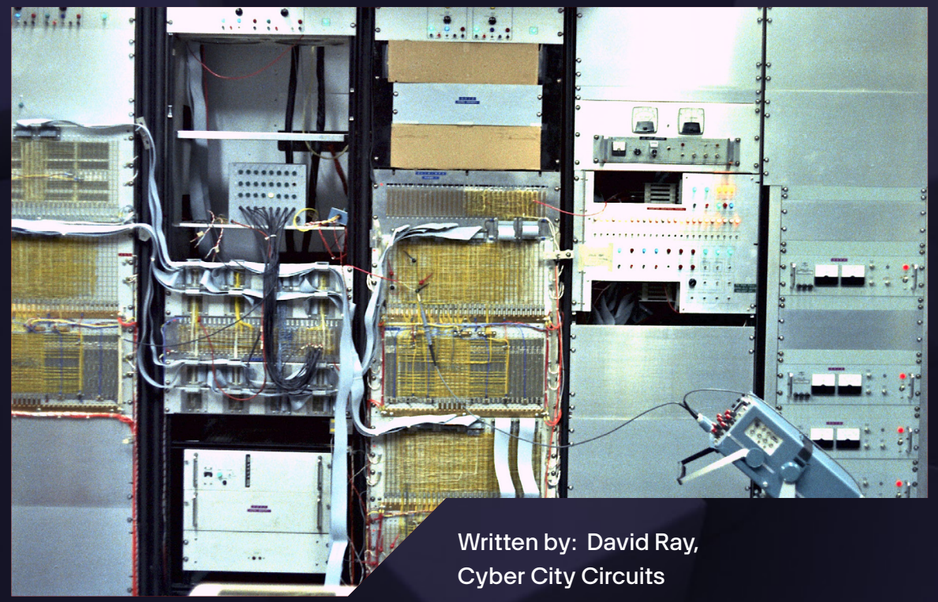


Figure 1: Image of the BCC-500 from an unknown time. ***Image source: Al Kossow and gunkie.org***

Written by: David Ray, Cyber City Circuits

Retro Electro: The ALOHA System: Task II

Introduction

Modern computing and networking history is rich with projects and innovations that laid the foundation for the information age, from ARPANET to the World Wide Web. Among these landmark projects, the BCC-500 stands out as a near-forgotten computer that was a key to developing the first wireless packet-switched computer network with the ALOHA system. One of a kind and built by hand, the system was the first wireless access ARPANET node, starting in 1974 until its decommissioning in 1980.

This is the story of the BCC-500.

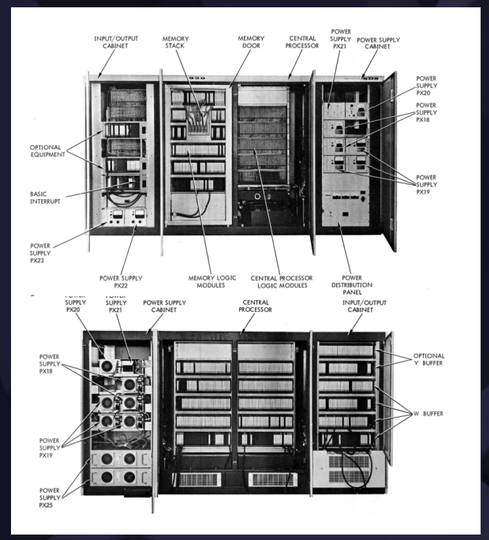
1960s - the GENIE, time share and The Cold War

The story of the BCC-500 begins in the 1960s with Project GENIE at the University of California, Berkeley, a system that revolutionized time-share computing and significantly influenced the development of computer networks.

In 1963, J.C.R. Licklider, a director in ARPA (Advanced Research Projects Administration), started funding the electrical engineering department at the University of California, Berkeley under a program called 'Project GENIE.' Graduate student Mel Pirtle started the program by developing a way for multiple people to use a computer simultaneously. The issue was that early computers could only have one user at a time. A user would need to submit a batch of requests and wait for their turn for their work to be processed and returned.

The SDS-930 was a computer designed by Scientific Data Systems in Santa Monica, California. The technical manual includes this description: 'The 930 Computer is a general-purpose, solid-state, digital computer designed for scientific and engineering computation and real-time applications.' Weighing up to 2,750 pounds and commanding a nearly 200 square-foot footprint, it was a marvel of human engineering while being a monster of a machine. When it arrived at Berkeley in September 1964, the machine was used to develop the first successful time-sharing system.

Figure 2: SDS-930 Front and Rear View
Image source: SDS-930 Technical Manual



A time-sharing system is a method that allows many users to share access to a computer system at the same time. It works by rapidly switching between users, assigning each a brief time slot to interact with the computer. This means the system can take instructions from one user, execute them for a few milliseconds, and then move on to the next user, repeating this process so quickly that it seems like all users are working simultaneously. This efficient use of computing resources makes it possible for multiple people to use the same computer without significant delays.

In a demonstration, Pirtle typed instructions and queries continuously for forty minutes, but the system only registered fifty-six seconds of computer time with dozens of other concurrent users. While this sounds trivial today, this was the real cutting-edge technology.

During the Cold War, developing time-sharing systems like Project GENIE had significant strategic importance. It allowed military and research institutions to maximize their computing resources, enabling rapid data processing and real-time simulations crucial for defense research and development. This technological edge provided the United States with enhanced capabilities in various fields, from cryptography to missile guidance, thus contributing to the critical technological superiority during the Cold War era.

1968 - Berkeley Computer Corporation

Once completed, the changes made to the SDS-930 to create the powerful time-share system found their way back to Scientific Data Systems, which implemented them in an upgraded model called the 'SDS-940'. First shipping in April 1966, this machine became the company's most successful computer, earning $40 million in sales. This caught the attention of Xerox, which purchased the company in May 1969. Xerox continued to sell this system with the model number 'XDS-940'.

The VP for Technology Strategy, Robert Spinrad, is quoted as saying, "Project Genie was the earliest useful realization of timesharing on a minicomputer. Their computer differed from earlier systems in that those were built on large, mainframe

computers. This system was attractive to SDS (Scientific Data Systems) because they made minicomputers and had been thinking of getting into the timesharing business."

When Project GENIE formally ended in 1968, the team split up in various ways mainly staying within academia and research, with some going to companies like Xerox, and a few took the route of entrepreneurship.

Dr. Melvin Pirtle, Dr. Butler Lampson, Chuck Thacker, and Alan Kay wasted little time and started their own company, the Berkeley Computer Corporation (BCC). Building on the successes of Project GENIE, BCC was formed with the goal of designing the BCC-500, a computer meant to push the boundaries of time-share computing even further.

Figure 3: Help Wanted Ad (November 16, 1969)


Berkeley Computer Corporation offers stock options, relocation assistance, an easy freeway commute from anywhere in the Bay Area and close proximity to the campus of the University of Calif. Interested? Qualified? Call collect! If unable to call, mail your resume to Mr. W. L. Harrison, Director of Administration, Berkeley Computer Corporation, 2398 Fourth Street, Berkeley, Calif. 94710. An equal opportunity employer.

BERKELEY COMPUTER CORPORATION

. . . A company growing in the right direction.

BCC was founded on December 19, 1968, and remained in business until January 1971. Its primary purposes were manufacturing and marketing large-scale time-sharing computer systems and related equipment and offering time-sharing services on BCC-owned systems. The original staff consisted of fourteen senior-level programmers, eight senior-level engineers, and many supporting technicians, draftsmen, and programmers.

Pirtle, the first Principal Investigator at Project GENIE, was the company's President, Lampson was the head system designer, and Chuck Thacker was the engineering project leader. An internal document named 'Supervisor's Manual' is available (and listed in the Suggested Reading section) that lists all the personnel with the company in October 1969.

Some remarks on a large new time-sharing system

Prior to going out of business, in what could have been a last-ditch effort to sell the BCC-500 prototype and stay in business, Lampson authored a report titled 'Some Remarks on a Large New Time-Sharing System'. This document is the earliest comprehensive overview of the BCC-500.

In this report, he lists BCC's four primary characteristics they had in mind when designing this system:

- Efficiency, obtained through specialized hardware
- Reliability, which depends on redundancy, error-checking, and good protection mechanisms
- Modularity in both hardware and software
- High-level Language Programming

Here, we find the claim that the BCC-500 could support up to five hundred concurrent time-sharing users. He also discussed the machine's unusual configuration. It consisted of two central processors, several small processors, and a rotating magnetic memory system, which allowed them to swap 250 users in and out in one second.

While this sounds like it would be impressive for the time, he continues to say:

'The Model 500 is not an unusually fast machine when measured in instructions per second, nor does it use extremely fast logic. It is the specialization of comparatively modest amounts of hardware for particular purposes, which make its overall efficiency high', arguing that even though many of the machine's specifications were worse compared to the

The BCC-500 was designed to get power from a motor generator so that brown-outs from the utility company would not affect it. Today, you would just use a $50 "battery backup" from the Digikey catalog.

competition, it is an application-specific design, comparable or better in efficiency.'

The BCC-500 would have shipped with a working FORTRAN IV and eBASIC compiler and an SDS/XDS-940 emulator. Its operating system was a highly machine-oriented program with more than 10,000 discrete instructions, which contained fewer than one hundred discrete machine instructions. This would have made writing software for the system easier and quicker.

The firmware was designed so that a user would get between 500ms and 750ms of allocation every five seconds. The 750ms limit was there so that the user wouldn't notice a wildly different quality of service outside of peak usage time.

1970 – financial difficulties

The recession of 1970 struck, and like many other new small businesses, BCC was feeling it. Early on, the company was able to raise two million dollars from Data Processing Financial & General Corporation (DPF&G), but that would soon come to an end.

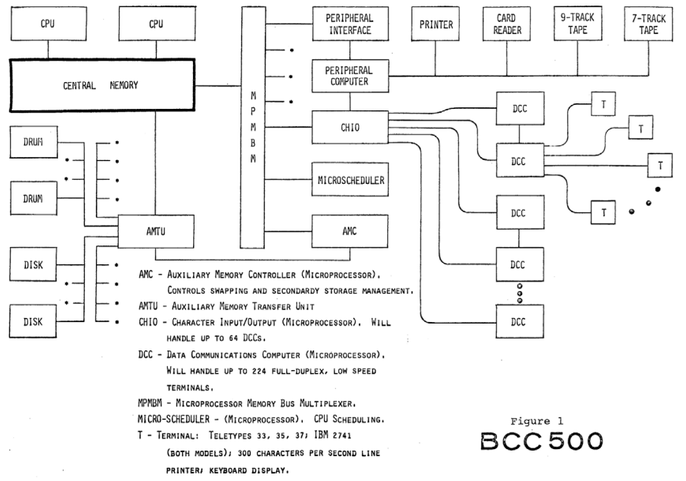


Figure 4: BCC 500 Block Diagram (Some Remarks on a Large New Time-Sharing System)

In April, Pirtle obtained a $1M investment from the University of California, Berkeley's investment portfolio. This investment was met with the slightest controversy. Recently, the University made some poor decisions and came under scrutiny for misuse of funds. At this same time, DPF&G was in an antitrust lawsuit with International Business Machines (IBM).

DPG&G invested money in developing the BCC-500 to avoid purchasing more IBM equipment in the future. When the lawsuit had settled and the BCC-500 wasn't completed towards the end of 1970, DPF&G

These are the same effects of the 1970 recession that affected Hans Camenzind and Signetics in Volume 5 of 'We Get Technical'. Check it out.

stopped funding BCC, deciding to stick with IBM equipment. With this they lost a contract for mainframe leasing with DPF&G along with additional funding.

When the Berkeley Computer Corporation closed in January 1971, the members split up. Many of the top talents went on to Xerox PARC to invent the first modern computer, the Xerox Alto, while some others followed the BCC-500 on its fated Hawaiian trip.

1968: beginnings of the ALOHA System at the University of Hawaii

In the 1960s, the University of Hawaii consisted of half a dozen campuses spread across many islands and several research institutes with researchers all around Hawaii, within a 200-mile radius of Honolulu. With the increasing availability of mainframes and mini-computers, the University purchased an IBM 360/65 to use with a time-sharing system, like many other Universities of the day.

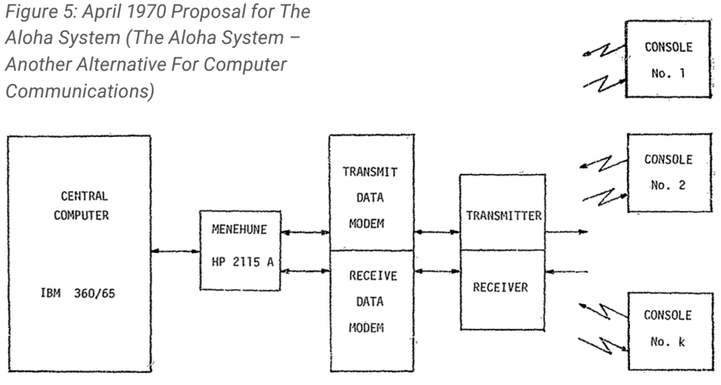
The central computer would be on one island, and students would have to 'dial in' to the system using antiquated inter-island phone lines that AT&T laid in the early twentieth century. The lines were unreliable, and the age and quality of the cables limited speeds.

In 1968, members from the then-new ethernet consortium were meeting in Honolulu for a conference, and members of the University of Hawaii's engineering department had the opportunity to pick the minds of some prominent figures in computing history. Soon after, in September, the University of Hawaii began researching radio communications for computer-to-computer networking. Early on, they were able to get funding from the Office of Aerospace Research (SRMA) and they started developing the system.

In 1970, while at a conference in Washington, DC, the Principal Investigator of the ALOHA system, Norman Abramson and Frank Kuo met with Larry Roberts and Bob Taylor at the Pentagon. During this meeting, they proposed an easily deployable, resilient, and repairable wireless system. This, of course, caught the attention of many people who saw the potential of such a system in the wars of the day.

In June 1971, the central UHF station of the ALOHA System had been tested by the first remote terminal. By the end of 1971, four remote terminals had been connected to the University's central computer through the ALOHA System.

Figure 5: April 1970 Proposal for The Aloha System (The Aloha System – Another Alternative For Computer Communications)



1971: ARPA grant and moving from Berkeley

It is unknown how or why ARPA gained possession of the BCC-500, but this writer speculates that the University of California Berkeley took it from the Berkeley Computer Corporation as a return for their very recent one-million-dollar investment, then it was given or sold to ARPA.

In November 1971, ARPA awarded contract number NAS2-6700 for the ALOHA System. The contract consisted of two primary tasks. Task I is the one that gets all the attention, The ALOHA System network. This is where all the blood and glory were made, interconnecting network systems wirelessly with radio and eventually with satellites. A lot of groundbreaking things came out of Task I, including the first use of packet switching, addressing, wireless collision detection, and ended up creating the first commercial use of wireless network computing in 1975.

> 'Converting to this kind of communication could save a lot of money. For example, ARPA spends in excess of $1 million per year for line charges. With this much money, it could buy a transponder on a domestic satellite.'
>
> - Norm Abramson

This story is not about Task I, it is about Task II.

Task II was titled 'Research in Multi-Processor Computing Systems' for the University of Hawaii to collect the prototype BCC-500 and move it to Hawaii to finish its development. The University of Hawaii built new facilities to house the computer, Holmes Hall, which was still under construction while the machine was being moved. In February 1972, a team from Hawaii and previous members of BCC started to disassemble the machine to move it to Honolulu.

Some sections could only be disassembled by cutting cables, but the final report submitted to ARPA notes that less than $100 of damage occurred throughout the move. In 2024 dollars, this is equal to $767 when accounting for inflation. The equipment was crated and flown to Hawaii on a 707 Jet Freighter. When the equipment arrived, it was all intact but far from operational. According to the report, they spent a lot of time deeply examining each piece for design flaws and inefficiencies. Several design weaknesses were identified and rectified. Often, this required adding wires to the existing printed circuit boards. However, in some instances, completely new boards were necessary. The revised items were then thoroughly tested and installed into newly designed cabinets that offered improved cooling.

The rebuild process was plagued with issues. The air conditioning would fail often, and both rotating drum memory units got sand in them, causing them to crash. It is Hawaii after all. Apparently, sand was such an issue that they turned that section of the building into a clean room with positive air pressure, air filtration and cleaning systems, etc.

The BCC 500 system first became operational as a complete working system in February 1973. It was initially only helpful for system programming due to poor reliability. Still, by March 1973, the system had a schedule for four user hours per day, with the remaining time dedicated to hardware and system development activities. This guaranteed user schedule was modified and expanded several times. By May 1974, the system operated 24 hours a day, seven days a week, and was continuously available to users except for Saturdays, reserved for hardware and software maintenance.

System reliability was better than 95% uptime during hours when Task personnel were present to assist with operations. Despite being often

Aloha Communications System Connects Hawaii to Arpanet

By Sasha Yasinin
CW Hawaii Correspondent

HONOLULU – The University of Hawaii's Aloha system is scoring innovative advances in satellite computer communications with an experimental satellite system which went on-line here recently.

The university also is involved in a joint study with the Advanced Research Projects Agency (Arpa), Bolt Beranek and Newman, Xerox and UCLA to design

Experimenting With Aloha

suitable protocol for packet communications via satellite.

As part of this project, Aloha becomes the first subscriber to a digital communications subsystem installed between the Comsat ground stations at Paumalu, Hawaii, and Jamesburg, Calif., last year.

With this installation Aloha becomes the first operational satellite node on Arpanet, via the Intelsat IV satellite over the Pacific.

The Comsat system enables data to be transmitted at the rate of 50 kbit/sec over a single voice-grade channel.

The single satellite channel can provide two up-link and two down-link data channels and each of these four channels could be simultaneously available to any Comsat ground station in sight of the satellite.

Aloha's director, Norman Abramson, sees this use of voice-grade channels as a big money-saver for computer communications.

As an example of the system's cost-saving potential, Abramson pointed to the possibilities in the next generation domestic satellite system.

"Each of these satellites," he told *Computerworld*, "will have 12 transponders with a bandwidth of 45MHz or so. If one of the potential carriers were to devote one transponder on the satellite to packet communications, it would be possible to transmit data at about 10 million baud. This single transponder could handle some five million on-line terminals.

"Converting to this kind of communications could save a lot of money. For example, Arpa spends in excess of $1 million per year for line charges. With this much money it could buy a transponder on a domestic satellite.

"If computer users are ever going to be free of incredibly high costs of transmitting data, some system similar to this has got to be set up," Abramson said.

But Abramson foresees difficulties with potential carriers in the next generation system devoting a transponder to packet data communications. He sees problems with regulatory agencies in determining how to charge for the service.

Figure 6: Aloha Communications System Connects Hawaii to Arpanet Image source: Computerworld Volume 7, Issue 15 on April 11, 1973

unattended, the system has maintained an overall uptime of approximately 80%.

1973: Aloha communications system connects Hawaii to ARPANET

Not long after coming online, in April 1973, the BCC-500 was connected to a NASA Satellite, making it the first operational satellite node on ARPANET. Otherwise, it was connected to a node at UCLA through trans-oceanic phone lines.

Over the next several years, many pieces of the machine were improved. Graduate students at the University of Hawaii wrote and published many reports and documents about different work done on the machine over the years. The BCC-500 remained operational until January 1980, when it was retired. A backup of the BCC-500 system from 1979 is available online if you would like to see more of the file structure.

From GENIE to ALOHA

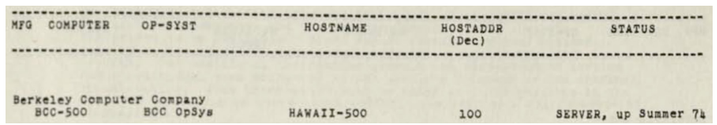
The BCC-500, a pioneering system in the evolution of computing history, was a testament to the ingenuity and collaborative efforts of the Berkeley Computer Corporation and later the University of Hawaii. Initially conceived under the shadow of Project GENIE, the BCC-500's journey from an ambitious project to a fully operational system encapsulates a significant era of innovation and development in time-sharing and networked computing.

Despite facing financial hurdles and operational challenges, the dedication of its developers ensured that the BCC-500 set the stage for future advancements in computing.

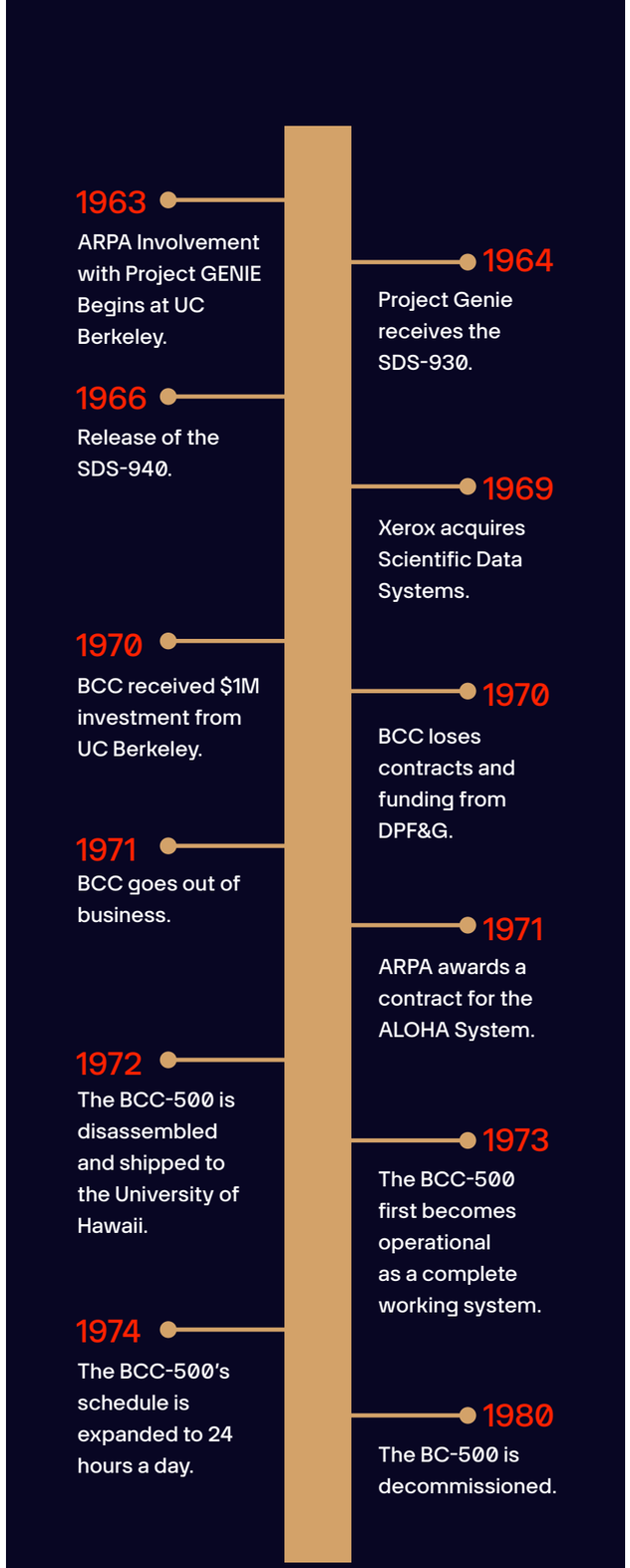
The BCC-500's integration into the ALOHA System marked a pivotal moment in the history of computer networking. Its deployment in Hawaii, under the ARPA grant, showcased the potential of wireless communication and satellite networking. This groundbreaking work connected remote terminals across the islands and linked Hawaii to the ARPANET, paving the way for modern internet technologies. The BCC-500 remained a vital component of the ALOHA System until its decommissioning in 1980, leaving behind a legacy of innovation that continues to influence contemporary computing and networking solutions.

Lars Brinkhoff (@LarsBrinkhoff) and Al Kossow were instrumental in locating resources for this article, and their assistance is appreciated.

Figure 7: ARPANET Directory Listing, 1974



MFG	COMPUTER	OP-SYST	HOSTNAME	HOSTADDR (Dec)	STATUS
Berkeley Computer Company	BCC-500	BCC OpSys	HAWAII-500	100	SERVER, up Summer 74



References

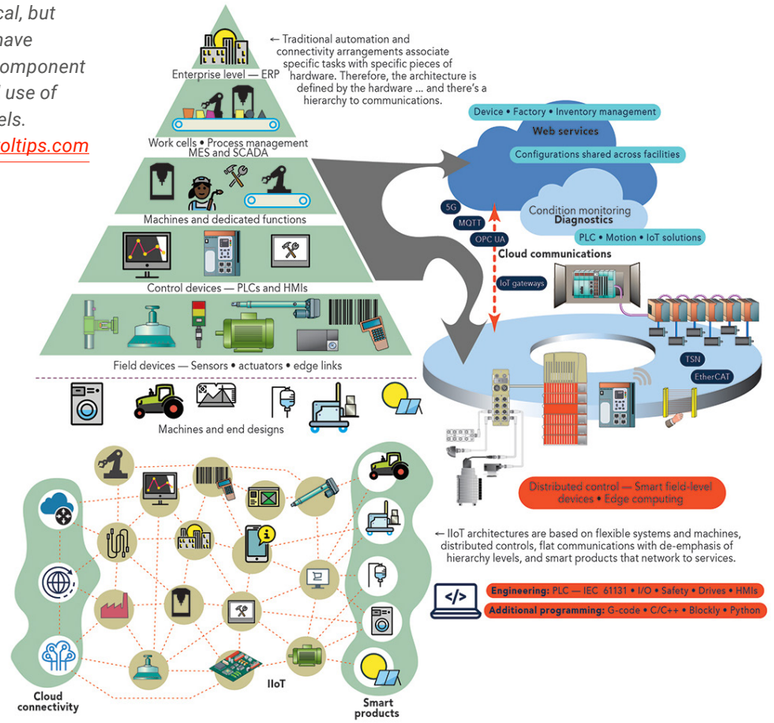
1. P. S. a. P. Meagher, "Project Genie: Berkeley's Piece of the Computer Revolution," Forefront - College of Engineering - University of California Berkeley, Fall 2007.
2. Scientific Data Systems, Technical Manual - Computer Model 930, Santa Monica, Ca, 1966.
3. "Genie has a Multi-Language Structure for Simultaneous Use," The Weekly Newsletter.
4. The New York Times, "Market's Wary of Peace Move," Butte The Montana Standard, 1969.
5. W. L. Harrison, "Supervisor' Manual," 1969.
6. B. Lampson, "Some Remarks on a Large New Time-Sharing System," Berkeley Computer Corporation, Berkeley, 1970.
7. V. Lieberman, "U.C Makes High-Risk Investments," The Faily Californian, 1970.
8. "Control Data Corp. v. International Business MacH. Corp., 306 F. Supp. 839 (D. Minn. 1969)," US District Court for the District of Minnesota, 1969.
9. United States District Court, New York, "IBM 1956 Consent Decree," 1956.
10. B. Ziegler, "IBM Reaches Settlement To End Consent Decree," The Wall Street Journal, 1996.
11. N. Abramson, "The Aloha System - Another Alternative for Computer Communications," University of Hawaii, 1970.
12. Panel, "Artifact Details: ALOHA Panel Oral History (Catalog 102802997)," Computer History Museum, 2023.
13. N. Abramson, "The ALOHA System (Jan 1972)," University of Hawaii, Honolulu, Hawaii, 1972.
14. N. Abramson, "Final Technical Report for Contract Number NAS2-6700," University of Hawaii, Honolulu, 1974.
15. "CPPI Inflation Calculator," Bureau of Labor Statitsics.
16. S. Yasinin, "Aloha Communications System Connects Hawaii to ARPANET," Computer World, 1973.

Application layer protocol options for M2M and IoT functionality

Written by:
Jody Muelaner
Contributing Author at DigiKey

Figure 1: IIoT functions in industrial automation rely on increasingly connected devices employing industrial protocols for networking. The abstracted layers of these networks require no knowledge of underlying layer functions … which is why so much design engineering focuses on machine networks' top (application) layer.
Image source: Getty Images

Figure 2: Traditional system architectures are hierarchical, but Cloud and Fog computing have blurred the lines between component functions. That's prompted use of new network protocol models.
Image source: motioncontroltips.com



With adoption of Internet of Things (IoT) and Industry 4.0 functions, devices are increasingly connected via industrial protocols. What's more, today's machine to machine (M2M) communications are rapidly standardizing on these protocols. Complicating matters is that IoT protocols don't describe a single application-layer protocol, as several standards are in operation. So, while early IoT implementations used standard Internet protocols, there are also dedicated IoT protocols now available.

Modelling communication structures and identifying the right protocol for a particular application can be daunting. This article outlines what various protocols do as well as the options available for these protocols – so design engineers can more easily select the most suitable to integrate.

Defining the application-layer protocol for industrial networks

The structures of communication protocols for digital M2M and IoT systems are conceptually broken into abstract layers, with the most common models having three, four, five, or seven layers. These conceptual frameworks assume

that every layer essentially 'hides' the detailed workings of a given device or software layer from other devices or algorithms with which it's communicating. That's because the layers are defined as containing just enough information for the data exchanges at hand.

No matter the model used, all establish an application layer as the highest abstraction layer between devices that communicate with each other on a network. Consider the application layer as a concept of the Open Systems Interconnection (OSI) model – in which it was first defined by the International Organisation for Standardisation (ISO) nearly three decades ago for network communications. This classic seven-layer model is somewhat overcomplicated for describing some of today's protocols, but is still useful for fully understanding data flow within systems:

A protocol's physical layer allows the transmission of raw data (digital bits) as electrical, radio, or optical signals. This layer specifies the pin layouts, voltage levels, data rates, and line impedances of the physical elements carrying the data. Ethernet is a common physical layer protocol. Read the DigiKey article EtherNet/IP versus PROFINET for more on this.

The data-link layer connects network nodes to let devices establish connections and correct errors at the physical layer. Within the IEEE 802 standard, the data-link layer is divided into a Medium Access Control (MAC) layer (again, to let devices connect) and a Logical Link Control (LLC) layer for identifying the next layer to be used (the network layer) as well as error checking and synchronization. Read more about the functions of the data-link layer in the DigiKey article Implementing Industrial Ethernet with 32-bit MCUs. In contrast, the network layer allows the forwarding of data packets to network addresses. Where Internet protocols refer to the Transmission Control Protocol and Internet Protocol (TCP/IP) model (covered in this article's next section) there's an Internet layer between the data link and network layers. In fact, the Internet layer is often considered a part of the network layer.

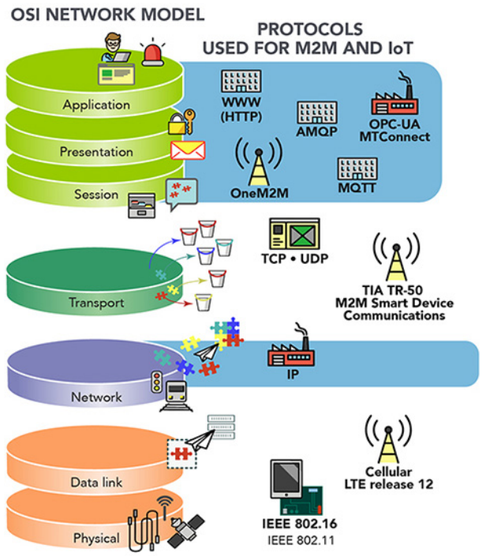


Figure 3: Modern networking protocols (and the application layer) are often described using the classic OSI model of industrial (and commercial) networks. In contrast, three-layer IoT architecture models set the application layer above perception and network layers; four-layer models put it above data processing, network, and sensing layers. Five-layer IoT protocol models are similar but add processing and enterprise layers. Image source: Design World

The first of the next three OSI-model layers is the transport layer, which ensures communication reliability and security during data-sequence transfers. Then the session layer controls when devices connect with each other and whether the connection is one-way (simplex) or in two directions (duplex). Finally, the presentation layer allows data translation so that devices using different syntaxes can communicate.

The focus of this article – the application layer – is the highest level of abstraction and the one with which users and system software interact.

Internet protocols in industrial automation

Internet protocols are data-communication systems so named for the way in which they relay data between networks (and usually reciprocally) for inter-boundary communications. Their functions are often described with the four-layer model of TCP/IP mentioned above. Here, the physical network or link layer is the same as the OSI model's physical layer. In contrast, the TCP/IP Internet layer (which roughly approximates a combination of the OSI model's data-link and network layer functions) handles connections as well as data packets. In IPv6, this layer uses 128-bit IP addresses to identify hosts on the network – and allows more than 1038 unique hosts.

The transport layer in TCP/IP generally consists of either the transmission control protocol (TCP) or the user datagram protocol (UDP). TCP is generally used for human interactions such as email and web browsing. It provides logical connections, acknowledgment of packets transmitted, retransmission of lost packets, and flow control. However, embedded systems use UDP to get lower overhead and better real-time performance. UDP works for domain name servers (DNS) and the dynamic host configuration protocol (DHCP) as well as new IoT applications.

The application layer is the highest level in the TCP/IP model of networks. Functions include those associated with the OSI model's session and presentation layers.

Generic TCP/IP application-layer protocols

Different application-layer protocols have different data bandwidths, real-time capabilities, and hardware requirements. These factors along with plant or OEM-team familiarity with a protocol are often an important selection criterion. Though early Internet protocols including the hypertext transfer protocol (HTTP) and simple mail transfer protocol (SMTP) are largely used for human-driven and human-consumed communications, TCP/IP protocols with an IIoT slant are more focused on machine to machine (M2M) and other industrial communications.

Complicating matters somewhat is how many established application-layer protocols used in TCP/IP for web-based human interactions with information also have consumer and industrial IoT uses. That's certainly true for HTTP and SMTP as well as the secure shell (SSH) and file transfer protocol (FTP). Implementing IoT functions with web technologies is usually feasible if eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) are also used. One caveat is that using HTTP has security implications. That's why it's usually best if any IoT devices in such systems only include a client – and not a server. This prevents the device from receiving connection requests that could allow unauthorized outside access to the network. Here, the WebSocket protocol can establish a full-duplex communication via HTTP. Otherwise, the extensible messaging and presence protocol (XMPP) may be preferred for installations that need to address a large number of devices with good security and real-time data communications.

When IoT projects are led by staff with an IT background, these familiar standards (from the human-readable web) may be preferred. However, newer IIoT protocols can in some instances be better suited to M2M and other industrial communications.

MQTT for vertical connectivity transport functions

Seeing the most rapid adoption in IIoT is the Message Queuing Telemetry Transport (MQTT) protocol – a lightweight protocol initially intended for IoT devices with limited memory. Offering operation on compact processing footprints and requiring minimal bandwidth, MQTT was first developed by IBM to connect sensors on oil pipelines. Unlike the constrained application protocol (CoAP), MQTT is already standardized according to ISO/IEC 20922. MQTT uses the somewhat more resource-intensive TCP transport layer and therefore

consumes more power … but messages can be two bytes – even smaller than those of CoAP.

Due to its open nature, MQTT can also be particularly easy to implement. No wonder Amazon Web Services' AWS IoT employs MQTT for message transport and (with caveats) supports MQTT based on the v3.1.1 specification.

MQTT does have some limitations, which may be due to the fact that MQTT was initially intended as a telemetry protocol – in contrast to the IoT-specific Lightweight Machine to Machine (LwM2M) protocol to be covered shortly. Features such as objects, connectivity monitoring, and remote device actions aren't included in the standard, so inclusion of these tends to be vendor specific which degrades the standardized protocol's value somewhat. MQTT also offers no error-handling abilities. Finally, although MQTT can be made secure with a full TLS protocol, doing so increases its overhead.

Primarily at the enterprise level: AMQP

Advanced Message Queuing Protocol (AMQP) is another open standard with some similarities with MQTT. It offers advanced features such as message queuing. However, the overhead of AMQP is higher than that of MQTT, making it poorly suited to connecting very constrained devices. No wonder its use in industrial IoT applications is less common than its use in performance enterprise messaging.

Figure 4: Nordic's SiP is a low-power MCU with an integrated LTE-M and narrowband (NB)-IoT modem, as well as GPS. A Software development kit allows setup of CoAP. Image source: Nordic Semiconductor

CoAP for connecting simple devices

The constrained application protocol (CoAP) from the Internet Engineering Task Force (IETF) allows communication on low-power networks between devices with minimal memory and processing power. It can operate with very low overhead and requests and responses can be as small as four bytes. CoAP eschews the use of a complex transport stack for use of UDP instead. Refer to the DigiKey article on EtherNet/IP versus PROFINET for more on UDP. Like HTTP, CoAP also uses the REST model – with servers making resources available under a URL and clients accessing them via POST, GET, DELETE and PUT methods. What's more, CoAP can be easily translated into HTTP for integration with other web functions … and integrates with XML and JSON. Engineers should find that connecting IoT devices with CoAP is very similar to connecting devices with Web API.

Connecting battery-powered devices with LwM2M

An application-layer protocol from the Open Mobile Alliance is the LwM2M protocol built specifically for IoT applications. Employed in smart-city applications, shipping-container, and cargo tracking, automated off-highway routines, and utilities monitoring, LwM2M is based on CoAP, so shares many of its attributes. The standard encompasses a wide range of clearly defined and maintained standard objects as well as connectivity-monitoring and remote-device actions. Automatic firmware upgrades also simplify the management of LwM2M-connected devices. Although the inclusion of modules such as JSON increases its overhead, it also makes design work easier for developers. Because LwM2M has been designed specifically for IoT applications, it can also operate a strong DTLS security protocol without increasing overhead.

DDS for real-time distributed applications

The data distribution service (DDS) is a little different – and is often classified as an M2M middleware rather than an application-layer protocol. It provides secure and high-performance connections in (among other things) autonomous vehicles, power generation, and air-traffic control systems. In these settings, DDS supports the connection of embedded systems for distributed control that's freed from overreliance on gateways. DDS also handles the routing and delivery of messages without requiring intervention from applications. Plus, the quality of DDS service parameters are configurable – so network operations can be optimized to the work within system constraints.

Conclusion: IIoT application layer protocols

All protocols have strengths and weaknesses, but open-source options allowing rapid deployment and security (preferably with low overhead) are the most suitable for IoT applications. Embedded systems and system-on-chip (SoC) devices with ever-increasing computing capabilities continue to spur IIoT implementation – and further expand what's possible at various protocols' application layers.

Real-time innovations Connext Drive uses DDS
Sample Autonomous Vehicle Architecture

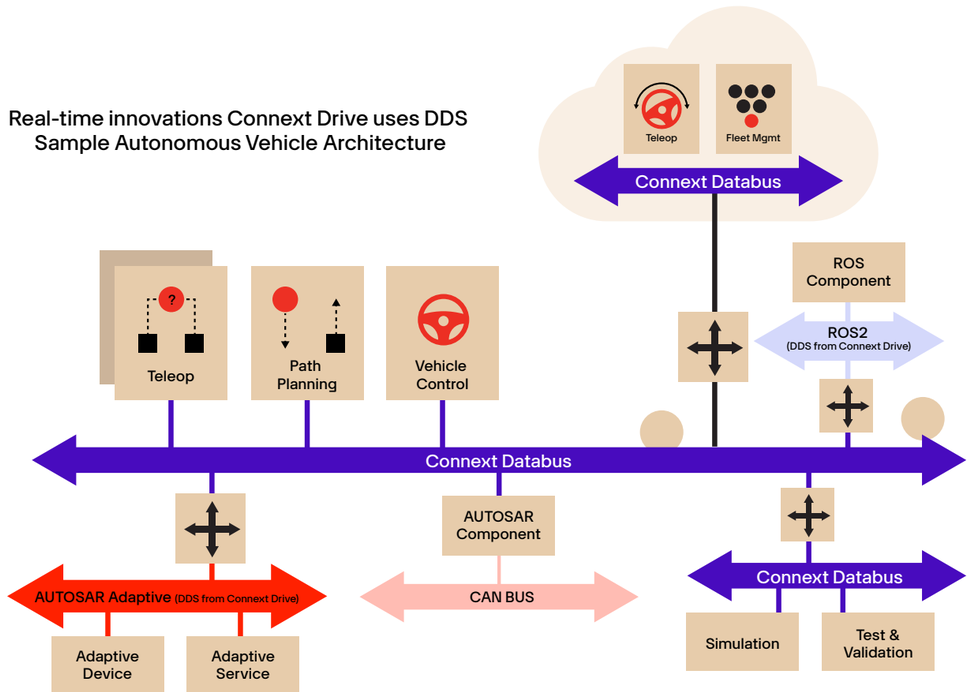


Figure 5: Connext Drive software for autonomous vehicles is built on data distribution service (DDS) middleware – which serves as part of the foundation for AUTomotive Open System ARchitecture (AUTOSAR) Adaptive and ROS2 software architectures. This is just one example of how DDS supports IoT software integration. Image source: Real-Time Innovations

Deploy a secure Cloud-connected IoT device network complete with Edge computing capabilities

Written by:
Stephen Evanczuk
Contributing Author at DigiKey

Though much in demand, the deployment of an Internet of Things (IoT) network with Edge computing resources can be a daunting undertaking with multiple requirements for endpoint devices, Edge computing systems, and secure Cloud connectivity. Although discrete elements of the required solution are readily available, integrating them all into a seamless, efficient IoT application requires immersion in the complex tasks of implementing not only the endpoint and Edge hardware platforms, but also the service interfaces, communications methods, and security protocols required by IoT Cloud providers.

Recently, a steady stream of more highly integrated IoT solutions has emerged to help developers get to market more quickly. For example, a set of Cloud-ready endpoint and Edge computing products from Microchip Technology provides an off-the-shelf solution designed to connect easily with Amazon Web Services (AWS) IoT services and the AWS IoT Greengrass Edge computing service.

This article will briefly discuss why intelligence should be deployed at the Edge. It will then introduce Microchip's AWS-qualified boards that serve as Cloud-ready sensor endpoint systems. The article will then show how those endpoints can be combined with an Edge computing platform based on a wireless system-on-module (SOM) preloaded with AWS credentials and service software to provide near transparent connectivity to the AWS Cloud.

Combining endpoint and Edge system

Ready availability of low-cost, low-power systems has simplified implementation of so-called endpoint systems, which are the sensor and actuator devices that make up the farthest reaches of the IoT application periphery.

Although these endpoint systems connect directly with the Cloud in many IoT applications, more complex applications often require deployment of so-called Edge systems, which lie functionally positioned between endpoints and the IoT Cloud.

By providing local processing capabilities inloops oroximity to a set of IoT endpoints, Edge systems can reduce latency in tight feedback loops, or meet timing requirements for industrial process controls. Edge systems provide local resources needed to process more complex algorithms such as machine learning inference, or sophisticated preprocessing routines used to clean data and reduce the volume and velocity of data driven to the Cloud. This local processing capability proves critical in supporting advanced security policies and privacy requirements such as data minimization prior to transfer across the public Internet.

Enhancing IoT applications with AWS IoT Greengrass

Amazon Web Services (AWS) formalises Edge processing capabilities with its AWS IoT Greengrass service, which provides a portion of its Cloud services running as Greengrass Core on the Edge device. Designed to work closely with Cloud services running on scalable AWS Cloud resources, Greengrass provides a relatively straightforward path for deploying and updating machine learning inference models built with tools such as the AWS SageMaker fully managed machine learning platform (Figure 1).

Local processing is only one of the benefits of an Edge service such as AWS Greengrass. In providing a sort of interface buffer between endpoint systems and Cloud resources, Edge systems also play a key role in meeting IoT application requirements for reduced latency, for enhanced privacy and security,

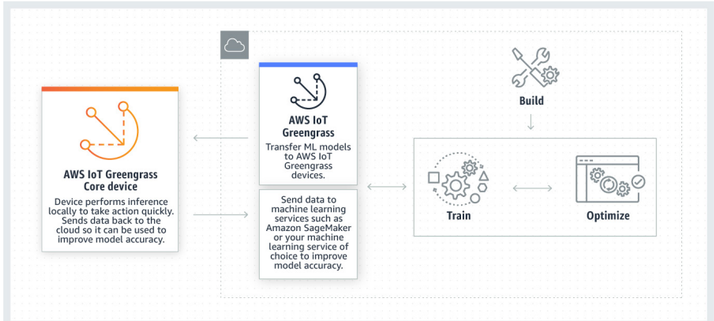


Figure 1: AWS IoT Greengrass simplifies local processing and Edge deployment of advanced functionality including machine learning models trained in the AWS SageMaker machine learning environment. Image source: Amazon Web Services

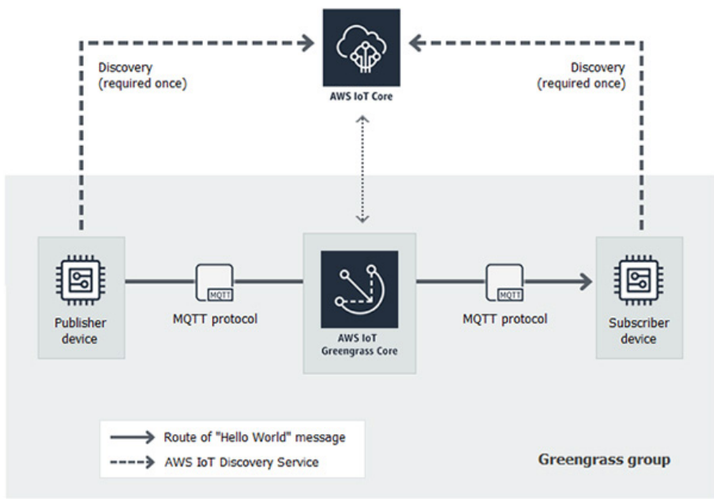


Figure 2: Within an AWS IoT Greengrass group, endpoint devices can communicate with each other and the Cloud using MQTT messaging managed by a Greengrass Core device. Image source: Amazon Web Services

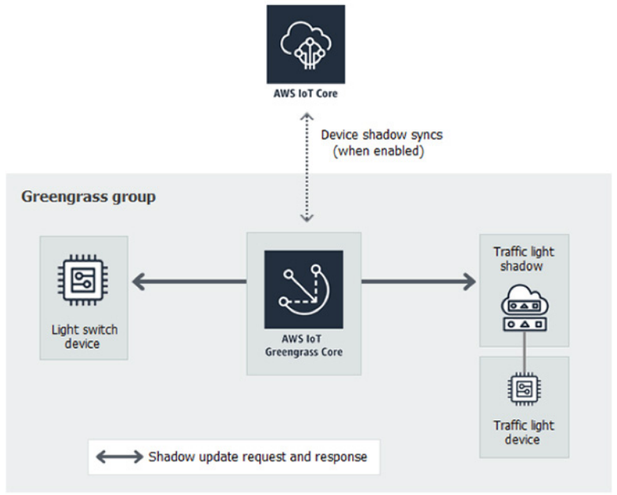


Figure 3: Edge computing service architectures like AWS IoT Greengrass help maintain availability by providing shadow devices that can maintain the latest device state data, allowing IoT applications to continue to function even if the associated physical device goes offline. Image source: Amazon Web Services

and for enhanced availability. AWS Greengrass provides the foundation for delivering these capabilities.

In the AWS Greengrass model, after a one-time discovery phase with Cloud services, endpoint devices within a defined Greengrass group interact with each other using MQ Telemetry Transport (MQTT) messaging managed by a Greengrass Core device (Figure 2).

Once deployed in a Greengrass group, devices can cooperate to avoid lengthy roundtrip delays found in IoT deployments using IoT devices that communicate directly with the Cloud. Instead, devices can signal each other directly through MQTT channels mediated by the local processing capabilities of the Greengrass Core device.

If connectivity to the Cloud is lost, devices can continue to function under management of the Greengrass core device. Conversely, if a device goes offline, other devices and the Cloud-based application can continue to function using data maintained by a virtual device shadow associated with each physical device (Figure 3).

Although straightforward in concept, implementing this coordination among a set of IoT devices can be challenging. For a typical IoT developer, taking full advantage of this Edge computing capability presents a daunting combination of hardware, software, and systems administration challenges. At the hardware level, a network of suitable endpoint and Edge computing devices needs to be built and deployed. Software needs to be written to implement secure communications within the local network of IoT endpoints and Edge devices, as well as with Cloud services. Finally, those devices need to be appropriately configured, provisioned with suitable private keys and certifications, and authenticated with each other and the IoT Cloud service.

To simplify the process, a set of Microchip boards provide AWS-qualified drop-in solutions for both endpoint and Edge devices able to connect simply and securely to the AWS Greengrass Core locally, and to the AWS IoT Core in the Cloud.

Cloud-ready endpoint systems

Designed for rapid deployment as endpoint systems, Microchip's PIC-IoT WA and AVR-IoT WA boards are designed to provide out-of-the-box connectivity with AWS IoT Core. The two boards offer the same overall functionality but are designed to provide familiar platforms for developers accustomed to working with the Microchip PIC microcontroller family, and to those working with the Microchip AVR ATmega microcontroller family. Based on the Microchip ATMEGA4808 8-bit microcontroller, the AVR-IoT WA board uses the same set of components as the PIC-IoT WA

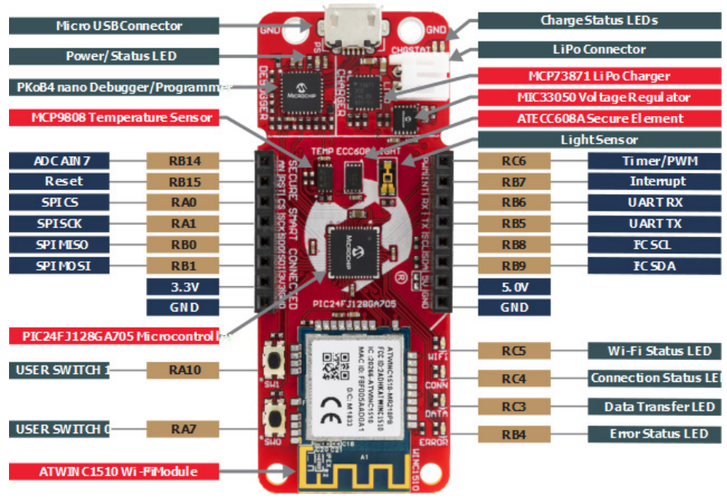


Figure 4: The Microchip AVR-IoT WA and PIC-IoT WA boards provide Cloud-ready endpoint systems that include the same complement of support devices built around different Microchip microcontrollers, including a 16-bit PIC microcontroller for the PIC-IoT WA board shown here. Image source: Microchip Technology

(Figure 4), which is based on the Microchip PIC24FJ128GA705 16-bit microcontroller.

For connectivity, the boards each include a Microchip ATWINC1510-MR210PB certified Wi-Fi module designed specifically for low-power IoT devices. The module integrates 8 megabits (Mbits) of flash and a complete transmission and receiver radio frequency (RF) signal chain including power amplifier (PA), low-noise amplifier (LNA), RF switch, power management, and printed antenna. Along with integrated boot read-only memory (ROM) for rapid firmware boot capability, the built-in network stack supports standard Internet protocols using hardware accelerators to speed Transport Layer Security (TLS) and Wi-Fi security protocols.

Besides a Microchip MCP9808 precision digital temperature sensor and a Vishay TEMT6000X01 photodiode sensor, each board includes a mikroBUS connector. Using this connector, developers can easily expand the hardware base by selecting add-on boards from the broad selection of available Mikroe Click boards. For power and battery management, the boards each include a Microchip MCP73871T-2CCI/ML device, which provides both system power and lithium-ion battery charging from a USB power source or wall adapter.

For security, each board includes a Microchip ATECC608A secure element. For these boards, this device comes pre-provisioned with keys and certificates to provide out-of-the-box support for AWS IoT authentication and security mechanisms.

Using their collection of on-board hardware components and pre-loaded firmware, the boards are designed to connect with minimal effort to AWS IoT Core. Developers need only power up the board using a micro USB cable connected to their personal computer. After the board connects to a local Wi-Fi access point using its own credentials or the developer's, it automatically establishes an MQTT connection with AWS IoT Core using the Wi-Fi module's built-in TCP/IP stack and pre-provisioned security credentials. After establishing that MQTT connection, the board immediately begins transmitting data from its temperature and light sensors. Developers can view the results on a device-specific page in a Microchip sandbox account.

Microchip provides this baseline application in separate repositories for PIC-IoT WA code and AVR-IoT WA code. By examining this code, developers can gain a quick understanding of the basic design patterns, such as the use of MQTT connections when communicating with the Cloud to send sensor data and to receive commands or data (Listing 1).

Developers can extend this code using a variety of development resources. Microchip supports custom software development

```
// This will get called every 1 second only while we have a valid Cloud connection
static void sendToCloud(void)
{
    static char json[PAYLOAD_SIZE];
    static char publishMqttTopic[PUBLISH_TOPIC_SIZE];
    ledTickState_t ledState;
    int rawTemperature = 0;
    int light = 0;
    int len = 0;
    memset((void*)publishMqttTopic, 0, sizeof(publishMqttTopic));
    sprintf(publishMqttTopic, "%s/sensors", cid);
    // This part runs every CFG_SEND_INTERVAL seconds
    if (shared_networking_params.haveAPConnection)
    {
        rawTemperature = SENSORS_getTempValue();
        light = SENSORS_getLightValue();
        len = sprintf(json,"{\"Light\":%d,\"Temp\":%d.%02d}", light,rawTemperature/100,abs(rawTemperature)%100);
    }
    if (len >0)
    {
        CLOUD_publishData((uint8_t*)publishMqttTopic ,(uint8_t*)json, len);
        if (holdCount)
        {
            holdCount--;
        }
        else
        {
            ledState.Full2Sec = LED_BLIP;
            LED_modeYellow(ledState);
        }
    }
```

```
//This handles messages published from the MQTT server when subscribed
static void receivedFromCloud(uint8_t *topic, uint8_t *payload)
{
    char *toggleToken = "\"toggle\":";
    char *subString;
    ledTickState_t ledState;
    sprintf(mqttSubscribeTopic, "$aws/things/%s/shadow/update/delta", cid);
    if (strncmp((void*) mqttSubscribeTopic, (void*) topic, strlen(mqttSubscribeTopic)) == 0)
    {
        if ((subString = strstr((char*)payload, toggleToken)))
        {
            if (subString[strlen(toggleToken)] == '1')
            {
                setToggleState(TOGGLE_ON);
                ledState.Full2Sec = LED_ON_STATIC;
                LED_modeYellow(ledState);
            }
            else
            {
                setToggleState(TOGGLE_OFF);
                ledState.Full2Sec = LED_OFF_STATIC;
                LED_modeYellow(ledState);
            }
            holdCount = 2;
        }
    }
    debug_printer(SEVERITY_NONE, LEVEL_NORMAL, "topic: %s", topic);
    debug_printer(SEVERITY_NONE, LEVEL_NORMAL, "payload: %s", payload);
    updateDeviceShadow();
}
```

Listing 1: Developers can examine code samples in Microchip's software repositories to a gain better understanding of key design patterns such as exchanging MQTT messages with Cloud services as shown in these two functions. Image source: Microchip Technology

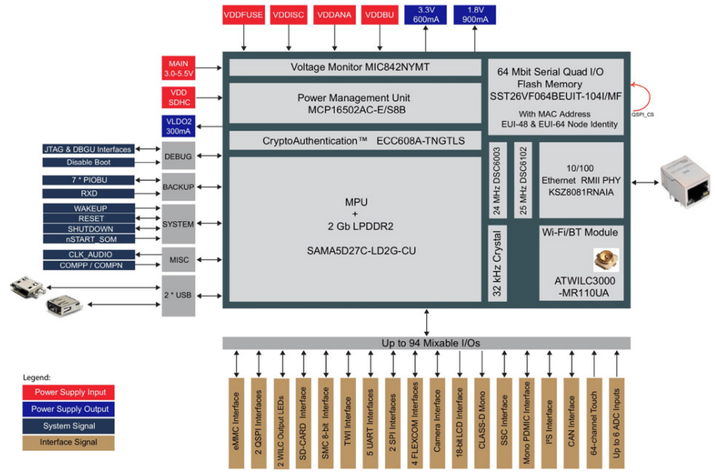


Figure 5: The Microchip ATSAMA5D27-WLSOM1 integrates a full complement of devices required to deliver an AWS IoT Greengrass qualified Edge computing system. Image source: Microchip Technology

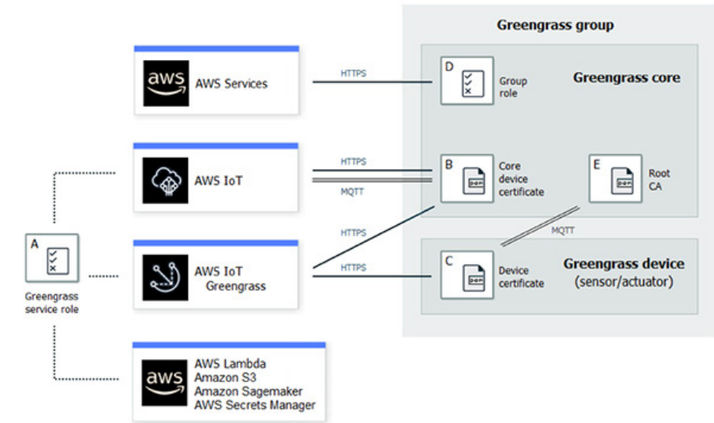


Figure 6: To ensure secure communications transactions, AWS Cloud services and AWS IoT Greengrass groups rely on multiple certificates backed by private keys stored in endpoints and the Greengrass Core device Image source: Amazon Web Service

with its MPLAB X integrated development environment (IDE), Cloud-based MPLAB Xpress IDE, and free MPLAB XC compilers. For debugging, each board includes the Microchip PICkit On-Board (PKOB) nano debugger, which eliminates the need for an additional debugging hardware interface. Developers access the PKOB debugger through the USB connection to their personal computer while working in the MPLAB X IDE.

AWS Greengrass-ready solution

Microchip makes augmenting their IoT network with Edge computing resources based on AWS Greengrass nearly as simple as deploying Cloud-connected endpoints.

For the Edge computing platform, Microchip provides its ATSAMA5D27-WLSOM1 wireless (WL) system-on-module (SoM) with AWS qualified AWS Greengrass support. As with the Microchip endpoint boards, the ATSAMA5D27-WLSOM1 provides a comprehensive hardware platform designed to connect easily to AWS IoT Core services (Figure 5).

For its host processor, the WLSOM1 uses the low-power SAMA5D27 system-in-package (SiP) ATSAMA5D27C-LD2G-CU, which integrates Microchip's high-performance Arm Cortex-A5 processor-based SAMA5D27, which contains two gigabits (Gbits) of low-power double data rate 2 synchronous dynamic random-access memory (LPDDR2-SDRAM).

As with its endpoint boards, Microchip's WLSOM1 includes a certified wireless module. In this case, Microchip uses its ATWILC3000, which supports both Wi-Fi and Bluetooth connectivity with coexistence

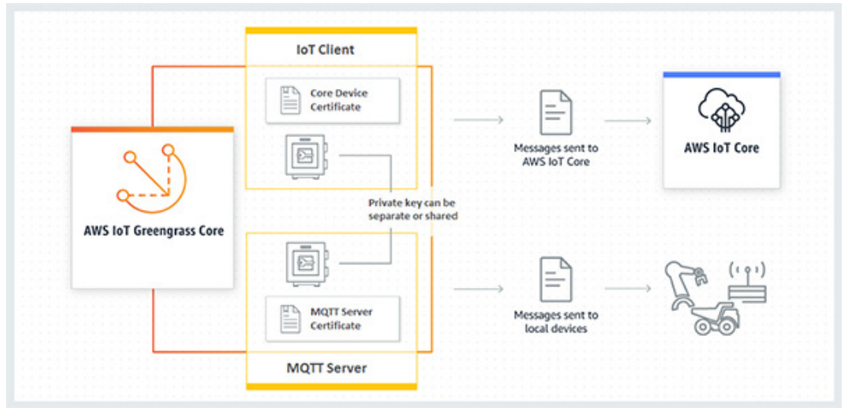


Figure 7: A Greengrass Core device relies on secure storage of private keys using secure elements such as the ATECC608A device integrated in the Microchip ATSAMA5D27-WLSOM1 wireless SOM. Image source: Amazon Web Service

using a combination of integrated hardware accelerators, integrated processors, and stack firmware. The WLSOM1 also offers wired connectivity managed by a Microchip KSZ8081RNAIA Ethernet transceiver. Microchip includes its 64 Mbit SST26VF064BEUI flash, which comes pre-provisioned with an IEEE allocated 6-byte extended unique identifier (EUI-48) and 8-byte EUI-64. This ensures a globally unique MAC address in order to reliably connect to the public Internet. (See 'Flash Memory with a Built-In MAC Address Can Really Help During Development'.)

Finally, the WLSOM1 includes the ATECC608A secure element for hardware-based security. Thanks to its high level of integration, the WLSOM1 requires relatively few components beyond decoupling capacitors and pullup resistors to implement the hardware interface in a board design.

Bringing up a WLSOM1-based board on AWS IoT Greengrass requires very little effort. In fact, most of the effort involves setting up AWS services for its use. Microchip provides developers with step-by-step guides for this, including how to create an AWS account and how to define a Greengrass group of Greengrass core and endpoint devices. After building the target system on a Linux development system, developers upload the target image, Greengrass Core software, and certificates to the WLSOM1, typically using a secure digital card (SDCard) flash drive.

Authentication and secure communications operate transparently to the developer thanks to the hardware-based security provided by the ATECC608A secure element. For Greengrass Edge systems, however, the ATECC608A plays a deeper role in protecting the private keys underlying secure communications between the Greengrass Core running on the Edge system and the AWS Cloud.

Devices in a Greengrass group rely on digital certificates to authenticate each other and their messages within the group and with Cloud-based AWS services (Figure 6). If the underlying security mechanisms and protocols are compromised due to exposed private keys or fraudulent certificates, the group and even Cloud-based resources can be compromised in turn.

AWS protects itself and its users' applications by permitting interactions only with trusted devices that incorporate a hardware secure element able to protect the private keys used for secure communications between the Greengrass Core device and the AWS IoT Core, and between the Greengrass Core device and endpoints (Figure 7).

AWS has identified the WLSOM1 as well as the ATECC608A secure element as Greengrass qualified solutions able to meet its security requirements. In fact, the ATECC608A supports AWS' enhanced security capability provided in IoT Greengrass Hardware Security Integration (HSI). HSI uses the Public Key Cryptography Standards #11, which defines an industry standard application programming interface (API) for communications between a processor and a hardware security module (HSM) used to store private keys. In the WLSOM1, the ATECC608A is designated as an AWS Greengrass qualified HSM Support for this standard security interface is particularly important for Linux-based systems used in Edge systems in general, and in Greengrass Core devices in particular.

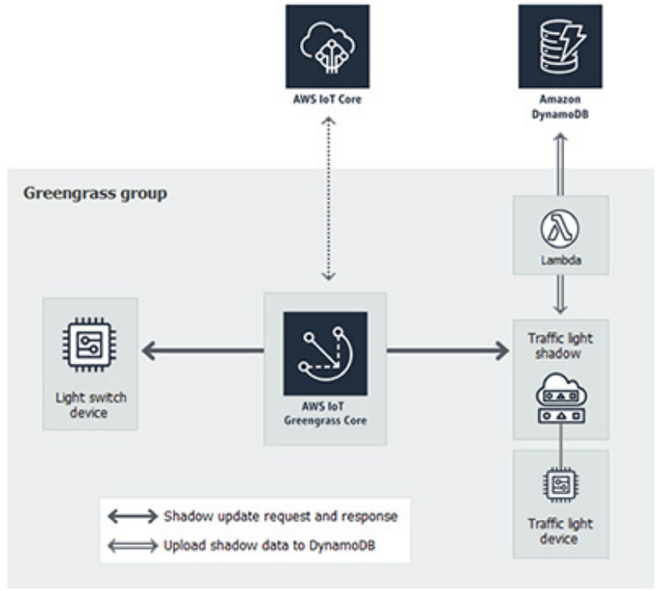


Figure 8: AWS IoT Greengrass enables Edge systems to provide local processing including use of AWS Lambda functions for simple integration with AWS Cloud services for data storage, machine learning, and other capabilities. Image source: Amazon Web Service

Using this secure software foundation, developers can safely extend their Greengrass Core Edge systems with local processing capabilities using AWS Lambda functions, which provide a relatively simple event-driven programming model. While custom code running on the Greengrass Core device can support specific application requirements, AWS Lambda functions allow these devices to interact directly with AWS Cloud services. For example, developers can easily implement Lambda functions that connect endpoints with AWS services, such as Amazon's NoSQL DynamoDB database management system for data storage or other services in the extensive set of AWS offerings (Figure 8).

Conclusion

Deployment of an IoT network with Edge computing resources can prove a daunting enterprise with multiple requirements for endpoint devices, Edge computing systems, and secure Cloud connectivity. Individual pieces of the required solution exist but integrating them into a coordinated IoT application has left developers to face the complex tasks of implementing the service interfaces, communications methods, and security protocols required by IoT Cloud providers.

As shown, a set of Cloud-ready endpoint and Edge computing products from Microchip Technology provides an off-the-shelf solution designed to connect easily with AWS IoT services and the AWS IoT Greengrass Edge computing service. Developers can use Microchip's AWS qualified endpoint boards and a wireless system-on-module Edge computing platform to provide near transparent connectivity to the AWS Cloud and accelerate IoT network deployment.

Further reading

Flash Memory with a Built-In MAC Address Can Really Help During Development

Use rugged multiband antennas to solve the mobile connectivity challenge

Written by:
Bill Schweber,
Contributing Author at DigiKey

Figure 1: Mobile connectivity using various standards and bands is now an expectation on mobile, high-speed installations such as trains, incurring challenges due to wind resistance and environmental ruggedness.
Image source: TE Connectivity

Along with smartphones and Internet of Things (IoT) devices, another major driver for mobile wireless connectivity is transportation applications, including railroads, trucks, and asset tracking.

These applications put a unique set of significant demands on the system antenna such as vibration, shock, temperature extremes, rain, humidity, and the need to operate across wide bandwidths and even multiple bands, all while providing consistent performance.

While it is possible to design and build a suitable antenna, in nearly all challenging applications it makes the most sense to use a standard, properly designed, well-built, fully characterized, off-the-shelf unit. Doing so reduces cost and development time while increasing the level of confidence in the final design.

This article examines the issues associated with transportation antenna design. It then introduces two multiband antennas from TE Connectivity designed to mount on the surface of an enclosure, including a basic 'box' and possibly an exposed moving vehicle.

Applications drive implementation

The antenna is the vital transducer between an electronic circuit and free-space electromagnetic (EM) fields, and so is often the most exposed element of the design. Yet it must deliver the desired electrical and RF performance despite harsh ambient conditions, using a form factor compatible with the overall system design.

For freight systems and especially high-speed passenger rail, it must also be easily integrated into an aerodynamic enclosure that both presents minimal wind resistance and can be protected from harsh environmental conditions (Figure 1). Similar constraints apply to asset tracking situations where the antenna must be exposed for receiving global navigation satellite system (GNSS) signals.

The optimal antenna is a careful blend of application-specific characteristics, including desired radiation patterns, proper impedance match, low voltage standing wave ratio (VSWR), mechanical integrity, enclosure suitability, and ease of electrical connections. There is also the need in many cases to enhance the signal path and to maximize the front-end signal-to-noise ratio (SNR) through the use of an active antenna with an integrated low-noise amplifier (LNA).

As with all components, there are some top-tier parameters used to characterize nearly all antenna designs and installations, as well as others which may be more or less critical in a given situation. For antennas, radiation patterns and performance across the specified band are key considerations.

Implementing antenna principles

The orientation of antennas used for transportation and asset tracking is a challenge as it is

Figure 2: The TE Connectivity 1-2309605-1 is a single module comprising two independent antennas, one for 698 to 960 MHz operation and the other for 1710 to 3800 MHz operation. Image source: TE Connectivity

random and changing, making it important for them to have a consistent, omnidirectional pattern for the top and side views throughout the specified band.

For example, the TE Connectivity 1-2309605-1 M2M MiMo LTE dual antenna is designed for both 698 to 960 megahertz (MHz) and 1710 to 3800 MHz bands and targets 2G, 3G, 4G, cellular, GSM, and LTE applications (Figure 2). A single antenna can be effective for this list of standards because it is agnostic with respect to the specific signal format it is conveying or standard it is supporting; its design is primarily defined by frequency, bandwidth, and power.

Note that a 'dual' antenna is not the same as a 'dual-band' antenna. A dual antenna, such as the 1-2309605-1, has two independent antennas in a single housing and each has its own feed; a dual-band unit is a single antenna with one feed, designed to support two (or more) bands.

Looking at the lower-band antenna of the 1-2309605-1, its radiation pattern for both top and side orientations is uniform across the bandwidth, from the lower end at around 700 MHz, extending through to the upper frequencies, at about 900 MHz (Figure 3).

At 700 MHz (the low end of the frequency band), the gain in decibels relative to an isotropic antenna (dBi) – a standard metric indicating antenna directivity – is just 1.5 dBi, which represents a fairly uniform radiation pattern. This uniformity and evenness contributes to consistent performance, regardless of antenna orientation. Further, the radiation pattern for the 900 MHz higher-frequency end is also fairly even with gain of just 4.5 dBi.

Another important antenna parameter is the VSWR, which is formally defined as the ratio of the maximum to minimum voltage, or the ratio between transmitted and reflected voltage standing waves on a lossless transmission line. In an ideal scenario, the VSWR would be 1:1. While this is often difficult to achieve, it's usually acceptable practice to work with a VSWR in the low single digits.

For the 1-2309605-1 M2M MiMo LTE dual antenna, which can handle up to 20 watts of transmit power, the maximum VSWR when measured with 3 meters (m) of RG174 cable is around 3:1 at one end, and closer to 1.5:1 through most of its bands of operation (Figure 4). In general, this is low enough for many of the targeted applications.

In Figure 4, green is the lower-frequency element #1, red is the higher-frequency element #2, and black is for elements #1 and #2 in free space, while blue is for elements #1 and #2 on a 400 × 400 millimeter (mm) ground plane.

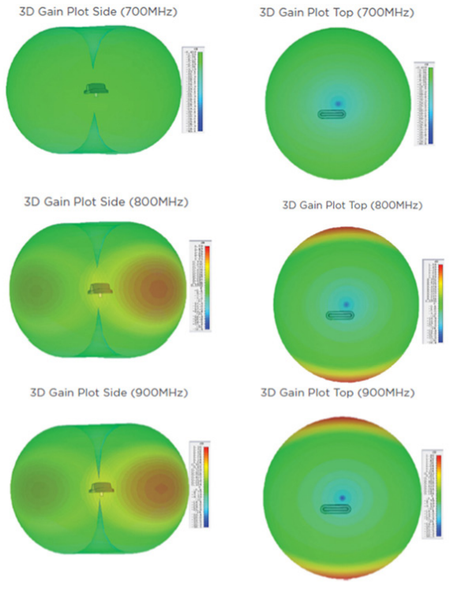


Figure 3: The side (left) and top (right) gain plots of the 1-2309605-1 at 700, 800, and 900 MHz (top row, middle row, bottom row, respectively) show a fairly uniform radiation pattern. Image source: TE Connectivity

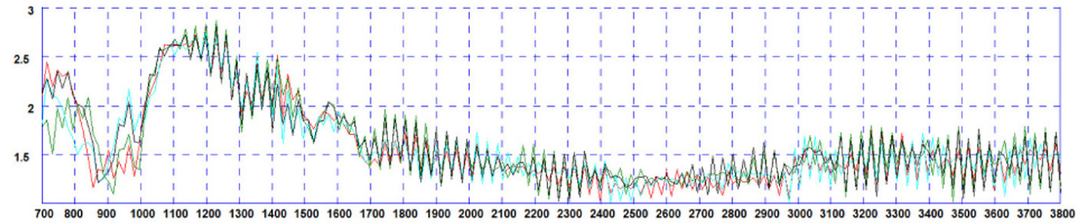


Figure 4: The VSWR (vertical axis) for the 1-2309605-1 M2M MiMo LTE dual antenna as measured with 3 m of RG174 cable shows a low value over the entire active frequency range (x-axis). Image source: TE Connectivity

Co-sited antennas

It is possible to co-locate two or more separate antennas to cover multiple bands. However, this leads to several potential problems. First, there's the obvious issue of space and mounting hardware required on a panel or other surface, as well as the associated installation costs. Second, there are concerns about EM interaction between antennas which will affect their patterns and performance; this constrains how they can be placed with respect to each other. This interaction is measured as antenna isolation, defining to what extent an antenna will pick up radiation from another antenna.

The solution to this quandary is to use a single antenna unit that combines multiple antennas within a single housing or enclosure. Mechanically, this reduces overall size, simplifies installation and antenna-cable routing, and presents a streamlined external appearance.

Electrically, it means that the isolation between the antennas can be measured and specified in advance, minimizing concerns about unexpected or unforeseen interaction. For the 1-2309605-1 M2M MiMo LTE dual antenna, the isolation is at least 15 dB, increasing towards the centers of both bands which the unit serves (Figure 5).

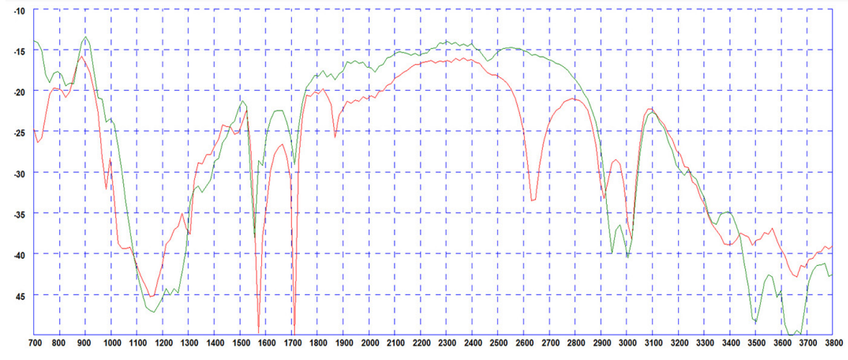


Figure 5: The isolation (y-axis, dB) between the two antennas within the 2309605-1 M2M MiMo LTE dual antenna module is 15 dB or better, measured as a function of frequency (x-axis, MHz). Image source: TE Connectivity

An active receive-antenna function

In addition to the two bands covered by the 1-2309605-1 dual antenna, many applications such as asset tracking also need to receive signals from GPS (US), Galileo (Europe), and Beidou (China) GNSS systems for position or timing

information. To simplify this task and avoid the need for another external discrete antenna, TE offers the 1-2309646-1. This adds a third, receive-only antenna for GNSS signals between 1562 – 1612 MHz to the two antennas of the dual-antenna unit.

However, the need to receive GNSS signals adds another challenge for the system designer that goes back to the basics of the transmit versus receive functions. When used for transmitting, the antenna and its feedline are in a deterministic situation. They take the known, controlled, well-defined signal from the transmitter power amplifier (PA) and radiate it. There is little concern about internal noise on that signal, in-band interference, or out-of-band signals between the PA and the antenna.

Due to the reciprocity principle which applies to all antennas, the same physical antenna used for transmitting can be used for receiving. However, the operating conditions for receiving are quite different than they are for transmitting. Since the antenna is trying to capture a signal with unknowns in the presence of in-band and even out-of-band interference and noise, the desired received signal is not deterministic as it has many random characteristics.

In addition, the received signal strength is low (on the order of microvolts to a few millivolts) and the SNR is also low. For GNSS signals, the received signal power is typically between -127 and -25 dB relative to one milliwatt (dBm), while the SNR is typically between 10 and 20 dB. This fragile signal will be attenuated due to losses in the cable between the antenna and receiver front-end, and it will also have its SNR degraded by unavoidable thermal and other noise in the transmission cable.

For these reasons, the 1-2309646-1 incorporates an LNA as another feature for its third, receive-only GNSS antenna. The LNA provides 42 dB gain for the GNSS signals, thereby significantly boosting the received signal strength. To simplify the use of the LNA, it receives its power (3 to 5 volts DC, at no more than 20 milliamps (mA)) via the amplified RF signal's coaxial cable using a well-established superimposition technique.

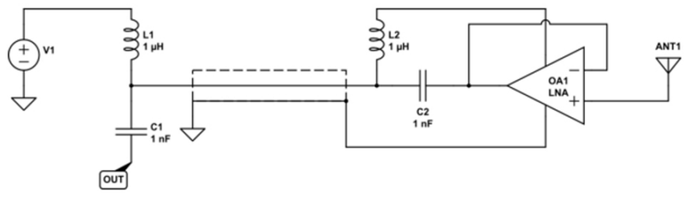


Figure 6: The DC power to the antenna LNA can be superimposed on the cable carrying the antenna/LNA output using a clever arrangement of inductors and capacitors which separate and isolate the DC power and RF signal at each end. Image source: Electronics Stack Exchange

DC power is sent on the cable between the receiver unit to the LNB (Figure 6). The DC power for the LNA (V1) is blocked from reaching the radio head unit (front-end) by small series capacitors (C1 and C2). These capacitors do allow the amplified RF signal from the antenna (ANT1) to pass to the radio head unit (OUT). At the same time, the amplified RF signal is blocked from going back to the power supply V1 by series inductors (chokes) L1 and L2. In this way, DC power to the LNA and amplified RF from the LNA to the radio head unit can share the same interconnection coaxial cable.

Making the physical connection

Any antenna or assembly of antenna elements needs to have a reliable, convenient, and electrically and mechanically secure way to be connected and disconnected from the radio front-end they serve. Further, the complete antenna assembly needs to be protected from the environment and be easy to mount with minimal impact on the mounting surface.

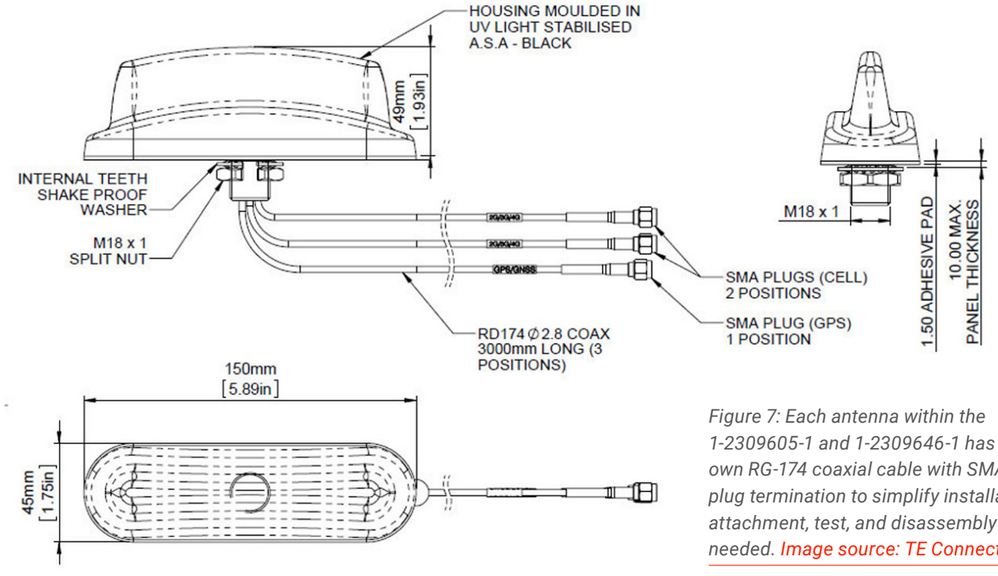


Figure 7: Each antenna within the 1-2309605-1 and 1-2309646-1 has its own RG-174 coaxial cable with SMA plug termination to simplify installation, attachment, test, and disassembly if needed. Image source: TE Connectivity

To meet these goals, each band of the two-band 1-2309605-1 and the three-band 1-2309646-1 is equipped with a 3-meter RG-174 coaxial cable, which is terminated with a standard SMA plug (Figure 7). As a result, connecting or disconnecting one or more of the antennas is straightforward and can easily be done in the factory during system assembly, or in the field as an add-on.

Further, attachment of the multi-antenna module to the system's surface is eased by use of a single internal 18 mm mounting rod, plus an acrylic adhesive pad around the bottom edge of the antenna housing. Attachment of the antenna is a quick operation that leaves no exposed hardware to rust, loosen, or be incorrectly torqued.

The housing of these antennas is optimised for mobile, high-speed motion applications. The streamlined unit is just 45 mm wide and 150 mm long with rounded edges (similar to the 'shark fin' on the roof of automobiles) to minimize its drag coefficient and wind resistance. Further, the UV-stabilized material of the enclosure ensures that exposure to sunlight will not weaken the housing over time.

Conclusion

Mobile, high-speed, multiband wireless connectivity for transportation requires an antenna assembly that can meet demanding electrical, environmental, and mechanical objectives. Two-antenna and three-antenna modules from TE Connectivity provide low band, high band, and optional GNSS band antennas, along with an internal LNA for the latter. These units are equipped with individual coaxial cables and connectors for each antenna, plus a simple surface or panel mount arrangement to facilitate installation and provide critical environmental ruggedness.

Related content

TE Connectivity, Antenna Products

DigiKey, Beyond Wires: Antennas Evolve and Adapt to Meet Demanding Wireless Requirements

DigiKey, Why a Good LNA is Key to a Viable Antenna Front-End

Getting started with Zephyr: a developer's guide to your first project

Written by:
Paige West, Editor at
Electronic Specifier

Zephyr OS is an open-source real-time operating system (RTOS) that, while developed as a project hosted by the Linux Foundation, operates independently within its ecosystem. Primarily designed for embedded devices, Zephyr OS has attracted contributions from a diverse range of industry leaders, establishing itself as a key player in embedded systems and IoT.

This article will introduce developers to Zephyr, outline core concepts, and help readers execute their first 'Hello World' project.

What is an RTOS and why use it?

For many simple applications, such as constantly reading a sensor and displaying the results, a developer may decide to program using a straightforward 'bare metal' approach. With this method, the program executes in a single 'round robin' or 'super loop'. This offers development simplicity but is not particularly efficient, either in terms of power consumption or utilization of the microcontroller's resources. Interrupt service routines (ISR) provide a means to interrupt program flow based on defined events – see Figure 1 left. As use cases become more complicated, for example, using more sophisticated sensors and lots of different system functions or tasks, implementing a state machine is a popular way to program embedded systems. Based on modelling an embedded system behavior according to defined conditions and transitions, state machines are used widely.

As application complexity grows, and particularly where the system is required to control time-constrained events, using an RTOS becomes essential. By real-time, the inference is that the application requires a predictable and deterministic response to performing tasks. This might include reading sensors and controlling associated actuators within a given time window rather than when the microcontroller has finished some other tasks. An RTOS provides an embedded developer with the capability to manage and schedule multiple tasks and control communications between individual tasks. Essentially, the RTOS 'sits above' the application code and takes control of the hardware domain, such as the processor resources, I/O, and peripheral interfaces, effectively decoupling the application from the physical hardware. The RTOS schedules individual threads tasks according to priority and resource availability – see Figure 1 right.

An RTOS is utilized for anything that requires a predictable and reliable behavior, such as controlling the

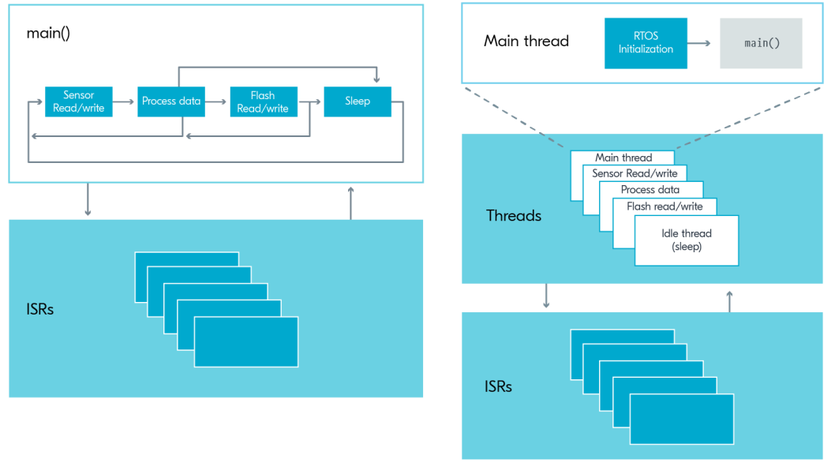
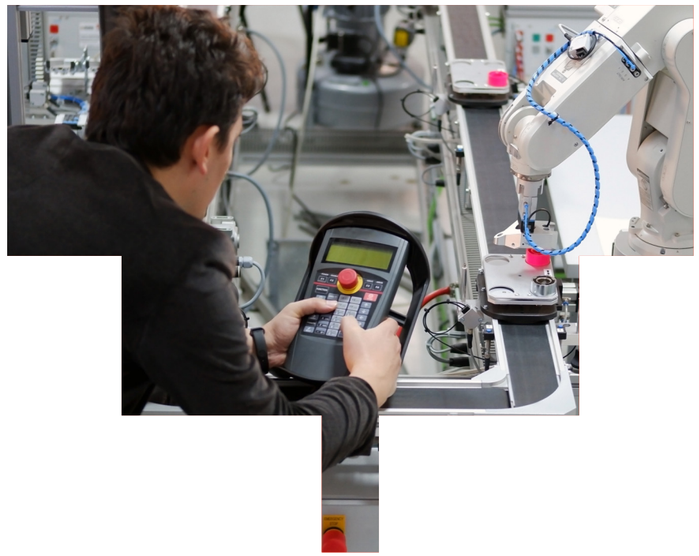


Figure 1: Program flow using a bare-metal approach (left) and an RTOS (right) Image source: Nordic Semiconductor



movement of an industrial robot. Other examples include those running wireless communication stacks, networking peripherals, or those used for motor control.

Basic architecture and key benefits

Zephyr OS is a small, yet highly scalable RTOS that requires less than 8KB Flash, less than 5KB RAM and suits a wide variety of use cases from small sensor nodes to complex multi-core systems. Zephyr's architecture is specifically tailored for lightweight and flexible design, which is ideal for embedded systems and IoT applications. Its basic architecture includes the following elements:

- Kernel configuration: although the Zephyr kernel primarily operates as a microkernel designed for minimal memory usage and modular functionality, it can also be configured as a monolithic kernel. This flexibility allows for optimisation based on the complexity and size constraints of the target embedded device.
- Device drivers: Zephyr offers an extensive library of device drivers for integration of various hardware components.
- Middleware: includes file systems, networking stacks, and protocol libraries essential for building complex embedded systems. This layer offers crucial services like connectivity, data management, and communication protocols.
- Application layer: Zephyr's application layer sits atop the underlying infrastructure and gives developers the flexibility to build custom apps utilizing its features and libraries.

The following are key benefits of Zephyr OS for IoT and embedded system applications:

Real-time capabilities

As an RTOS, a standout feature of Zephyr OS is its real-time capabilities. These capabilities are rooted in efficient scheduling and interrupt handling mechanisms, which are vital for applications requiring precise timing and high reliability, such as industrial automation or medical devices.

Scalability and modularity

The scalability of Zephyr is largely attributed to its modular design which allows developers to include or exclude components based on project requirements, enhancing both the flexibility and scalability of the system. This modular architecture makes Zephyr suitable for a broad range of devices, from simple sensors to smart devices.

Hardware support

Zephyr OS is compatible with various hardware architectures, including x86, ARM, and RISC-V. However, it supports an extensive array of microcontrollers and processors.

The Zephyr kernel supports the following architectures:

- ARCv2 (EM and HS) and ARCv3 (HS6X)
- ARMv6-M, ARMv7-M, and ARMv8-M (Cortex-M)
- ARMv7-A and ARMv8-A (Cortex-A, 32- and 64-bit)
- ARMv7-R, ARMv8-R (Cortex-R, 32- and 64-bit)
- RISC-V (32- and 64-bit)
- SPARC V8
- Intel x86 (32- and 64-bit)
- MIPS (MIPS32 Release 1 specification)
- NIOS II Gen 2
- Tensilica Xtensa

Connectivity

Zephyr OS supports various wireless standards and protocols, including Bluetooth Low Energy (BLE), Wi-Fi, and LoRa. It also accommodates various wired connectivity options and integrates with most networking stacks, allowing devices to connect and communicate in diverse network environments. This robust connectivity is crucial for devices that need to transmit data, receive updates, and interact with other devices and services.

Security features

Security is a key concern in IoT, and Zephyr addresses this with a robust emphasis on features designed to protect devices against threats. These features include secure boot, cryptographic libraries, and regular updates, which are key considerations for developers and manufacturers deploying IoT devices in sensitive or critical applications.

Zephyr OS has a vibrant community which extends beyond developers and commercial entities to include academic institutions and research bodies contributing to its continuous development and improvement.

Community and ecosystem

Zephyr OS has a vibrant community which extends beyond developers and commercial entities to include academic institutions and research bodies contributing to its continuous development and improvement. This community-driven approach ensures that Zephyr OS remains up-to-date with the latest trends and requirements in embedded systems and IoT applications.

Development tools and support

In terms of development tools and support, Zephyr OS is compatible with popular development tools and integrated development environments (IDEs), such as Eclipse, Visual Studio Code, and West (Zephyr's command-line tool). This compatibility facilitates the development process, allowing builders to efficiently create, test, and deploy their applications

Selecting a development board

While Zephyr supports 500+ development boards, consider starting with a popular choice like Arduino, Nordic, or STM32 series for their excellent community support and extensive documentation.

Setting up the development environment

The first step to working with Zephyr is to install a Zephyr command line environment on the computer you intend to use for development. The instructions below are for a Ubuntu-based platform, for Microsoft Windows and macOS see here.

From your computer's root directory enter,

sudo apt update

sudo apt upgrade

Next you will need to check and install some host dependencies using the distribution's package manager.

You can verify which versions of

CMake, Python, and Devicetree complier are currently installed using the following commands.

```
cmake --version

python3 --version

dtc --version
```

The minimum required values are Cmake 3.20.5, Python 3.8, Devicetree compiler 1.4.6.

To install the required dependencies, enter,

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1
```

Once installed, confirm that the required versions are now installed with the above command.

You are now ready to install Zephyr and its Python dependencies. You can opt to install these into a virtual (separate) environment accessible just to Zephyr or to a globally, so all Python applications access them. Since Python package incompatibilities may be experienced when shared in a global environment, it is recommended to install within a virtual environment.

Use these instructions to install Zephyr and its Python dependencies.

Installing the Zephyr software development kit

Zephyr's SDK is compatible with Linux, macOS, and Windows and comes in a compressed file. It provides toolchains for each of Zephyr's supported architectures, which include a compiler, assembler, linker, and other programs required to build Zephyr applications. Install it by extracting the file and executing the setup script. Further OS-specific guidelines are provided below.

By default, the build system uses the Zephyr SDK toolchain if no other is specified. Set the environment variable `ZEPHYR_TOOLCHAIN_VARIANT` to `zephyr` to ensure this.

For installations in non-standard locations, the Zephyr SDK must be registered in the CMake package registry via the setup script for automatic detection. Alternatively, specify the installation path by setting `ZEPHYR_SDK_INSTALL_DIR`.

1. Visit the Zephyr project website to download the SDK installer for your operating system (Linux, macOS, or Windows). Download and verify the latest Zephyr SDK bundle here.

   Here are the steps for downloading Zephyr SDK on Ubuntu:

   ```
   cd ~

   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5/zephyr-sdk-0.16.5_linux-x86_64.tar.xz

   wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5/sha256.sum | shasum --check --ignore-missing
   ```

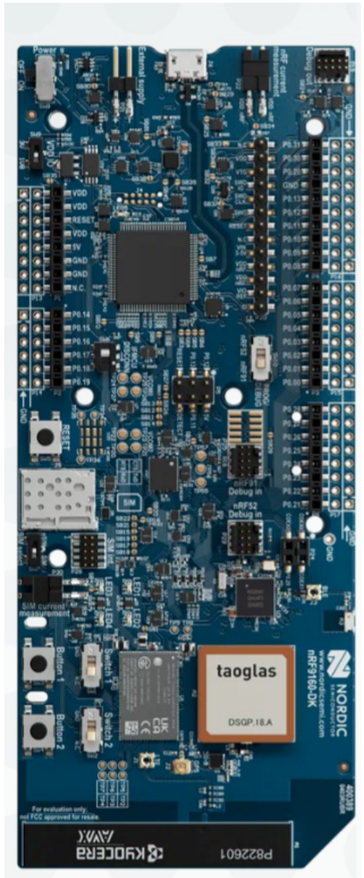


Figure 2: The Nordic nRF9160 cellular IoT development board Image source: Nordic Semiconductor

2. Extract the Zephyr SDK bundle archive:

   ```
   tar xvf zephyr-sdk-0.16.5_linux-x86_64.tar.xz
   ```

   The official documentation recommends that you extract the SDK bundle at one of the following default locations:
   - $HOME
   - $HOME/.local
   - $HOME/.local/opt
   - $HOME/bin
   - /opt
   - /usr/local

   The SDK bundle should contain the zephyr-sdk-0.15.1 directory and, when extracted under $HOME, the resulting installation path will be $HOME/zephyr-sdk-<version>.

3. Next, you'll need to run the Zephyr SDK bundle setup script:

   ```
   cd zephyr-sdk-0.16.5

   ./setup.sh
   ```

   The Zephyr SDK includes a useful command line tool, west. West's features provide a multiple repository management system similar to Git submodes. Zephyr uses west for building applications, flashing, and debugging them.

Setting up an IDE

Using an IDE like VSCode or Eclipse can enhance your development experience with features like code development and debugging. The choice of IDE largely depends on the microcontroller target you are using. Some IDEs provide direct support for the Zephyr RTOS from within its workflow.

Devicetree node label	Analog switch name
vcom0_pins_routing	nRF91_UART1 (nRF91_APP1)
vcom2_pins_routing	nRF91_UART2 (nRF91_APP2)
led1_pin_routing	nRF91_LED1
led2_pin_routing	nRF91_LED2
led3_pin_routing	nRF91_LED3
led4_pin_routing	nRF91_LED4
switch1_pin_routing	nRF91_SWITCH1
switch2_pin_routing	nRF91_SWITCH2
button1_pin_routing	nRF91_BUTTON1
button2_pin_routing	nRF91_BUTTON2
nrf_interface_pins_0_2_routing	nRF_IF0-2_CTRL (nRF91_GPIO)
nrf_interface_pins_3_5_routing	nRF_IF3-5_CTRL (nRF91_TRACE)
nrf_interface_pins_6_8_routing	nRF_IF6-8_CTRL (nRF91_COEX)

Figure 3: The Zephyr OS devicetree for the Nordic Semiconductor nRF9160-DK Image source: Zephyr

Configuring the environment

Follow the Zephyr documentation to set up your build system and environment variables. This will involve configuring the PATH to include the Zephyr SDK and setting up necessary environment variables.

Zephyr development board - Nordic Semiconductor nRF9160-DK

An example development board supported by Zephyr OS is the nRF9160-DK cellular IoT single-board from Nordic Semiconductor. Based on the Nordic nRF9160 system-in-package, it integrates an arm Cortex-M33 core and a dedicated radio cellular transceiver for LTE-M and NB-IoT in addition to a GNSS receiver. The nRF9160-DK also incorporates Nordic nRF52840 multiprotocol wireless SoC for short range communications using Bluetooth 5 and NFC together with integrated antennas. The board hosts an Arduino Uno 3 shield header, four user-programmable LEDs, two buttons, and two switches.

The nRF9160-DK is fully supported by Nordic's nRF Connect SDK.

yr / samples / basic / blinky / src / main.c

pkoscik and fabiobaltieri samples: blinky: add verbose printf output

de Blame 48 lines (39 loc) · 925 Bytes

```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <stdio.h>
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>

/* 1000 msec = 1 sec */
#define SLEEP_TIME_MS   1000

/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)

/*
 * A build error on this line means your board is unsupported.
 * See the sample documentation for information on how to fix this.
 */
static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(LED0_NODE, gpios);

int main(void)
{
        int ret;
        bool led_state = true;

        if (!gpio_is_ready_dt(&led)) {
                return 0;
        }

        ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
        if (ret < 0) {
                return 0;
        }

        while (1) {
                ret = gpio_pin_toggle_dt(&led);
                if (ret < 0) {
                        return 0;
                }

                led_state = !led_state;
                printf("LED state: %s\n", led_state ? "ON" : "OFF");
                k_msleep(SLEEP_TIME_MS);
        }
        return 0;
}
```

Figure 4: The source code of the Zephyr Blinky example Image source: Zephyr

Zephyr OS is integrated within the nRF Connect SDK together with a selection of code examples, application protocols, protocol stacks, libraries, and hardware drivers.

Your first Zephyr project

An established way of confirming you can connect to and flash a development board is by blinking an on-board LED or displaying 'Hello, world' on the system console. The Zephyr SDK provides code samples for these popular tasks in addition to many others. A comprehensive list is available here, divided into functional and sub-system categories. Clicking on each example provides more detailed information, including functional requirements, command line instructions, and board constraints.

Before commencing to try a code sample, it is necessary to determine the development board's identity according to Zephyr's supported board list. For our first project, we'll use the nRF9160-DK development board highlighted above. To blink an LED on the board we must ascertain how the physical LED is mapped into Zephyr OS. The devicetree node label table – see Figure 3 – provides the definitions for the board's LEDs, buttons, and switches. A readable text file of a board's devicetree has a file extension .dts. The devicetree source file for the nRF9160 highlights the mapping and routing of board features.

The Blinky code sample uses Zephyr's GPIO API to configure the required LED using the devicetree and, when executed, will continuously toggle the required LED. You can access the c source code files on the Zephyr GitHub repository – see Figure 4.

Use the Zephyr command line tool west, to build and flash the Blinky example to your nRF9160-DK development board.

west build -b nrf9160dk_nrf52840 samples/basic/blinky

west flash

During execution, the LED flashes and a system console displays the current LED state, either ON or OFF.

If you encounter a build error that points to the struct gpio_dt_spec LED variable, it is likely that you either have an unsupported board or an incorrect LED assignment.

Another test is to use the Hello World example. Since the nRF9160-DK does not have any on-board display, you must use a terminal program to access the board's system console output. For example, using the minicom terminal program from a Linux command line shell, the syntax is -

$ minicom -D <tty_device> -b 115200

<tty_device> defines the port that the nRF52480 is connected to. In most cases using Linux, it will be /

dev/ttyACM1 or /dev/ttyACM0 for the nRF9160.

You can then proceed to build and flash the sample code, using west from the root directory of the Zephyr respository.

west build -b nrf9160dk_nrf52840 samples/hello_world

west flash

Next steps in Zephyr development

Once you have successfully connected to the board and run one or both of the above examples, why not explore more functionality of your chosen development board with one of the more advanced Zephyr code samples?

For example, if you choose an STMicro STM32F3 Discovery board as your development platform, you could access its onboard LSM303DLHC 6-axis accelerometer and magnetometer using this code sample.

Another sensor example is using the popular Bosch BME280 environmental device that measures temperature, humidity and air pressure. Equipped with both SPI and I2C interfaces, the BMS280 is available as an add-on shield or Click board for many embedded development boards.

Zephyr – the proven, open-source RTOS for embedded developers

Zephyr OS offers a stepping-stone for embedded developers who wish to advance their skills from single loop-based applications to real-time, thread-based designs. Supporting all popular microcontroller and microprocessor architectures, and over 500+ development boards, Zephyr is accessible and well-documented.

Start your journey into RTOS development today with Zephyr.

We've got the new products your ideas deserve

We have over 400,000 new, name-brand products in stock and ready to ship–with more added daily. If you can design it, we can help you build it.

Find what you need at digikey.com/new or call 1.800.344.4539

DigiKey

we get technical