CS433: Internet of Things
NCS463: Internet of Things

Dr. Ahmed Shalaby

http://bu.edu.eg/staff/ahmedshalaby14

AWS IoT Core

- AWS IoT Core provides **secure, bidirectional communication** between internet-connected devices, such as sensors, actuators, embedded microcontrollers, or smart appliances, and the **AWS Cloud**. Using AWS IoT Core, you can collect, store, and analyze telemetry data from multiple devices. You can also **create applications** where your users can control these devices from their phones or other mobile devices

- AWS IoT Core is composed of six main components:

  1. *Identity service* – Provides authentication, authorization, and device provisioning.
  2. *Device gateway* – Securely connects IP-connected devices and edge gateways to the AWS Cloud and other devices at scale.
  3. *Message broker* – Processes and routes data messages to the AWS Cloud.
  4. *Rules* – Invokes actions in the AWS Cloud.
  5. *Device Shadow service* – Maintains a shadow of your device so the device can be accessed and controlled at any time.
  6. *Registry* – Stores information about devices and their attributes.

Source: AWS IoTCore

AWS IoTLayers

- IoT applications are composed of many devices (or things) that **securely connect and interact with complementary cloud-based components** to deliver business value.
- IoT applications gather, process, analyze, and act on data generated by connected devices.
- There are six distinct logical layers you should consider when building an IoT workload:
  - Edge layer
  - Provision layer
  - Communications layer
  - Ingestion layer
  - Analytics layer
  - Application layer

Source: AWS Edge Computing

AWS Edge layer .

- The edge layer of your IoT workloads is responsible **for sensing and acting on other peripheral devices**. Common use cases include reading sensors connected to an edge device or changing the state of a peripheral based on a user action, such as *turning on a light when a motion sensor is activated*.
- There are three components: ***physical hardware***, ***embedded operating system***, and ***device firmware***
  - Device Firmware: This component consists of the software and instructions programmed onto your IoT devices.
- Real-Time Operating System (RTOS)?
  - In reality, the processor can only execute one task at a time. By using the scheduler, the processor manages to rapidly switch between tasks. In an RTOS, the scheduler provides a predictable or deterministic execution pattern. Real-time requirements that must respond to a certain event within a strictly defined time window (the deadline). Only an operating system with a deterministic scheduler can guarantee to meet such real-time requirements. A deterministic scheduler enables a user to assign a priority to each task, also called the thread of execution, and then uses this priority to determine the next thread to run.

AWS Edge layer . .

Free Real-Time Operating System (FreeRTOS)?

- FreeRTOS is a class of RTOS designed to run on a microcontroller, which is a small resource-constrained processor.
- FreeRTOS offers a smaller and easier real-time processing alternative for applications where embedded Linux (or Real-time Linux) **won't fit**, are not appropriate, or are not available.
- Amazon FreeRTOS is based on the **FreeRTOS kernel**, the most widely adopted and deployed real-time OS for microcontrollers. Amazon FreeRTOS extends the FreeRTOS kernel with software libraries, making it easy to securely connect those small, low-power devices to AWS cloud services.

AWS Edge layer: FreeRTOS

- **FreeRTOS Architecture Overview**

  FreeRTOS is a relatively small application. The minimum core ofsourcesTOS is only three source (.c) files and a totaling header files, totaling just under 9000 lines of code, including comments and blank lines. A typical binary code image is less than **10KB**. FreeRTOS's code breaks down into three main areas: tasks, communication, and hardware interfacing.

  - **Tasks**: Almost 50% of FreeRTOS's core code deals with the central concern in many operating systems: tasks. A task is a user-defined C function with a given priority. tasks.c and task.h do all the heavy lifting for creating, scheduling, and maintaining tasks.

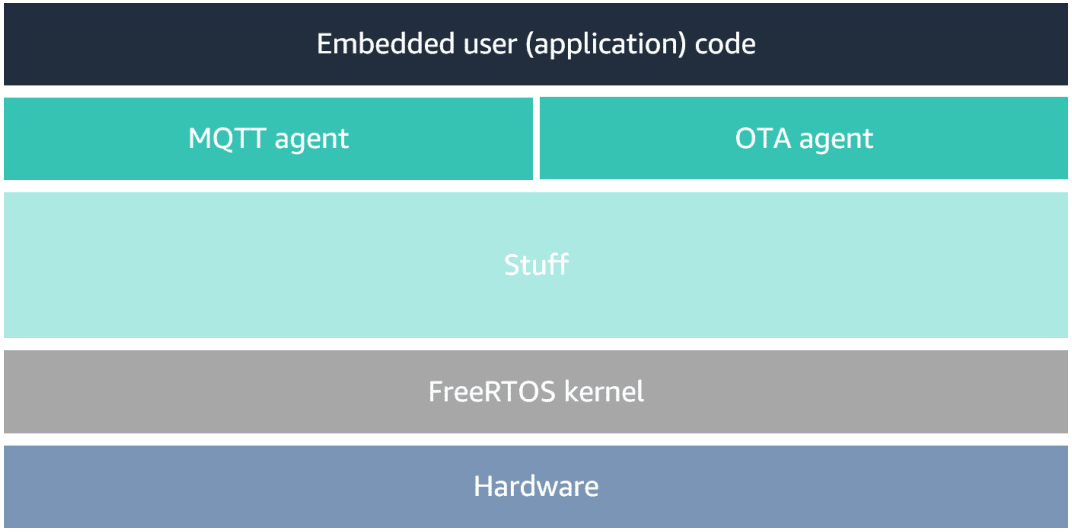
  - **Communication**: Tasks are good, but tasks that can communicate with each other are even better!. About 40% of FreeRTOS's core code deals with communication. Queue.c and queue.h handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes.

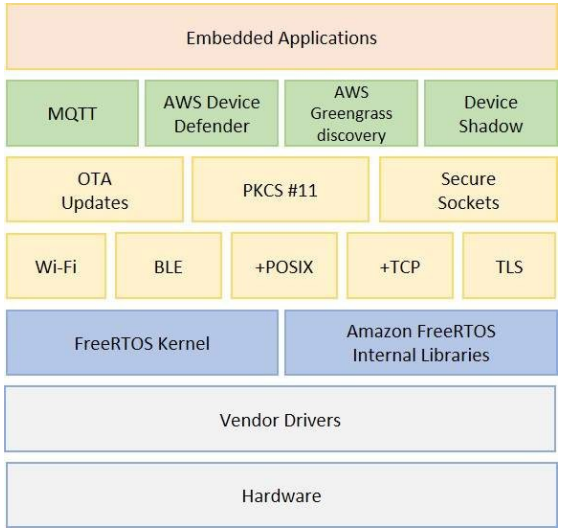
  - **The Hardware Whisperer**: FreeRTOS is hardware-independent; the same code runs whether FreeRTOS is running on the humble 8051 or ARM core. About 6% of core code acts as an interface between the hardware-independent FreeRTOS core and the hardware-dependent code.

https://www.freertos.org/
The Architecture of Open Source Applications

AWS Edge layer: FreeRTOS

- Amazon Free Real-Time Operating System?





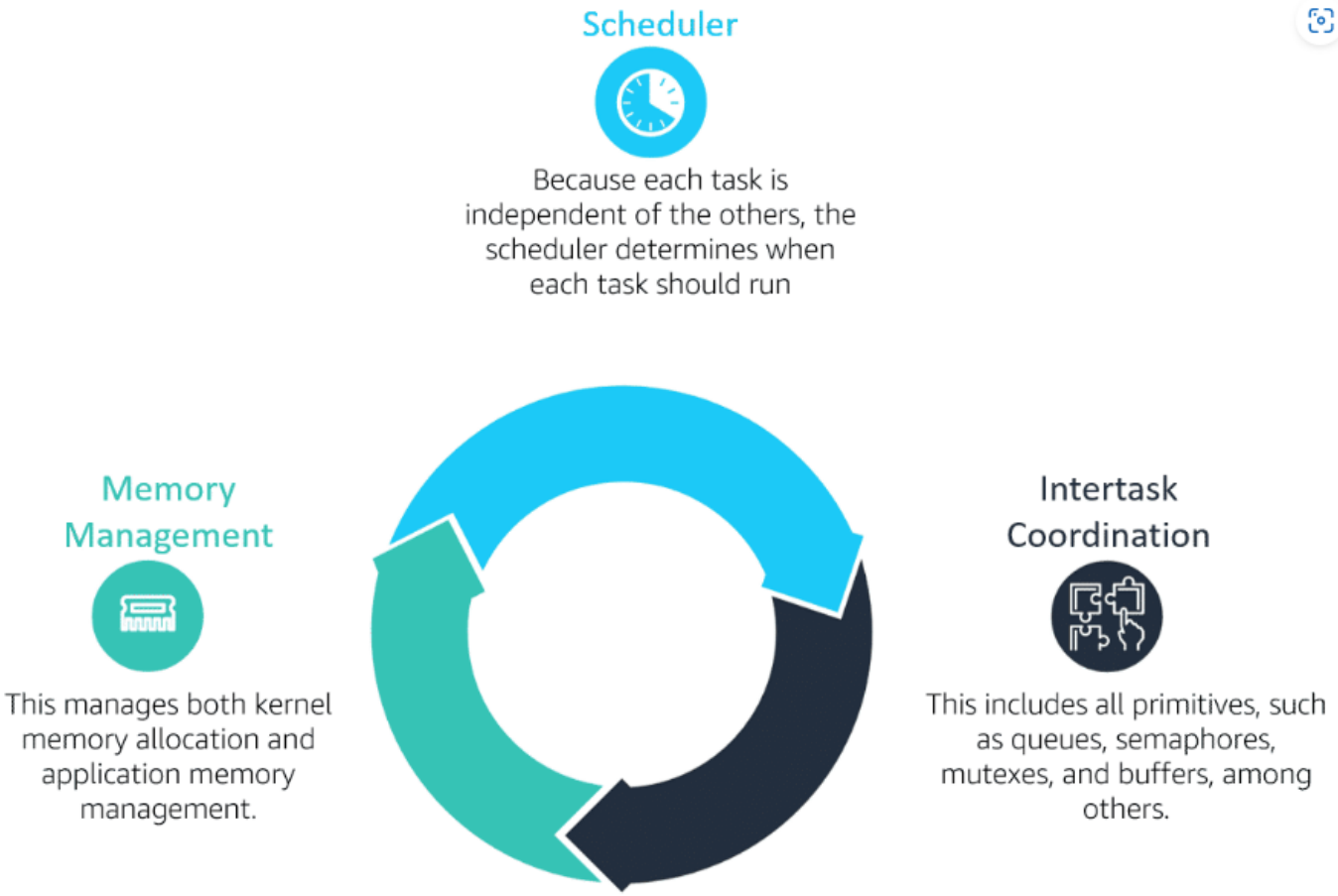
Over the Air (OTA) Updates make it possible to update device firmware without an expensive recall or technician visit.

AWS Edge layer: FreeRTOS

- Amazon FreeRTOS supports a range of MCU hardware options of varying power and capacity, which are pre-qualified to interoperate with Amazon FreeRTOS. This lets Original Equipment Manufacturers (OEM) focus on product innovation rather than complex software development, delivery, and maintenance.
  - Embedded developers don't need to be cloud experts. Amazon FreeRTOS packages all relevant software libraries necessary to enable secure connectivity to the cloud.
  - The architecture has two main components: the Amazon FreeRTOS kernel and the FreeRTOS libraries.

AWS Edge layer: FreeRTOS

❑ **The Amazon FreeRTOS kernel**



AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel

- Is designed to be small, simple, and easy to use. A typical RTOS kernel binary image
- Is in the range of 4KB to 9 KB.
- Never performs non-deterministic operations, such as walking a linked list, inside a critical section or interrupt.
- Includes an efficient software timer implementation that does not use any CPU time unless a timer needs servicing.
- Does not require blocked tasks to engage in time-consuming periodic servicing.
- Enables direct-to-task notifications for fast task signaling, with practically no RAM overhead.

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel

- An embedded application is structured as a set of independent tasks. The FreeRTOS scheduler:
  - Runs only one task at a time
  - Determines when each task should run
  - Allows each task to be assigned a priority
- Each task executes within its own context, is independent of other tasks, and is provided with its own stack. When a task is swapped out by the scheduler, its execution context is saved to its stack. This context is restored when the task is later swapped back in to resume the execution of the task.
- RTOS ensures the highest priority executable task is given processing time. It enables sharing processing time between tasks of equal priority if they are ready to run simultaneously. FreeRTOS also creates an idle task that executes only when no other tasks are ready to run.

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Memory management

- The RTOS kernel needs RAM each time a task, queue, or other RTOS object is created. The RAM can be allocated:
  - Statically at compile time.
  - Dynamically from the RTOS heap by the RTOS API object creation functions.
- When RTOS objects are created dynamically, it's **not always appropriate** to use the standard C library ***malloc( )*** and ***free( )*** functions because they:
  - May be unavailable on embedded systems.
  - May use valuable code space.
  - Are not typically thread-safe.
  - Are not deterministic.
- Therefore, FreeRTOS keeps the memory allocation API in its portable layer. When the RTOS kernel requires RAM, it calls pvPortMalloc( ) instead of malloc( ). When RAM is being freed, the RTOS kernel calls vPortFree( ) instead of free( ).

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Application memory allocation

- When applications need memory, they can allocate it from the FreeRTOS heap. FreeRTOS offers several heap management schemes that range in complexity and features. You can also provide your own heap implementation.
- The FreeRTOS kernel includes five heap implementations:
  - heap_1 is the simplest implementation. It does not permit memory to be freed.
  - heap_2 permits memory to be freed, but does not coalesce adjacent free blocks.
  - heap_3 wraps the standard malloc( ) and free( ) for thread safety.
  - heap_4 merges adjacent free blocks to avoid fragmentation. This includes an absolute address placement option.
  - heap_5 is similar to heap_4 and can span the heap across multiple, non-adjacent memory areas.

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Intertask coordination

➢ Queues:

- Are the primary form of intertask communication.
- Can be used to send messages between tasks and between interrupts and tasks.
- Are used as thread-safe first-in-first-out (FIFO) buffers with new data being sent to the back of the queue.
- Send messages by copy, meaning the data (which can be a pointer to larger buffers) is itself copied into the queue rather than simply storing a reference to the data.

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Intertask coordination

➢ Semaphores:

- Binary semaphores can have only two values and are a good choice for implementing synchronization (between tasks or between tasks and an interrupt).
- Counting semaphores take more than two values, enabling multiple tasks to share resources or perform more complex synchronization operations.

➢ Mutexes:

- Are binary semaphores that include a priority inheritance mechanism.
- Allow tasks to control the mutex according to their priority.
- Temporarily raise a low-priority task's level to the level of the higher-priority task to ensure the higher-priority task's blocked state exists for the shortest time possible.

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Intertask coordination

➢ Direct-to-task notifications:

- Allow tasks to interact with other tasks.
- Allow tasks to synchronize with interrupt service routines (ISRs) without the need for a separate communication object like a semaphore
- Provide each RTOS task a 32-bit notification value that stores the notification content (if any)
- Send RTOS task notifications, which is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value
- Can be used as a **faster and lightweight alternative** to binary and counting semaphores and, in some cases, queues
- Can only be used when there is only one task that can be the recipient of the event

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Intertask coordination

➢ Stream Buffers:

- Allow a stream of bytes to be passed from an interrupt service routine to a task, or from one task to another
- Can include a byte stream that can be of arbitrary
- Can read or write any number of bytes at one time
- Assume that there is only one task or interrupt that writes to the buffer (the writer), and only one task or interrupt that reads from the buffer (the reader)
- Implement the direct-to-task notification mechanism

➢ Message Buffers:

- Allow variable-length discrete messages to be passed from an interrupt service routine to a task, or from one task to another
- Are built on top of stream buffer implementation
- Assume there is only one task or interrupt that writes to the buffer (the writer), and only one task or interrupt that reads from the buffer (the reader)
- Implement the direct-to-task notification mechanism

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Software timers

- A software timer allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's callback function. The time between a timer being started and its callback function being executed is called the timer's period.
- The FreeRTOS kernel provides an efficient software timer implementation because:
  - It doesn't execute timer callback functions from an interrupt context
  - It doesn't consume any processing time unless a timer has actually expired
  - It doesn't add any processing overhead to the tick interrupt
  - It doesn't walk any link list structures while interrupts are disabled

AWS Edge layer: FreeRTOS

❑ The Amazon FreeRTOS kernel: Low-power support

- Like most embedded operating systems, the FreeRTOS kernel uses a hardware timer to generate periodic tick interrupts, which are used to measure time.
- FreeRTOS includes a tickless timer mode, which also includes the tickless timer idle mode, for low-power applications. The timer goes into its idle mode when the scheduler is not running any tasks, i.e., when it is idle and is part of the tickless timer. The tickless idle mode stops the periodic tick interrupt during idle periods (periods when there are no application tasks that are executable) and then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted.
- Stopping the tick interrupt allows the MCU to remain in a deep power saving state until either an interrupt occurs or it is time for the RTOS kernel to transition a task into the ready state.

AWS Edge layer: FreeRTOS

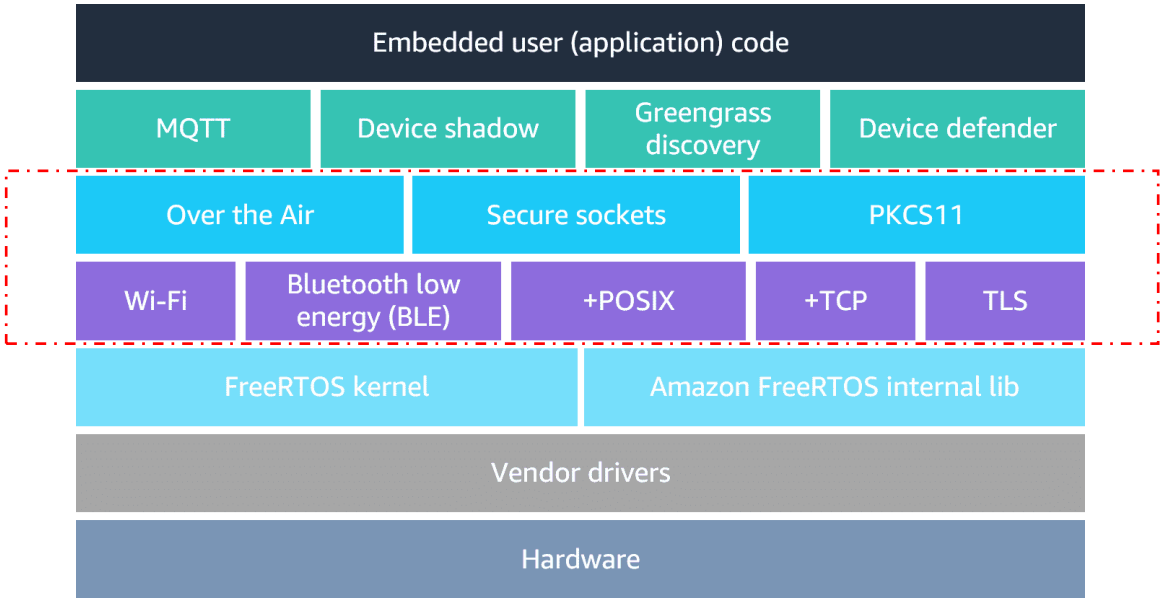
❑ Amazon FreeRTOS libraries

- Amazon FreeRTOS libraries provide additional functionality to the FreeRTOS kernel and its internal libraries. You can use Amazon FreeRTOS libraries for **networking & security** in embedded applications. Amazon FreeRTOS libraries also enable your applications to interact with AWS IoT services.
- Amazon FreeRTOS porting libraries are included in configurations of Amazon FreeRTOS that are available for download on the Amazon FreeRTOS console. These libraries are platform-dependent. Their contents change depending on your hardware platform. (WiFi, Bluetooth Low energy, etc..)
- Amazon FreeRTOS application libraries: You can **optionally** include the following standalone application libraries in your Amazon FreeRTOS configuration to interact with AWS IoT.

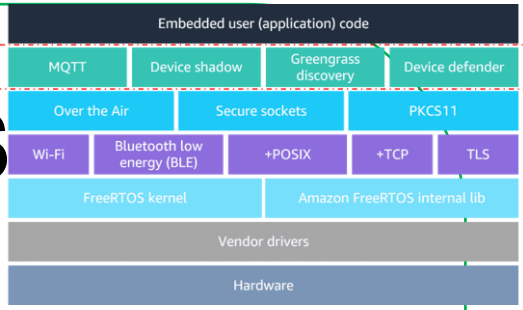
AWS Edge layer: FreeRTOS

❑ Amazon FreeRTOS porting libraries

- The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. FreeRTOS-Plus-POSIX implements a small subset of the POSIX threading API. This subset allows application developers familiar with POSIX API to develop a FreeRTOS application using POSIX-like threading primitives. FreeRTOS-Plus-POSIX only implements about 20% of the POSIX API. Therefore, an existing POSIX-compliant application or a POSIX-compliant library cannot be ported to run on FreeRTOS Kernel using only this wrapper.



AWS Edge layer: FreeRTOS



❑ Amazon FreeRTOS application libraries

- AWS IoT Device Shadow library to store and retrieve the current state (the shadow) of every registered device. The device's shadow is a persistent, virtual representation of your device that you can interact with in your web applications even if the device is offline.
- AWS IoT Device Defender library to send security metrics from your IoT devices to AWS IoT Device Defender. You can use AWS IoT Device Defender to continuously monitor these security metrics from devices for deviations from what you have defined as appropriate behavior for each device. If something doesn't look right, AWS IoT Device Defender sends out an alert so that you can take action to fix the issue.
- AWS IoT Greengrass is software that extends cloud capabilities to local devices. This enables devices to collect and analyze data closer to the source of information, react autonomously to local events, and communicate securely with each other on local networks. Local devices can also communicate securely with AWS IoT Core and export IoT data to the AWS Cloud.

AWS Edge layer: FreeRTOS

❑ **Amazon FreeRTOS Supported microcontroller units (MCUs)**

- Texas Instruments CC3220SF-LAUNCHXL
- STMicroelectronics STM32L4 Discovery Kit IoT Node
- NXP LPC54018 IoT Module
- Microchip Curiosity PIC32MZEF
- Espressif ESP32-DevKitC and the ESP-WROVER-KIT
- Infineon XMC4800 IoT Connectivity Kit
- Xilinx Avnet MicroZed Industrial IoT Kit
- Renesas Starter Kit+ for RX65N-2MB
- MediaTek MT7697Hx Development Kit
- FreeRTOS Windows Simulator
- Nordic nRF52840-DK

AWS IoT

- There are six distinct logical layers you should consider when building an IoT workload:
  - Edge layer
  - Provision layer
  - Communications layer
  - Ingestion layer
  - Analytics layer
  - Application layer

AWS Provision layer

- The provisioning layer of IoT workloads consists of:
  - The private key infrastructure (PKI) used to create unique identities of the devices.
  - The process by which firmware is first installed on devices, and the application workflow that provides configuration data to the device.
  - The ongoing maintenance and eventual decommissioning of devices over time. IoT applications need a robust and automated provisioning layer so that devices can be added and managed by IoT applications in a frictionless way.
  - When you provision IoT devices, you need to install X.509 certificates onto them.
    - X.509 certificates are an ideal identity mechanism for constrained devices with limited memory and processing capabilities.
    - X.509 certificate securely creates a trusted identity for IoT devices that can be used to authenticate and authorize against the communication layer. X.509 certificates are issued by a trusted entity called a Certificate Authority (CA).

AWS Provision layer

- AWS Certificate Manager Private CA

  AWS Certificate Manager Private CA helps to automate the process of managing the lifecycle of private certificates for IoT devices that use APIs. Private certificates, such as X.509 certificates, provide a secure way to give a device a long-term identity that can be created during provisioning and used to identify and authorize device permissions for your IoT application.

- AWS IoT Just-in-Time Registration

  AWS IoT Just-in-Time Registration (JITR) lets you programmatically register devices to be used with managed IoT platforms, such as AWS IoT Core. With JITR, when devices are first connected to the AWS IoT Core endpoint, you can automatically trigger a workflow that determines the validity of the certificate identity and what permissions should be granted.

  Time-to-Check Time-to-Use

AWS Provision layer

Registration Process

- ### Step 1: Install device identity

  AWS IoT Core supports X.509 certificates as device identities. In IoT, it's common for applications to use device certificates such as X.509 security certificates.

- ### Step 2: Register device using certificates

  In AWS IoT Core, the device is registered using its certificate along with a unique thing identifier.

- ### Step 3: Associate device with IoT policy

  The registered device is then associated with an IoT policy. An IoT policy gives the ability to create fine-grained permissions per device. Fine-grained permissions make it so that a specific device has permission to interact with its own MQTT topics and messages. This registration process ensures that a device is recognized as an IoT asset and that the data it generates can be consumed through AWS IoT by the rest of the AWS ecosystem.

AWS Provision layer

Registration Process

- Step 4: Enable automatic registration for device provisioning

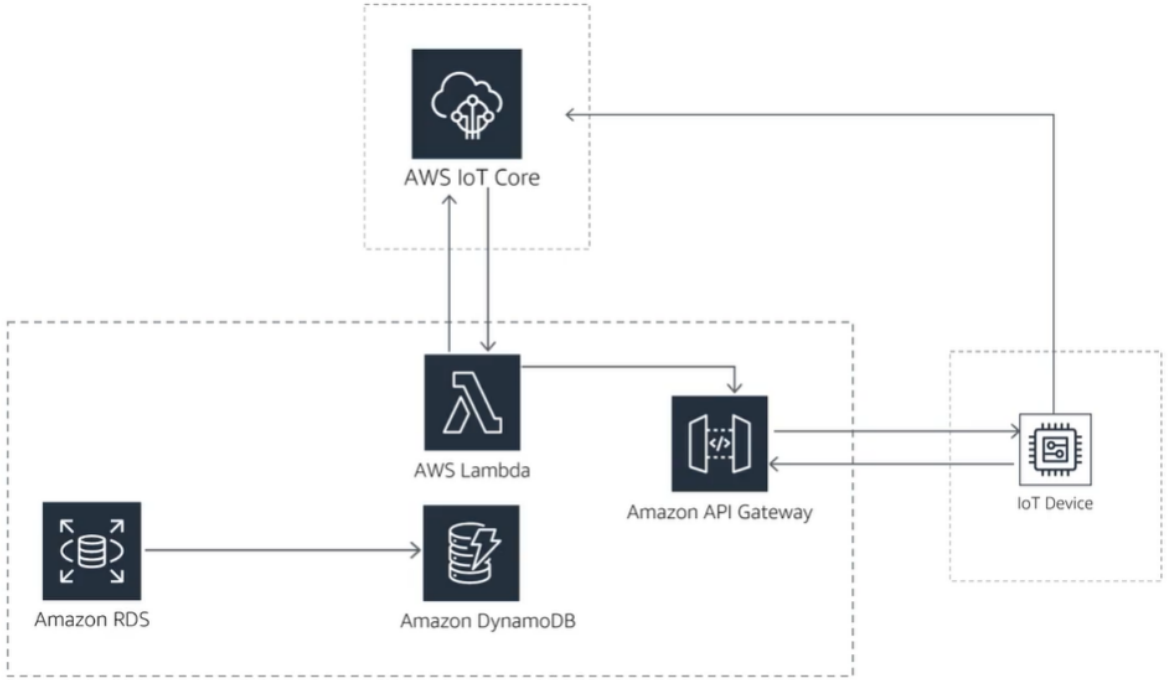
  To provision a device, you must enable automatic registration and associate a provisioning template or an AWS Lambda function with the initial device provisioning event.

  This certificate provisioning mechanism relies on the fact that during manufacturing the device will receive an initial device certificate, which will be used to authenticate to the IoT application - in this case, AWS IoT.

  One advantage of this approach is that the device can be transferred to another entity and the registration process can be repeated with the new owner's AWS IoT account details.

AWS Provision layer

Registration Process



The device provisioning process

AWS Provision layer

Amazon FreeRTOS Public Key Cryptography Standard (PKCS) #11 library

- **What is PKCS #11?**
  - The **Public-Key Cryptography Standards**(PKCS) comprise a group of cryptographic standards that provide **guidelines and application programming interfaces (APIs) for the usage of cryptographic methods**. As the name PKCS suggests, these standards put an emphasis on the usage of public key (that is, **asymmetric**) cryptography.
  - PKCS #11 is a cryptographic token interface standard, which specifies an API, called Cryptoki. With this API, applications can address cryptographic devices as tokens and can perform cryptographic functions as implemented by these tokens. This standard, first developed by the RSA Laboratories in cooperation with representatives from industry, science, and governments, is now an open standard lead-managed by the OASIS PKCS 11 Technical Committee.
  - It follows an object-based approach, addressing the goals of technology independence (any kind of HW device) and resource sharing. It also presents to applications a common, logical view of the device that is called a cryptographic token. PKCS #11 assigns a slot ID to each token. An application identifies the token that it wants to access by specifying the appropriate slot ID.

PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 (oasis-open.org)

AWS Provision layer

Amazon FreeRTOS Public Key Cryptography Standard (PKCS) #11 library

- **Introduction to the Amazon FreeRTOS PKCS library**

  It's a cryptographic API that abstracts key storage, get/set properties for cryptographic objects, and session. The source files for the Amazon FreeRTOS PKCS#11 library are located in AmazonFreeRTOS/lib/secure_sockets/portable.

  In the Amazon FreeRTOS reference implementation, PKCS#11 API calls are made:

  - Perform TLS client authentication during SOCKETS_Connect
  - Import a TLS client certificate and private key for authentication to the AWS IoT MQTT broker.

  Those two use cases--provisioning and TLS client authentication - require the implementation of only a small subset of the PKCS#11 interface standard.

- **Provisioning API**

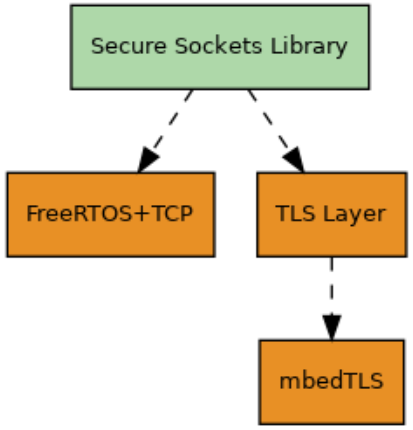
  - GetFunctionList
  - C_Initialize
  - C_CreateObject CKO_PRIVATE_KEY (for device private key)
  - C_CreateObject CKO_CERTIFICATE (for device certificate and code verification certificate)
  - C_GenerateKeyPair

AWS Provision layer

Amazon FreeRTOS Secure Sockets Library

Create embedded applications that communicate securely and make onboarding easy for software developers from various network programming backgrounds.

- The library is based on the Berkeley sockets interface, with an additional secure communication option by TLS protocol. The source files for the Amazon FreeRTOS Secure Sockets library are located in AmazonFreeRTOS/lib/secure_sockets/portable.
- Dependencies and requirements: The Amazon FreeRTOS Secure Sockets library *depends on a TCP/IP stack and a TLS implementation*.
- Usage Restrictions
  - Only TCP sockets are supported by the Amazon FreeRTOS Secure Sockets library. UDP sockets are not supported.
  - Only client APIs are supported by the Amazon FreeRTOS Secure Sockets library. Server APIs, including Bind, Accept, and Listen, are not supported.



Secure Sockets: Main Page (aws.github.io)

AWS IoT

- There are six distinct logical layers you should consider when building an IoT workload:
  - Edge layer
  - Provision layer
  - Communications layer
  - Ingestion layer
  - Analytics layer
  - Application layer

AWS Communication Layer

- The communication layer handles connectivity, message routing among remote devices, and routing between devices and the cloud. The communication layer lets you establish how IoT messages are sent and received by devices and how devices represent and store their physical state in the cloud. Components of this layer encapsulate the MQTT protocols or set of rules that are used by the devices implementing Amazon FreeRTOS to communicate with the AWS IoT Cloud.
- **Components of the communication layer**
  - AWS IoT Core: helps you build IoT applications by providing a managed message broker that supports the use of the MQTT protocol to publish and subscribe IoT messages between devices.
  - AWS IoT Device Registry: Devices connected to AWS IoT are represented by **IoT things** in the AWS IoT registry. The registry allows you to keep a record of all of the devices that are registered to your AWS IoT account.
  - Amazon API Gateway: With Amazon API Gateway, your **IoT applications can make HTTP requests to control your IoT devices**. IoT applications require API interfaces for internal systems, such as dashboards for remote technicians, and external systems, such as a home consumer mobile application. With Amazon API Gateway, IoT customers can facilitate creating common API interfaces without provisioning and managing the underlying infrastructure.

AWS Communication Layer

- **Message broker** for AWS IoT:
  - The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT.
  - When communicating with AWS IoT, a client sends a message addressed to a **topic** like temp/sensorid/room1. The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing. The act of registering to receive messages for a topic filter is referred to as subscribing.
  - The message broker maintains a list of all client sessions and the subscriptions for each session. When a message is published on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the published message to all sessions that have a currently connected client.
  - The message broker supports the use of the **MQTT** protocol to publish and subscribe and the **HTTPS** protocol to publish. Both protocols are supported through IP version 4 and IP version 6. The message broker also supports **MQTT over the WebSocket protocol**.

AWS Communication Layer

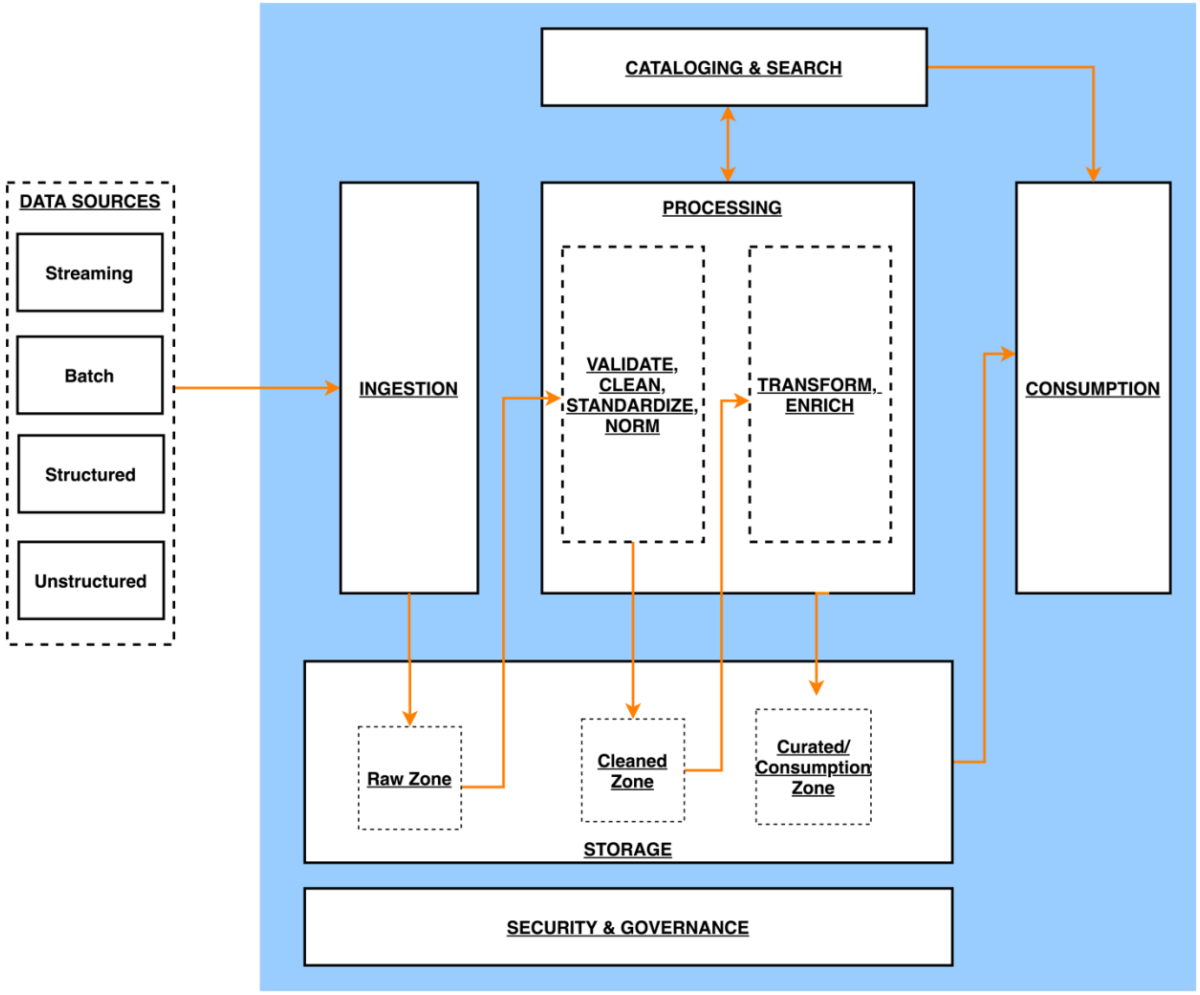
- Amazon FreeRTOS **MQTT Library**:
  - Amazon FreeRTOS includes an open source MQTT client library that you can use to create applications that publish and subscribe to MQTT topics as MQTT clients on a network. The source files for the Amazon FreeRTOS MQTT library are located in AmazonFreeRTOS/lib/mqtt
  - Configuration: Configuration settings for the Amazon FreeRTOS MQTT library are defined as C preprocessor constants. Set configuration settings as #define constants in a file named AWS_IOT_CONFIG_FILE or by using a compiler option such as -D in gcc. Because configuration settings are defined as compile-time constants, a library must be rebuilt if a configuration setting is changed. The MQTT library uses default values when configuration settings are not defined.

AWS IoT layers

- IoT applications are composed of many devices (or things) that **securely connect and interact with complementary cloud-based components** to deliver business value.
- IoT applications gather, process, analyze, and act on data generated by connected devices.
- There are six distinct logical layers you should consider when building an IoT workload:
  - Edge layer
  - Provision layer
  - Communications layer
  - Ingestion layer
  - Analytics layer
  - Application layer

Source: AWS data-analytics

AWS IoT Ingestion layer .



Architecture of a data lake-centric analytics platform

AWS IoT Ingestion layer . .

- **Ingestion layer**: is responsible for bringing data into the data lake. It provides the ability to connect to internal and external data sources over a variety of protocols. It can ingest batch and stream data into the storage layer. The ingestion layer is also responsible for delivering ingested data to a diverse set of targets in the data storage layer including the object store, databases, and warehouses.

- **Storage layer**: is responsible for providing durable, scalable, secure, and money- saving components to store vast quantities of data. It supports storing unstructured data and datasets of a variety of structures and formats. It supports storing source data as-is without first needing to structure it to conform to a target schema or format.

- Components from all other layers provide easy and native integration with the storage layer. To store data based on its consumption readiness for different personas across the organization, the storage layer is organized into the following zones:

  - **Raw zone** –This is a transient area where data is ingested from sources as-is. Typically, data engineering personas interact with the data stored in this zone.

  - **Cleaned zone** – After the preliminary quality checks, the data from the raw zone is moved to the cleaned zone for permanent storage where the data stored in this zone is stored in its original format. Data engineering and data science personas typically interact.

  - **Curated zone** – This zone hosts data that is in the most consumption-ready state and conforms to organizational standards and data models. Datasets in the curated zone are typically partitioned, cataloged, and stored in formats that support performant and cost-effective access by the consumption layer.

AWS IoT Ingestion layer . . .

- **Cataloging and search layer:** is responsible for storing business and technical metadata about datasets hosted in the storage layer. It provides the ability to track schema and the granular partitioning of dataset information in the lake.

- **Processing layer**: is responsible for transforming data into a consumable state through data validation, cleanup, normalization, transformation, and enrichment. The processing layer can handle large data volumes and support schema-on-read, partitioned data, and diverse data formats. The processing layer also provides the ability to build and orchestrate multi-step data processing pipelines that use purpose-built components for each step.

- **Consumption layer:** is responsible for providing scalable and performant tools to gain insights from the vast amount of data in the data lake. It supports analytics across all personas across the organization through several purpose-built analytics tools that support analysis methods, including SQL, batch analytics, BI dashboards, reporting, and ML.

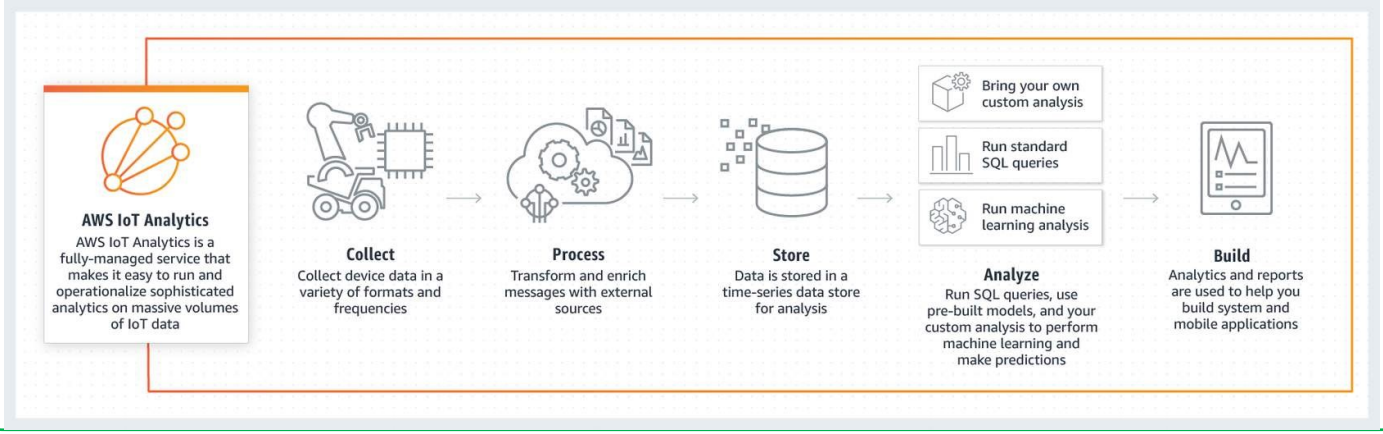
- **Security and governance layer:** is responsible for protecting the data in the storage layer and processing resources in all other layers. It provides mechanisms for access control, encryption, network protection, usage monitoring, and auditing. The security layer also monitors the activities of all components in other layers and generates a detailed audit trail.

AWS IoT Analytics .

- **AWS IoT Analytics** automates the steps required to analyze data from AWS IoT devices. You configure the service to collect only the data you need from your devices, apply transformations to process the data, and enrich the data with device-specific metadata, such as device type and location, before storing it. Then, you can analyze your data by running queries using the built-in SQL query engine or perform more complex analytics and machine learning inference.
- AWS IoT Analytics terminology
  - **Channel** collects and archives raw, unprocessed message data before publishing this data to a pipeline.
  - **Pipeline** consumes messages from a channel and helps you to process and filter the messages before storing them in a data store.
  - **Data store** is not a database, but it is a scalable and queryable repository of your messages. You can have multiple data stores for messages that come from different devices or locations.
  - **Dataset** contains SQL statements and expressions that you use to query the data store along with an optional schedule that repeats the query at a day and time that you specify.
  - **Dataset contents** are the results of running your dataset. After you create dataset contents, you can view them from the AWS IoT console or by using the AWS IoT API or AWS Command Line Interface (AWS CLI).

AWS IoT Analytics . .

- Analysis Process
  - **Collect**: Begin by defining an AWS IoT Analytics **channel** and selecting the specific data to collect, such as temperature sensor readings.
  - **Process**: Configure AWS IoT Analytics **pipelines** to process your data. AWS IoT Analytics pipelines support transformations, such as Celsius to Fahrenheit conversion, conditional statements, message filtering, and message enrichment, using external data sources and AWS Lambda functions.
  - **Store**: After processing the data in the pipeline, AWS IoT Analytics **stores** it in an IoT-optimized data store for analysis.
  - **Analyze**: Query the data store by using the built-in **SQL query engine** in AWS IoT Analytics to answer specific business questions.
  - **Build**: Build **visualizations** and **dashboards** to get business insights quickly from your AWS IoT Analytics data using Amazon **QuickSight**.



AWS IoT Application Layer .

AWS IoT provides several ways to ease the way cloud-native applications consume data generated by IoT devices. These connected capabilities include features from serverless computing, relational databases to create materialized views of your IoT data, and management applications to operate, inspect, secure, and manage your IoT operations.

- **Management Applications**: creates scalable ways to operate your devices once they are deployed in the field. Common operational tasks such as inspecting the connectivity state of a device, ensuring device credentials are configured correctly, and querying devices based on their current state must be in place before launch so that your system has the required visibility to troubleshoot applications.
  - AWS IoT Device Defender is a fully managed service that audits your device fleets, detects abnormal device behavior, alerts you to security issues, and helps you investigate and mitigate commonly encountered IoT security issues.
  - AWS IoT Device Management eases the organizing, monitoring, and managing of IoT devices at scale. At scale, customers are managing fleets of devices across multiple physical locations. AWS IoT Device Management enables you to group devices for easier management. You can also enable real-time search indexing against the current state of your devices through Device Management Fleet Indexing. Both Device Groups and Fleet Indexing can be used with Over the Air Updates (OTA) when determining which target devices must be updated.

AWS IoT Application Layer . .

- **User Applications**: support end-consumer views, business operational dashboards, and the other net-new applications you build over time, you will need several other technologies that can receive the required information from your connectivity and ingestion layer and format them to be used by other systems.
  - Amazon DynamoDB is a fast and flexible NoSQL database service for IoT data. With IoT applications, customers often require flexible data models with reliable performance and automatic scaling of throughput capacity.
  - With Amazon Aurora your IoT architecture can store structured data in a performant and cost-effective open-source database. When your data needs to be accessible to other IoT applications for predefined SQL queries, relational databases provide you with another mechanism for decoupling the device stream of the ingestion layer from your eventual business applications, which need to act on discrete segments of your data.

Best practices

- Start from the cloud and then move on to the device running Amazon freeRTOS:
  - Create a thing in the registry
  - Create a certificate
  - Create and attach a policy
- Start with the demo code and modify it to work with multiple topics: publish, subscribe, and act on received data (modify device behavior, as this is what matters most, in the end).
- Use GitHub (or similar) to manage the code.
- Consider scaling early: end-point(s), CA, management, security, registry, types, MQTT topic namespaces, and thing groups.
- Document the existing demo MCU tasks' timing and behavior before adding your own tasks and application code.
- Consider drawing the architecture of the app as it relates to the native MCU layers and embraces iterative development.
- Use a tool, such as Percepio Tracealyzer, to record all the events that are occurring in the application. A developer can then view how many tasks exist, how they interact, how the heap is used, and even CPU utilization. Use the tool with the demo code and then use it as the development progresses.