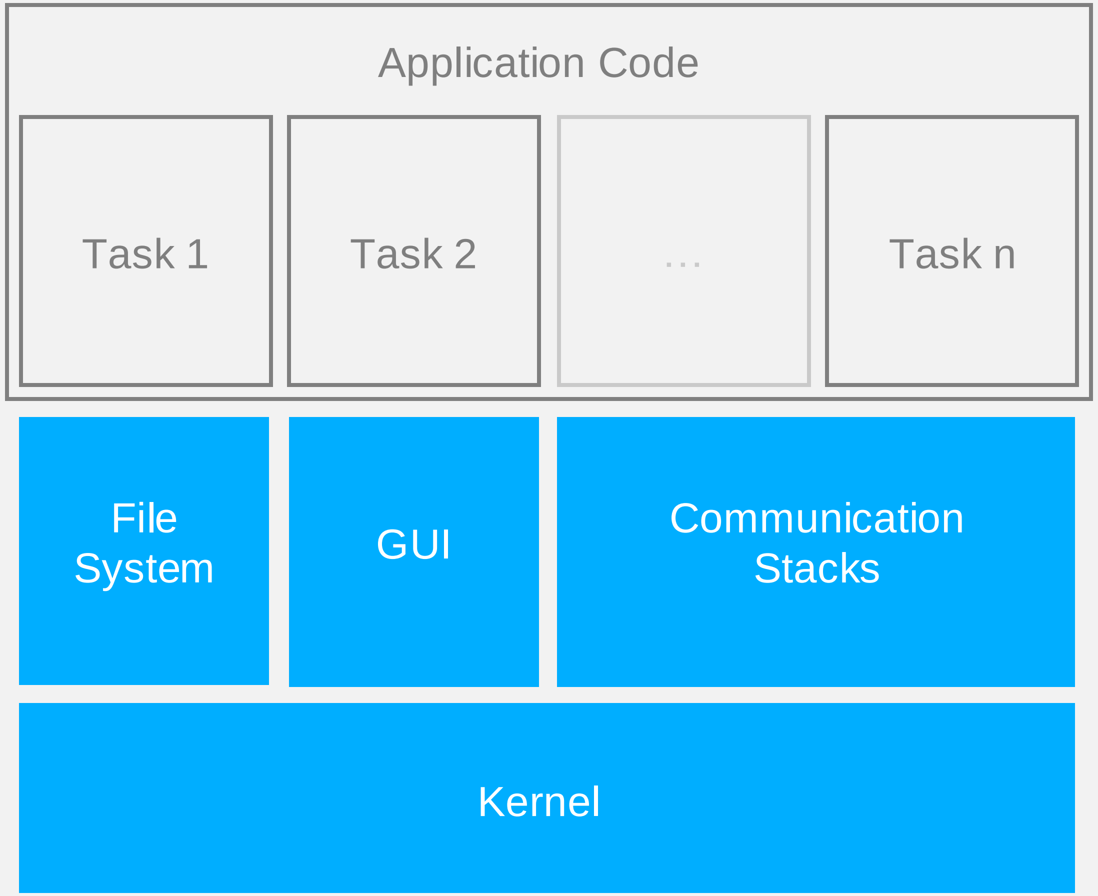
Simplifying IoT Development with an RTOS

Matt Gordon
Sr. Product Manager, RTOS and SW Platform
SILICON LABS

A High-Level RTOS Introduction



Real-Time Operating System (RTOS)

Framework for writing multi-task applications

- Alternative to bare-metal, or super-loop, architecture

Central component is a kernel

- First and foremost, a task scheduler
- Tends to be very small (<15 kBytes of code)

RTOS may include additional software

- File system, GUI, protocol stacks, drivers, etc.
- All components generally lightweight, efficient

Differs from desktop and mobile OSes

- Written for resource-constrained devices
- Often delivered as source code and built with application
- MMU is usually not a requirement
- In some cases, there is no user/kernel mode distinction

What Is the Impact of Using an RTOS?

Advantages

Logical framework for SW development

- Application divided into prioritized tasks
- Easy to assign tasks to different developers
- Add low-priority tasks w/o impact to high-priority tasks

Reuse of existing app code

- For popular RTOSes, large amounts of examples exist

Community support

- Other users may have tips and tricks

Mitigate HW complexity

- Built in support for power mgmt., other HW features

Disadvantages

Learning curve

- New APIs and, in some cases, development environment

Overhead

- Occupies Flash and RAM
- Consumes small portion of CPU's cycles

Example Application 1 (No Kernel)

Simplified USB device

- Goal is to receive and respond to USB packets

ISR is triggered by packet reception

- Packet not fully processed in ISR
- Flag variable is set for USB_Packet()

USB_Packet() called periodically from main()

- Responsible for processing packet
- Frequency depends on contents of main()

ISR

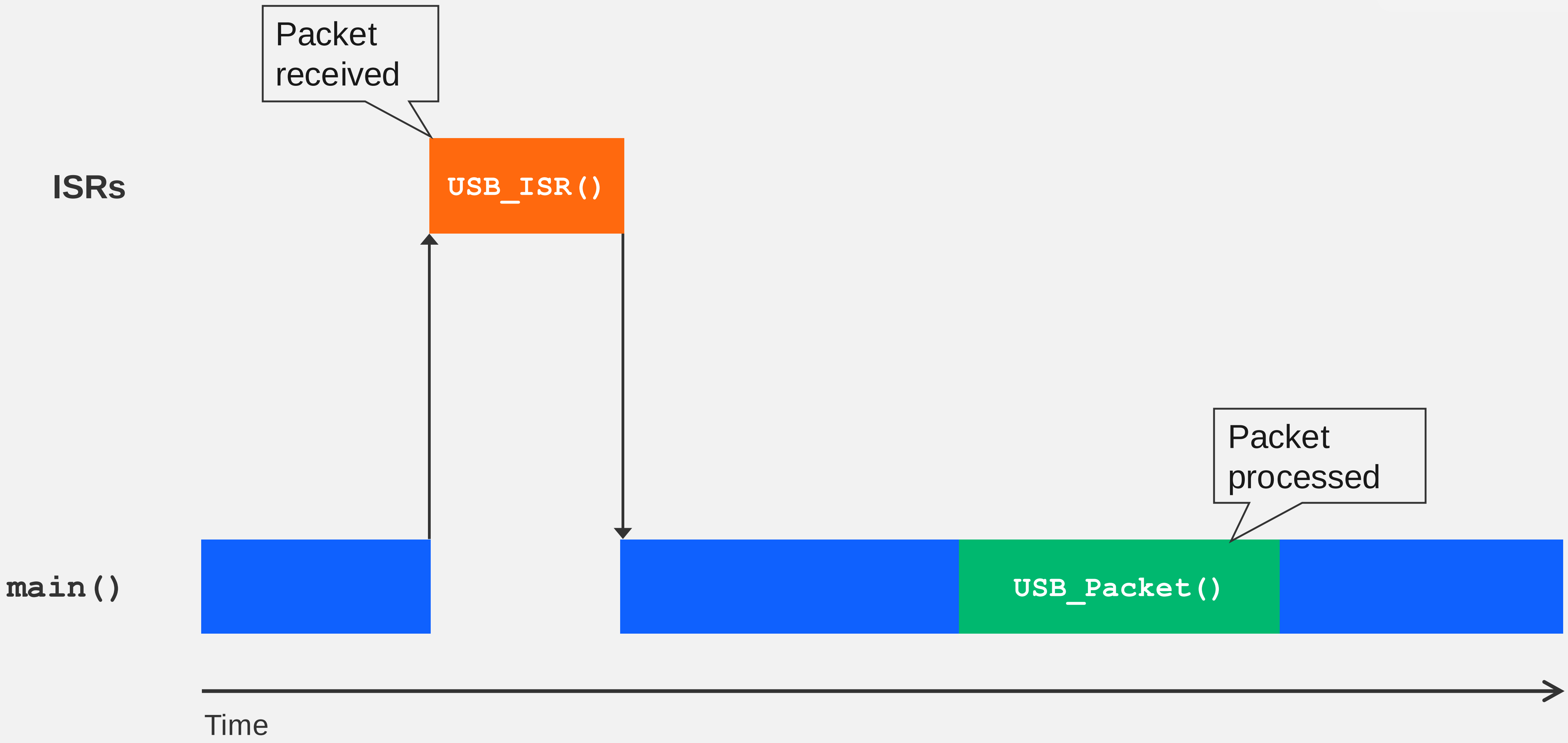
```
void USB_ISR (void)
{
  Clear HW bit;
  Set flag variable;
  …
}
```

Main Loop

```
int main (void)
{
  while (1) {
    USB_Packet();
    ...
  }
}
```

```
void USB_Packet (void)
{
  Check flag variable;
  Process packet;
}
```

Execution Diagram 1 (No Kernel)



Example Application 2 (Kernel-Based)

Same objective as Example 1

- Receive and respond to USB packets

ISR is triggered by packet reception

- Packet not fully processed in ISR
- Kernel function used to signal task

Kernel enables USB task based on ISR signal

- May run immediately after ISR
- Kernel runs other tasks while USB_Task() waits
- Lower priority tasks do not impact packet response time

ISR

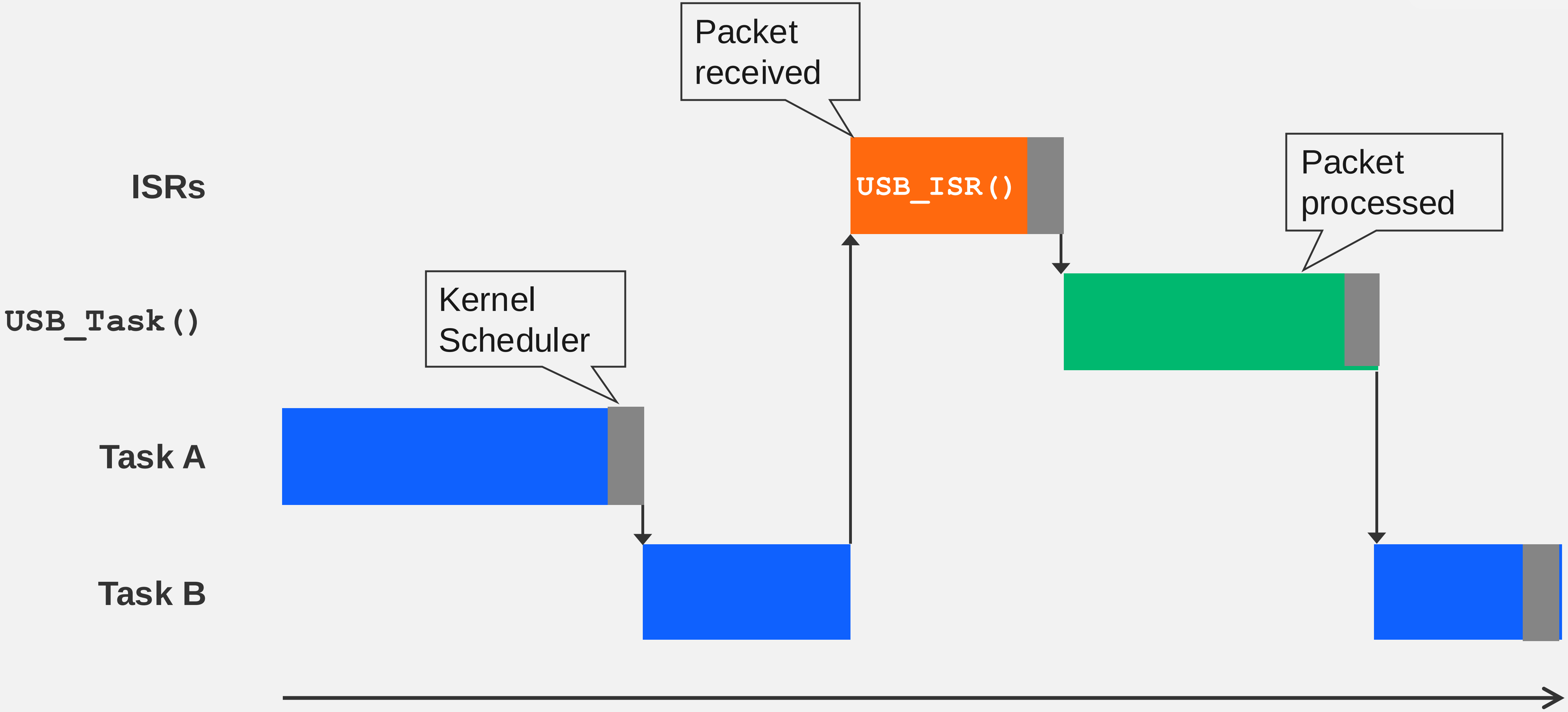
```
void USB_ISR (void)
{
  Clear HW bit;
  Signal task;
  …
}
```

Tasks (Managed by the Kernel)

```
void USB_Task (void* task_arg)
{
  while (1) {
    Wait for signal;
    Process packet;
    ...
  }
}

...
```

Execution Diagram 2 (Kernel-Based)



Full-Featured RTOS-Based Platforms

In the past, “RTOS” and “kernel” tended to be used interchangeably in the embedded space

- Developers moved to an RTOS primarily for multi-task scheduling

As RTOS adoption has expanded, so have the capabilities of the typical RTOS

- Commercial RTOS providers have for years offered their kernels alongside various stacks, middleware
- Open-source RTOSes are now the norm and are often, likewise, combined with other software components
- In many cases, hardware providers, like Silicon Labs, combine an RTOS with a broader software platform

Stacks, services, and middleware components can substantially accelerate development time

- Eliminate the need for application developers to write thousands of lines of complex code
- Make it easier to leverage the full potential of power hardware components

Beyond the Kernel: Power Manager

Coordinates system & app power needs

- Receives EM requirements from drivers, stacks, etc.
- Selects appropriate EM based on input
- Sends EM commands to the device

Enables custom code on transitions

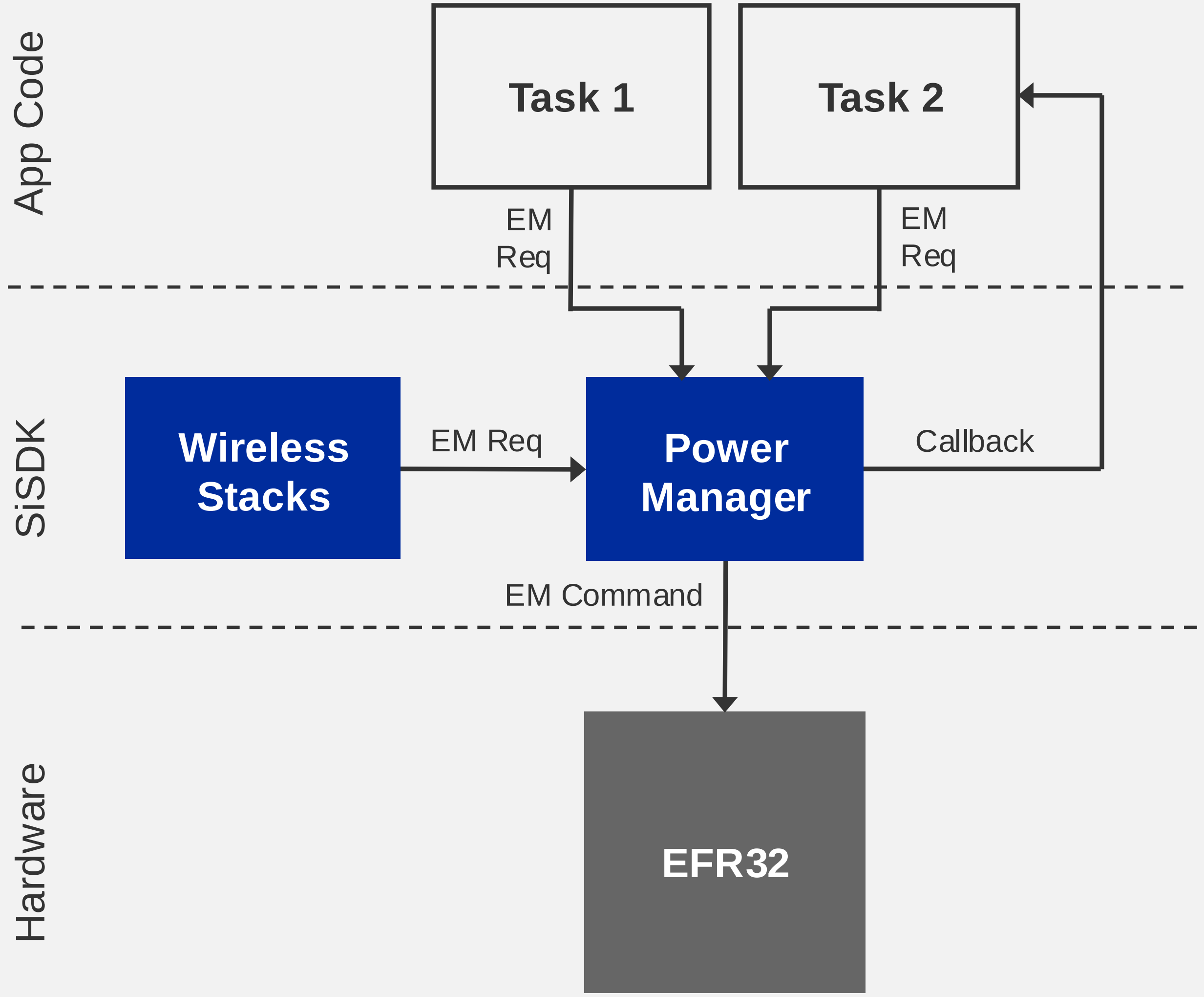
- Various callbacks provided to application

Controls power-hungry resources

- Default state for clocks after certain events

Part of Silicon Lab's software platform

- Integrated with stacks, other system code



Beyond the Kernel: Memory Manager

Simplest versions replace malloc()

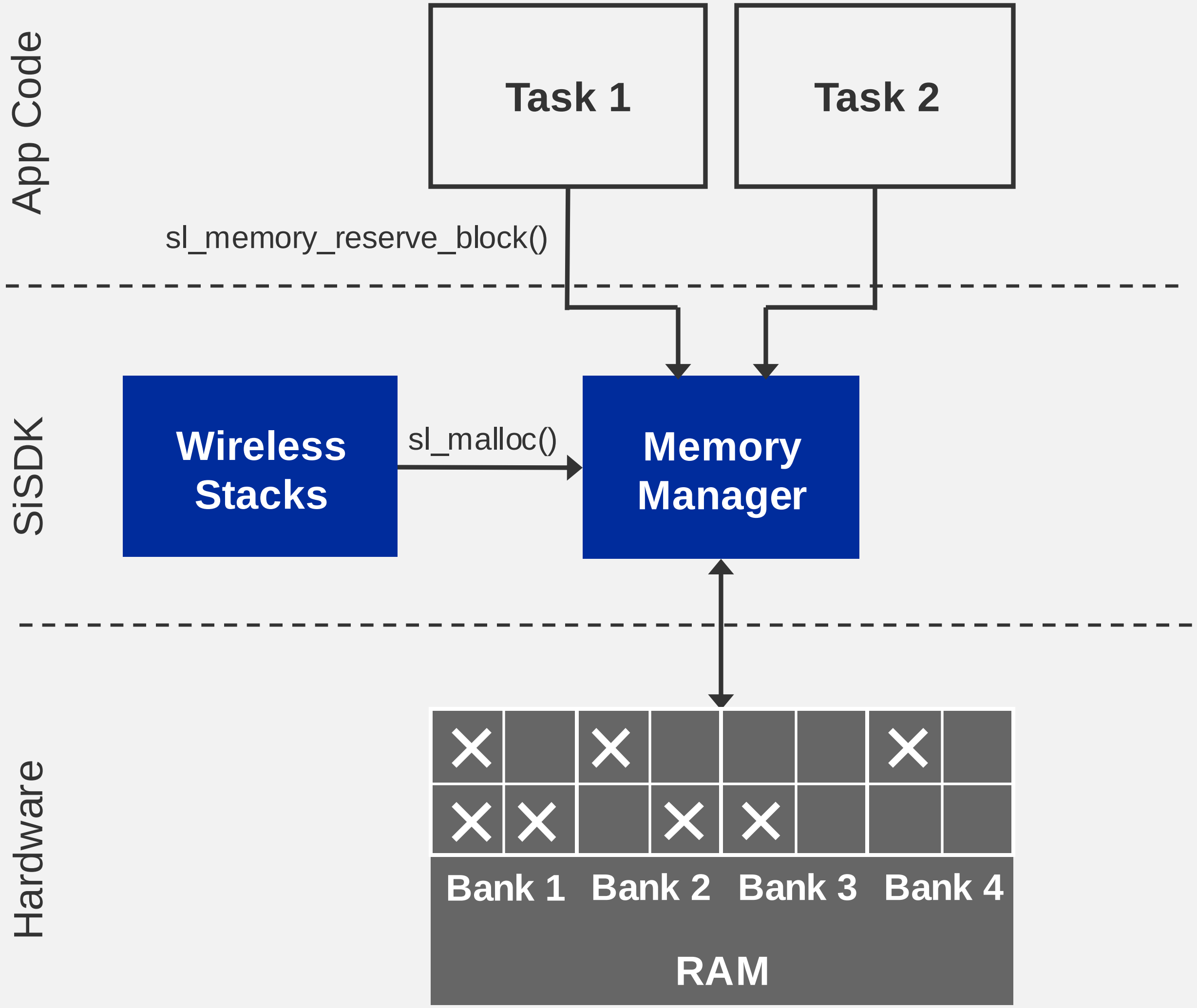
- Enable dynamic memory allocation
- Generally use fixed-size blocks

Silicon Labs offers additional features

- Distinguishes long-term/short-term storage
- Abstracts details of underlying memory
- Facilitates shutdown of unused RAM banks
- Provides app code with detailed statistics

Part of Silicon Labs' software platform

- Integrated with stacks, other system code



Silicon Labs RTOS Option #1: FreeRTOS

Highly popular option with >20-year track record

- Acquired by Amazon in 2017

Kernel is lightweight and efficient

- Supports semaphores, mutexes, queues, other common kernel features
- 5-10 kBytes of Flash and <1 kByte RAM (excluding tasks stacks)
- Context switch time ~100 CPU cycles on Cortex-M

Various connectivity modules implemented for use alongside kernel

- Added following Amazon acquisition

Fully integrated into Silicon Labs SiSDK

- Wireless stacks leverage FreeRTOS functionality to ensure optimal performance in multi-task systems
- Various Amazon connectivity modules also included in SDK

Silicon Labs RTOS Option #2: Zephyr

Established in 2016 by Linux Foundation

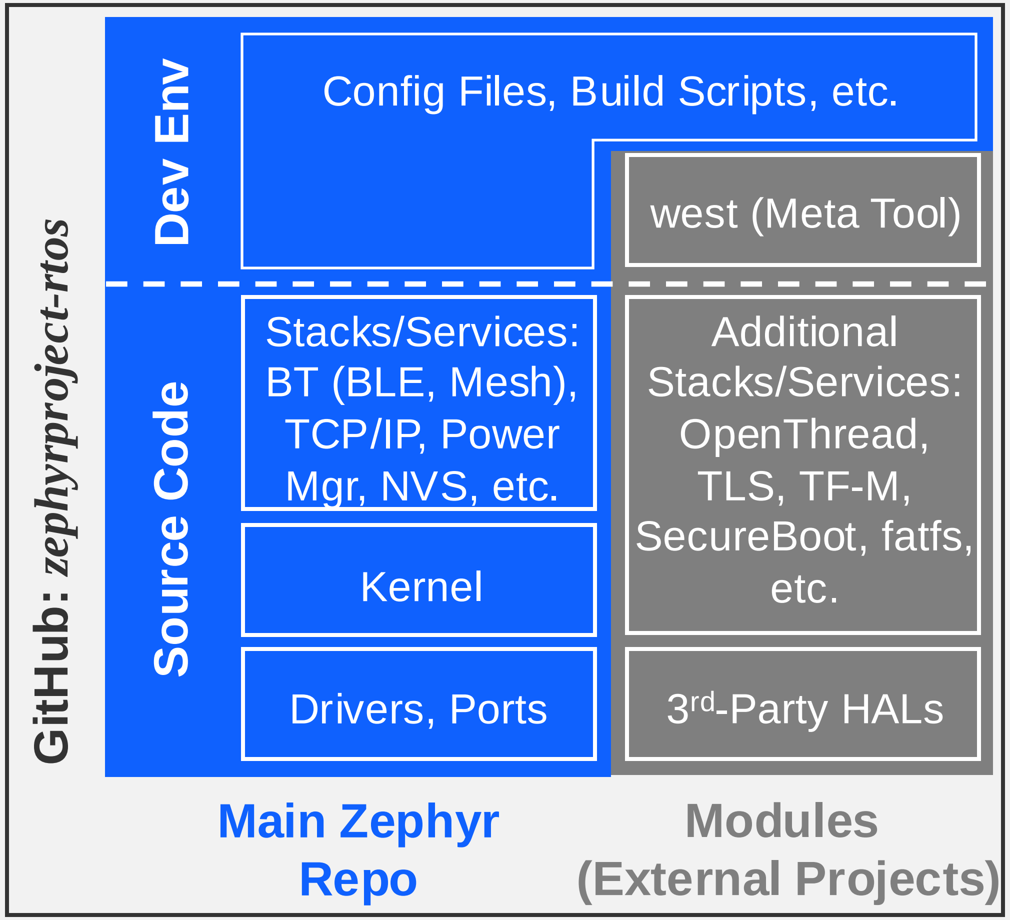
- Open-source RTOS with active community
- Permissive (Apache 2.0) licensing
- Supported Financially by member companies

Combines kernel with full software platform

- Includes stacks, middleware, drivers, etc.
- CL development environment based on west tool

Work ongoing to expand Silicon Labs support

- Silicon Labs joined as Zephyr member in 2021
- Handful of BLE projects released over past few years
- New, full-featured projects coming in 2025
- Focus on BLE and Wi-Fi
- Projects both in public repo and downstream fork



Getting Started with an RTOS on Silicon Labs HW

FreeRTOS

1. **Select a HW kit suitable for your project**
2. **Download and install Simplicity Studio**
3. **Browse for FreeRTOS projects for your HW**
   - FreeRTOS can also be added via Proj. Configurator
   - Additional info: https://www.silabs.com/developers/rtos/freertos

Zephyr

Coming Soon!

Thank You

Conclusion

An RTOS is a system software component that helps manage the underlying hardware

- Key component is a multi-task kernel
- RTOSes may also incorporate drivers, middleware, and stacks

Although there is a learning curve, an RTOS can be a beneficial addition to an IoT project

- Kernel enables efficient use of CPU cycles
- Stacks and middleware components simplify work of application developers

Silicon Labs' RTOS support policy covers both FreeRTOS and Zephyr

- FreeRTOS is fully integrated into SiSDK and used by Silicon Labs' wireless stack
- Expanded Zephyr support is under development and will be offered through Zephyr repo and downstream fork

Getting started with an RTOS may require just a few minutes of time!

- Addition of FreeRTOS to projects automated in Simplicity Studio with configurator
- Zephyr support will be based on existing Getting Started Guide