Internet of Things

Embedded Operating Systems

Ioannis Chatzigiannakis

Sapienza University of Rome
Department of Computer, Control, and Management Engineering (DIAG)

Lecture 3:
Embedded Operating Systems

Objectives of an Operating System

- Make the computer system convenient to use in an efficient manner.
- Hide the details of the hardware resources from the users.
- Provide users a convenient interface to use the computer system.
- Act as an intermediary between the hardware and its users, making it easier for the users to access and use other resources.
- Manage the resources of a computer system.
- Keep track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users.
- Provide efficient and fair sharing of resources among users and programs.

Main elements of an Operating System

Operating System (OS)

A collection of software that manages the hardware resources and provides various service to the users.

1. Processor
   - Performs the data processing functions.
   - Controls the operation of the computer.
2. Main Memory
   - volatile storage.
3. I/O Modules
   - non volatile storage,
   - communications,
   - terminals.
4. System Bus
   - communication among components.

Main services of an Operating System

Operating System (OS)

A collection of software that manages the hardware resources and provides various service to the users.

1. Process management.
   - Load into main memory, execute programs.
2. Resource Allocation
3. Memory Management
   - Allocate/de-allocate memory.
   - Virtual memory.
4. File System
5. Communication
   - Interprocess communication, across Computer Networks.
6. Error detection and response.
7. Authentication, Authorization, Accounting.

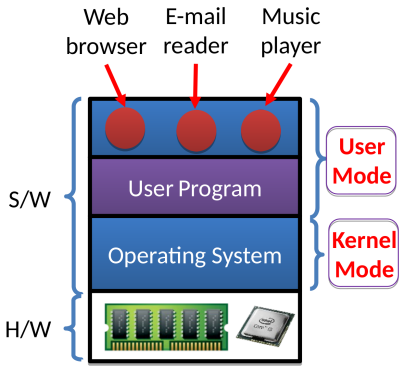
Interacting with the Operating System

Kernel mode:

- has complete access to all the hardware
- can execute any instruction that a machine is capable of executing
- has high privileged (rights)

User mode:

- can execute only subset (few) of the machine instructions
- has less privileged (rights)



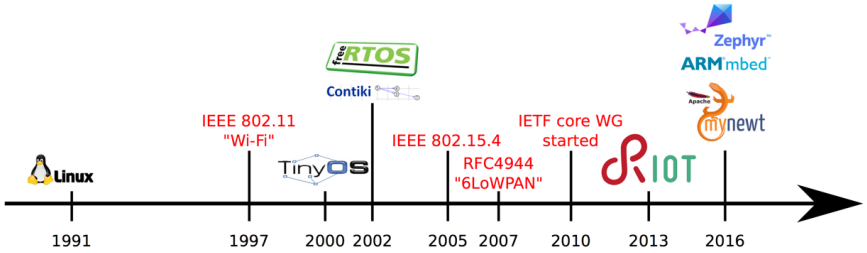
Type of Operating Systems

1. Batch systems: jobs entered as a whole and in sequence.
   - Purpose specific.
2. Interactive systems: allow multiple jobs
   - Purpose specific or General purpose.
   - Turnaround: Faster than batch – slower than real-time.
3. Hybrid systems – combination of batch and interactive.
4. Network operating systems – for network devices.
5. Real-time systems
   - Purpose specific – reliability is critical.
   - Used in time-critical environments – fixed time constraints.
   - Hard vs Soft:
     - Hard: total failure if deadline missed
     - Soft: suffer performance degradation due to a missed deadline.
6. Embedded systems – single machine / purpose specific.
7. Wireless sensor systems – multiple nodes / purpose specific.
8. Smart card operating systems

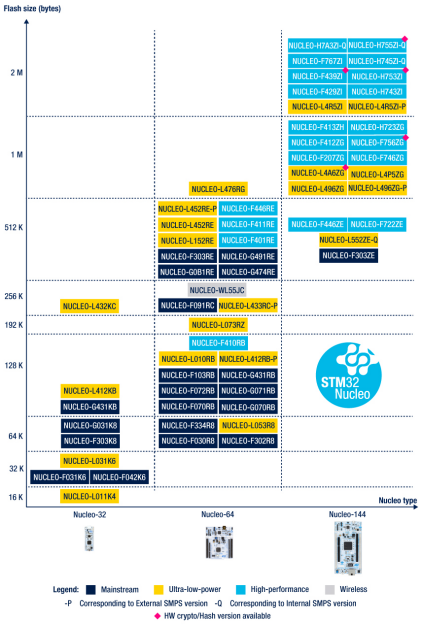
Software platforms for IoT

- Memory: Kilobytes instead of Gigabytes.
- CPU: Megahertz instead of Gigahertz.
- Energy: Milliwatt instead of Watt.
- Potentially limited storage capabilities.
- Probably without direct connectivity to Internet.
- Diverse hardware peripherals that can be connected.
- . . . no standard platform to develop applications.

Operating systems for IoT

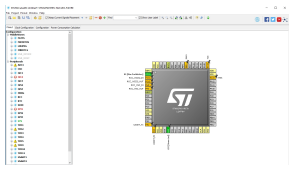


- TinyOS – statically linked applications.
- ContikiOS – dynamic linking of binaries, no thread support.
- FreeRTOS – real-time operating system.
- RIOT – focuses on constrained devices + interoperability.
- ARM mbed – targeting ARM Cortex M series.
- Zephyr – real-time on constrained devices.



STM32 family provides wide range of devices with limited resources:

- Memory as low as 16KB.
- Some may have direct connectivity to Internet.
- Applications can be developed by tools:
  1. STM32CubeMX
  2. Middleware
  3. IDE

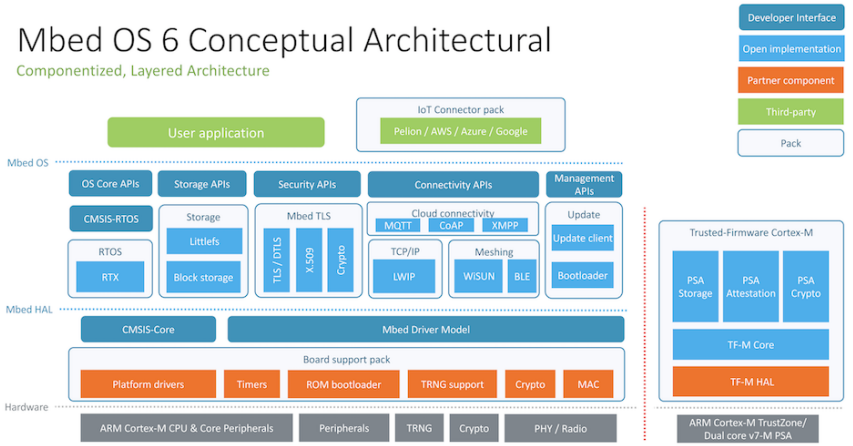


ARM Mbed OS

Arm Mbed OS is a free, open-source embedded operating system designed specifically for the "things" in the Internet of Things. It includes all the features you need to develop a connected product based on an Arm Cortex-M microcontroller (32 bit).

Main Features

- Modular - Ease of Use
  - With a modular library structure, the necessary underlying support for your application will be automatically included on your device.
  - By using the Mbed OS API, your application code can remain clean, portable and simple.
- End to End Security
  - It addresses security in device hardware, software, communication.
- Open Source
  - `https://github.com/ARMmbed/mbed-os`
- Community
  - Big community support and *Forum*.
  - Possibility to use drivers and libraries developed by third party.



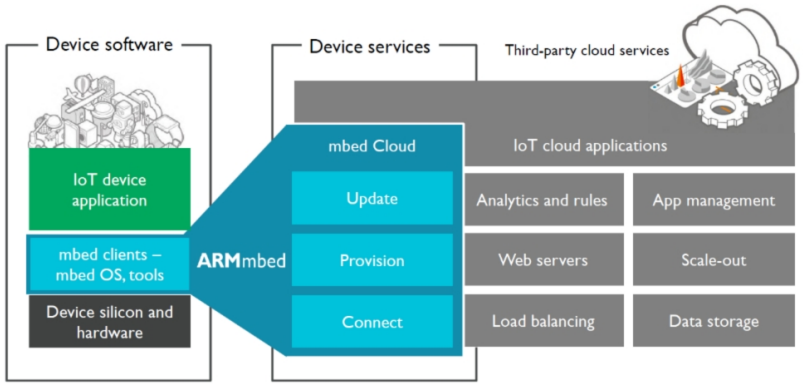
Toolchains and IDE

Two possibilities:

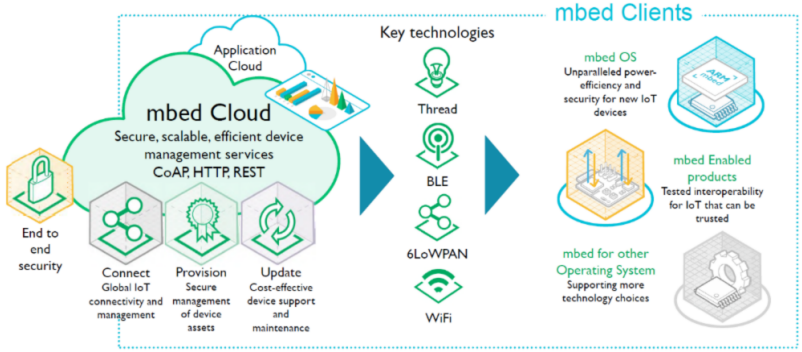
Online IDE, no setup and easiest way to get started.

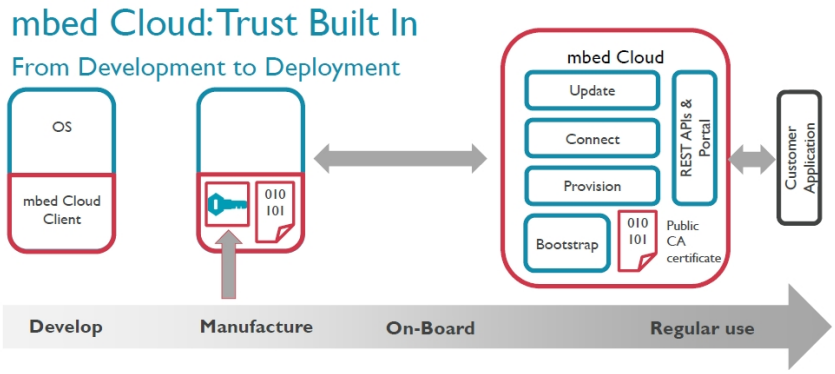
Offline toolchains, (Arm Compiler 6, GCC and IAR) that support Arm Mbed CLI.

ARM MCU to Cloud



ARM MBED Cloud Platform





mbed Cloud enables the business of IoT by empowering customers to develop, deploy at the factory, bring on-board and maintain connected products securely and efficiently.

Zephyr OS

Zephyr OS is a scalable, real-time operating system (RTOS) for embedded devices.

- Open source.
- Cross-architecture with broad SoC and development board support.

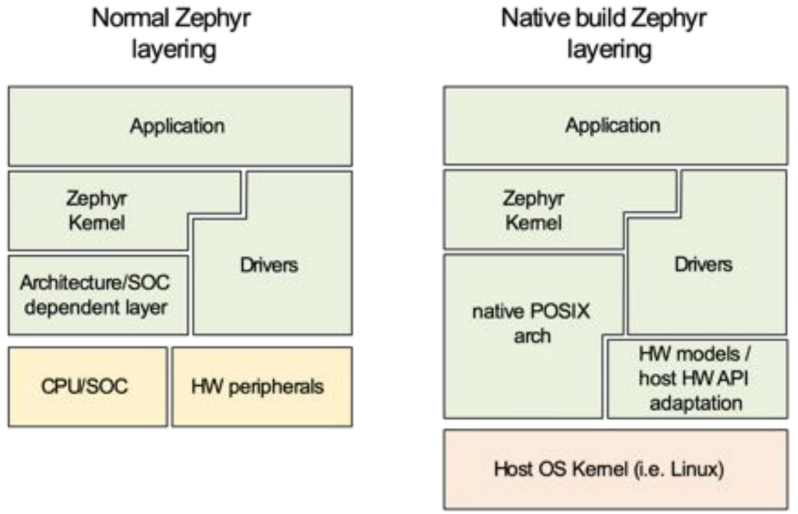
Main Features

- Provide an OS that runs best on MCUs for wearable and IoT devices, where the cost of the silicon is minimal.
- Highly Configurable.
- Kernel mode only
  - Nanokernel – Limited functionality targeting small footprint (below 10k)
  - Microkernel (superset of nanokernel): with additional functionality and features.
- No user space and no dynamic runtimes
- Memory and Resources are typically statically allocated
- Cross architecture (IA32, ARM, ARC)

Native Execution

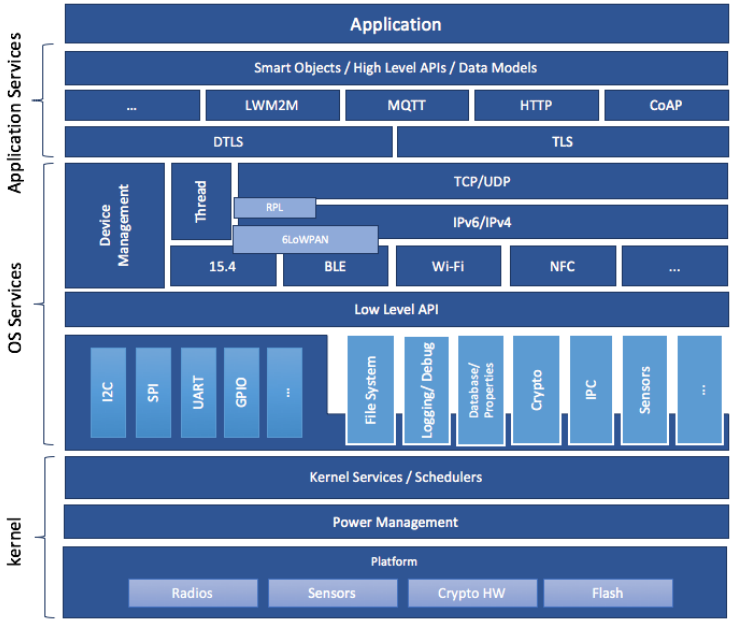
- Build Zephyr as native Linux application
- Enable large scale simulation of network or Bluetooth tests without involving HW
- Improve test coverage of application layers
- Use any native tools available for debugging and profiling
- Develop GUI applications entirely on the desktop
- Optionally connect to real devices with TCP/IP, Bluetooth, and CAN
- Reduce requirements for HW test platforms during development

Native vs Normal



Zephyr Architecture

- Highly Configurable, Highly Modular
- Cooperative and Preemptive Threading
- Memory and Resources are typically statically allocated
- Integrated device driver interface
- Memory Protection: Stack overflow protection, Kernel object and device driver permission tracking, Thread isolation
- Bluetooth Low Energy (BLE 5.1) with both controller and host, BLE Mesh
- 802.15.4 OpenThread
- Native, fully featured and optimized networking stack



Zephyr Ecosystem

Zephyr OS

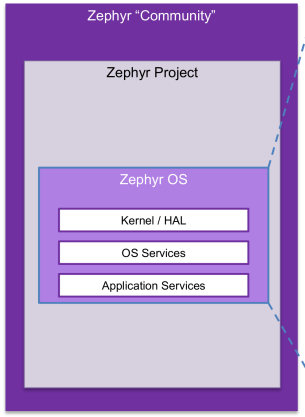
- The kernel and HAL
- OS Services such as IPC, Logging, file systems, crypto

Zephyr Project

- SDK, west, tools and development environment
- Additional middleware and features
- Device Management and Bootloader

Zephyr Community

- 3rd Party modules and libraries
- Support for Zephyr in 3rd party projects, for example: micro-ROS, Tensorflow LITE, Micropython, Jerryscript



Kernel / HAL

- Scheduler
- Kernel objects and services
- low-level architecture and board support
- power management hooks and low level interfaces to hardware

OS Services and Low level APIs

- Platform specific drivers
- Generic implementation of I/O APIs
- File systems, Logging, Debugging and IPC
- Cryptography Services
- Networking and Connectivity
- Device Management

Application Services

- High Level APIs
- Access to standardized data models
- High Level networking protocols

Zephyr Project Governance

Goal: Separate business decisions from meritocracy, technical decisions

Governing Board

- Decides project goals and strategic objectives
- Makes business , marketing and legal decisions
- Prioritizes investments and oversees budget
- Oversees marketing such as PR/AR, branding, others
- Identifies member requirements

Technical Steering Committee

- Serves as the highest technical decision body consisting of project maintainers and voting members
- Sets technical direction for the project
- Coordinates X-community collaboration
  - Sets up new projects
  - Coordinates releases
  - Enforces development processes
  - Moderates working groups
- Oversees relationships with other relevant projects

Community

- Code base open to all contributors, need not be a member to contribute.
- Path to committer and maintainer status through peer assessed merit of contributions and code reviews
- Ecosystem enablement

Goals for IoT Operating Systems

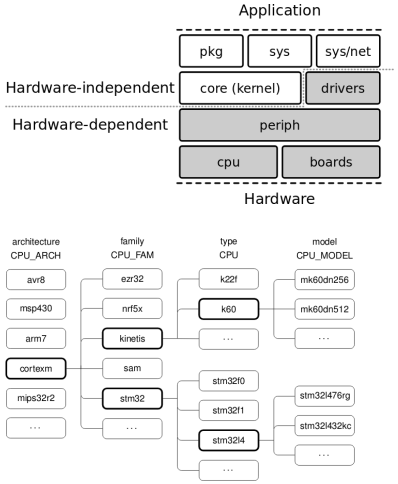
- Driven by a grassroots community
  - instead from the hardware providers/manufacturers.
  - Linux-like approach.
  - Open source + Free licensed under LGPLv2.1.
  - Less garbage with less IoT device lock-down
- Functionalities of a full-fledged operating system.
  - Needed memory & energy efficiency to fit IoT devices.
  - Faster innovation by spreading IoT software dev. costs
- Setting Privace & Security at core:
  1. Long-term IoT software robustness & security
  2. Trust, transparency & protection of IoT users' privacy

Interoperability Goal

- Modular structure, adaptive to diverse hardware
  - Micro-kernel architecture (contrary to Linux)
    - minimal requirements around 1kB RAM
    - 2.8kB RAM, 3.2kB ROM on 32-bit Cortex-M
  - support for 50+ different IoT boards/devices.
- Efficient Hardware Abstraction Layer (HAL)
  - minimized hardware-dependent code.
  - support for 8/16/32 bit, ARM, AVR, MIPS.
  - Supported vendors: Microchip, NXP, STMicroelectronics, Nordic, TI, ESP, RISC-V, etc.
  - Large list of sensors and actuators supported (e.g drivers)
  - native board: run RIOT as process on your computer
  - +100 boards supported
- Compliance with common system standards
  - POSIX sockets, pthreads
  - standard C, C++ application coding

Hardware Abstraction Layer

- 3 blocks: boards, cpus, drivers
- CPUs are organized as follows:
  - architecture (ARM) → family (stm32) → type (stm32l4) → model (stm32l476rg)
- Generic API for cpu peripherals (gpio, uart, spi, pwm, etc)
  - same API for all architectures
- Only based on vendor header files (CMSIS)
  - less code duplication, more efficient but more work

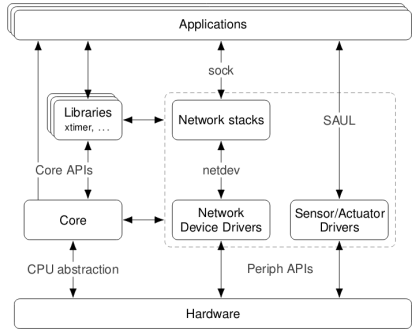


One application → one board → one cpu model

Modular Operating System

Features are libraries → only build what's required

- xtimer – high-level timer subsystem
  - Full abstraction from the hardware timer
  - Can set callbacks, put a thread to sleep, etc
- shell – interactive command line
- Others: crypto, fmt, math, etc



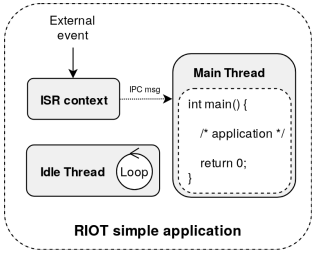
Also: External packages provided and maintained by third parties.

Core: Real-time scheduler

- Tickless scheduler – energy efficient.
- Fixed priorities – deterministic.
  - Highest priority thread runs until finished or blocked.
  - 16 priority levels.
  - the lower level the higher priority.
  - Idle thread has priority 15.
  - Main thread has priority 7.
- Preemption O(1) operations – real-time.
  - Interrupt service runtime (ISR) can preempt any thread at any time
  - If all threads are blocked:
    - Switch to special IDLE thread
    - Goes into lowest possible power mode
- Low latency interrupt handler – reactivity.

Core: Multi-Threading and IPC

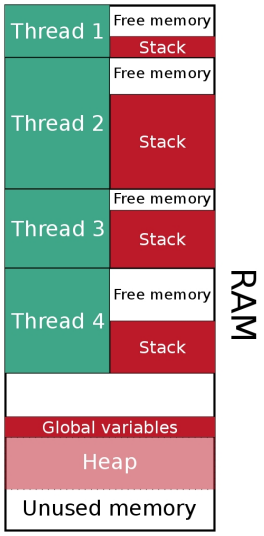
- Separate thread contexts with separate thread memory stack.
- Minimal thread control block (TCB).
- Thread synchronization using mutexes, semaphores and messaging.



2 threads by default:

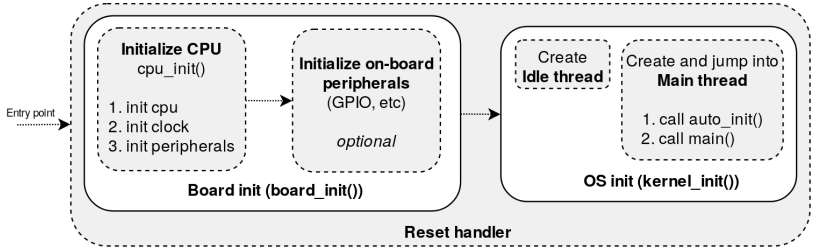
1. the main thread: running the main function
2. the idle thread:
   - lowest priority
     → fallback thread when all other threads are blocked or terminated
   - switches the system to low-power mode

Core: Memory Management



- Threads have their own memory stack
- The stack also contains the thread control block (tcb)
- There is no memory protection
- A stack overflow can destroy another stack

Overview of the boot sequence



1. `board_init()` is implemented in `boards/<board name>/board.c` file
2. `cpu_unit()` is implemented in `cpu/<cpu model>/cpu.c` file
3. `kernel_init()` is implemented in `core/kernel_init.c` file

Example for ARM Cortex-M:

- the entry point is `reset_handler_default`
- implemented in `cpu/cortexm_common/vectors_cortexm.c`

Network Stacks

IP oriented stacks: designed for Ethernet, WiFi, 802.15.4 networks

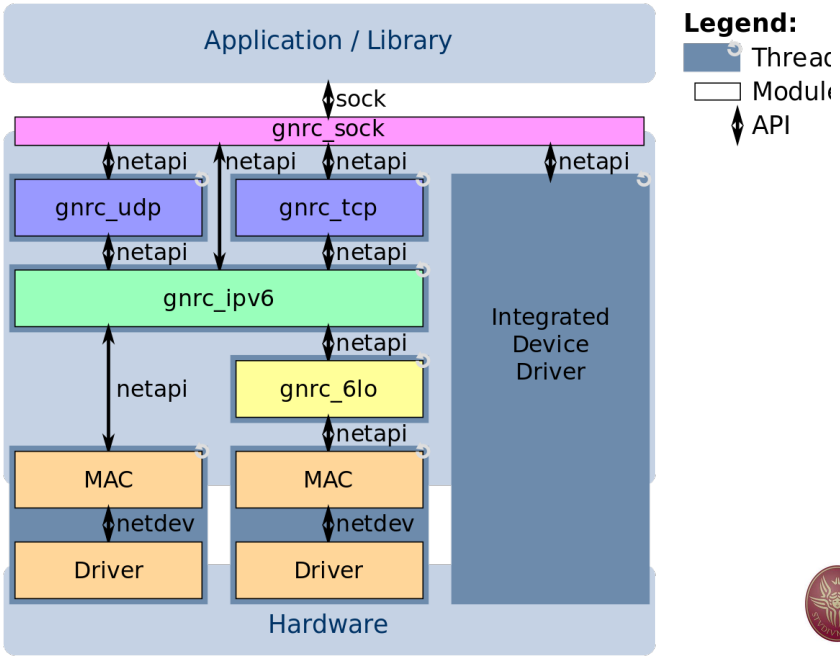
- GNRC: the in-house 802.15.4/6LowPAN/IPv6 stack of RIOT
- Thread: 802.15.4 IPv6 stack provided by the ThreadGroup
- OpenWSN (experimental): a deterministic MAC layer implementing the IEEE 802.15.4e TSCH protocol

Other IPv6 stacks:

- lwIP: full-featured network stack designed for low memory consumption
- emb6: A fork of Contiki network stack that can be used without proto-threads
- IPv6 over BLE, NimBLE stack support

Other Network Support

- In-house Controller Area Network (CAN)
- LoRaWAN stack – Compliant with LoRaWAN 1.0.2
- SigFox support for ATA8520e modules
- Full featured USB stack (CDC-ACM, CDC-ECM, etc)
- SUIT: Standard and secure software update implementation



Source code organization

- boards: board specific definitions, cpu model, clock, peripherals config, documentation, serial and flasher config
- core: kernel initialization, thread, ipc
- cpu: support for microcontroller (cmsis, definitions, peripheral drivers), entry point (`_reset_handler_default_`)
- dist: management and utility tools (script, ci, static checkers, etc)
- doc: doxygen documentation
- drivers: high-level device drivers (sensors, actuators, radios), HAL API
- examples: sample applications
- makefiles: build and management system makefiles
- pkg: external packages
- sys: system libraries, network, shell, xtimer, etc
- tests: unittests, test applications (can be used as examples)