Arduino cores, frameworks, mbedOS, and ‘bare metal’

In working with the Nano 33 BLE Sense, you have interacted with some of the underlying layers between your application and the board’s physical hardware. Here, we quickly define some key terms (Arduino core, embedded frameworks, mbedOS, RTOS, and bare metal) before discussing implications of these concepts to your application.

What is an Arduino core?

An Arduino “core” is central to a particular microcontroller’s compatibility with the Arduino framework. You can think of it as a low-level software API for a specific chip or family, like AVR or SAMD. Since each microcontroller has its own architecture, the concomitant core acts as an abstraction layer between your application and the physical hardware. In this context, libraries can be thought of as extensions to the core that are portable between chips or board variants for a particular chip.

More granularly, each Arduino core contains a pair of generic files: `arduino.h` that handles declarations for what are often called ‘built-in’ functions like `pinMode()`, `digitalWrite()`, `analogRead()`, `delay()`, et cetera as well as macros for constants, like `HIGH` or `INPUT_PULLUP`, and operations like `bitToggle()` or `abs()`; `main.cpp` declares the basic program structure for a sketch, as it relates to setup() and loop(). We should note that `arduino.h` also sets the stage for board-specific pin mappings defined in `pins_arduino.h` at `<architecture>/variants/<board name>`.

The remainder of a core is largely architecture specific. There is a collection of C files with names `wiring.c` and `wiring_<type>.c` that define hardware initialization, timing, and functions related to IO of various types, like `analogRead()`. This nomenclature is a nod to the Wiring API that Arduino utilizes. There are also header and C++ files for serial classes Stream, Print, and HardwareSerial, alongside a light USB stack. If you’re interested in digging deeper into a specific core check out this appendix document.

What is an embedded framework?

Loosely, to account for variation between manifestations, an embedded framework is a collection of development tools for microcontrollers that may or may not be portable and often serve as some combination of low-level API, software development kit (SDK), and hardware abstraction layer (HAL). For instance, the Arduino framework comprises its integrated development environment (IDE), built-in functions (defined by cores), as well as library extensions. As an open source project, the Arduino framework is perhaps a bit more difficult to put bounds around, given its reliance on related projects, namely Wiring and its predecessor Processing, born out of the MIT Media Lab. Here are some other examples:

CMSIS - Arm Cortex Microcontroller Software Interface Standard
FreeRTOS - a real-time operating system (RTOS) kernel for embedded devices
Mbed - an embedded RTOS for Internet-of-Things, drivers for I/O devices
STM32Cube - HAL between STM32 devices and other libraries

If you work with more than one framework, we recommend PlatformIO, an extension of Visual Studio Code, that enables you to call on many frameworks, with a unified workflow.

What is an RTOS?

Okay, so two of the examples we provide for embedded frameworks make mention of a real-time operating system, or RTOS. In a circular definition, an RTOS is an operating system that serves real-time applications. More helpfully, an embedded RTOS is an extremely light-weight system software that organizes the attention of microcontroller processing cores, often singular, to minimize unnecessary task switching and to prioritize time-sensitive computation so that the processing time required for a given task is ideally less than the interval to the next input. We won't dwell on the details here, but you can learn more about task scheduling, apparent multitasking (threads), and other RTOS fundamentals, here.

What is Mbed and mbedOS?

Mbed is a competing embedded framework to Arduino, with a fair bit of overlap. A glance at the 'What is Mbed?' section of their landing page illustrates this. Both are centered around a C++ API and feature IDEs, example code, libraries, and drivers for common components. Perhaps most striking is the difference in accessibility and other professional-grade considerations Mbed makes, like security. Notably, while mbedOS can be and usually is an RTOS, for applications that require thread management, there is also a limited 'bare metal' profile that you can utilize to cut down on the memory utilization tied to RTOS overhead.

Why are we talking about mbedOS? Well, the Nano 33 BLE Sense runs on it. Here is an Arduino blog post from Martino Facchin, the head of Arduino's firmware development team, that explains the Nano 33 BLE Sense unique core, and it's reliance on mbedOS.

What is bare metal?

You may happen upon variations of this definition, but at a high, inclusive level, 'bare metal' programming in an embedded systems context means utilizing microcontroller hardware independent of an operating system or scheduler, at the least. Typically, there is a base super loop, not unlike the Arduino sketch structure, and all processing is defined by your application. In many cases, this also means your code is written independent of pre-existing frameworks. This is where you'll find some variation in definitions. The coloquial spirit of bare metal programming is counter to most developer mindsets, in that because microcontroller tasks are often simple but mission critical, their developers want strict control over processing that precludes reliance on abstraction.

A ‘true’ bare metal project, then, will have a developer that either writes or, if supplemented by a light-weight framework, fully understands each layer, from the physical hardware on up, using a MCU/SoC datasheet as their guide to register definitions and hardware architecture.

Implications for your application

As you can imagine, your choice of framework (or lack thereof) should stem from the needs of your application. If you have no hard time-constraints, an RTOS could be eating away at precious resources that may be constrained. However, there are obvious challenges to developing a bare metal solution, which can largely be summarized by ‘recreating known solutions.’ If your task is simple, time-insensitive, mission critical, or needs to be deployed on very constrained hardware, a bare metal approach may be appropriate, assuming you have the knowledge and time required for development. If you have a complex web of tasks in front of you that could be organized by a scheduler, an RTOS is really a no-brainer. In most cases, there is room for innovation in the middle, where you place trust in proven frameworks and libraries, but manage processing attention in a simple base loop, perhaps organized as a finite state machine, or FSM.