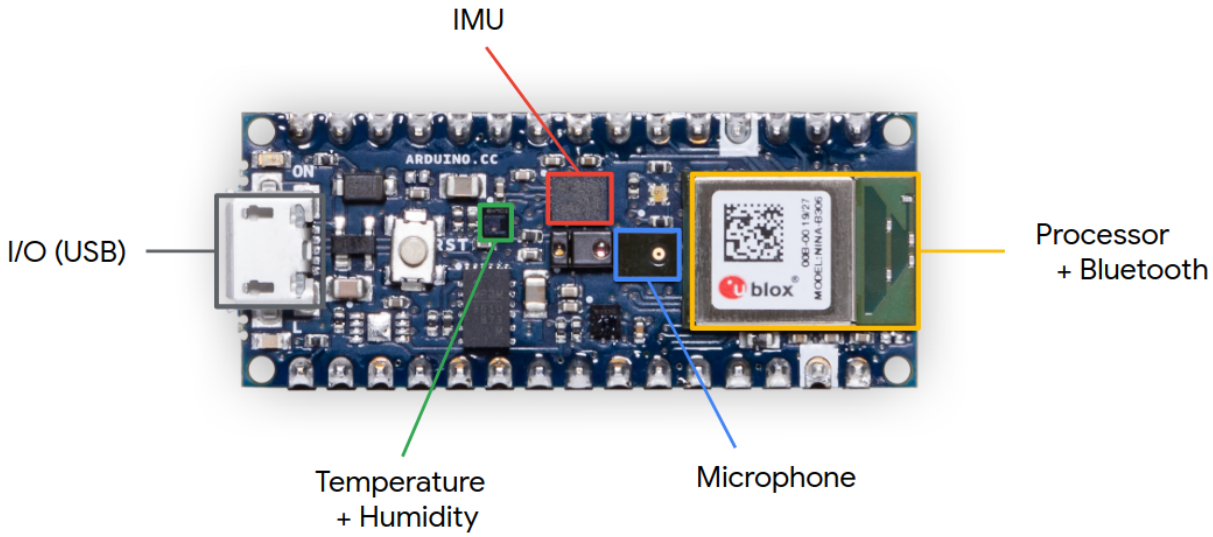
Embedded Systems

In this reading, we will explore the diversity of embedded systems.

If you take a few minutes to look around the room you're in, you'll no doubt recognize the proliferation of embedded hardware within our world. Embedded systems are increasingly ubiquitous because they serve to complete specific computational tasks within their environment, allowing us to not only collect all sorts of data from distributed sensors but also to process the resulting information locally and act accordingly. That we can accomplish this in situ, or within the environment that this hardware is embedded within, at a fraction of the cost and physical scale of most general-purpose computing hardware, opens the door to many advancements in automation and generally to the creation of smart, connected things.

While the specifications and capabilities of a given embedded system ought to be tailored for its application, in general, we can look to the common construct for an embedded system that links sensing to processing and later actuation through the process of transduction, the conversion of one form of energy into another, for throughlines. In sense, we convert energy from a physical phenomenon into an electrical signal we can go on to digitize and compute with. In actuation, the converse. In this course, we'll focus predominantly on the central element of said construct: the computing hardware, and review specifications and technical considerations that vary between implementations of embedded systems.



Development Boards

In the context of this course, we have talked about and deployed a microcontroller unit (or MCU) development board from Arduino, the Nano 33 BLE Sense. Here, we distinguish the nRF52840 MCU from the development board it is soldered to. What's interesting about this board is that despite its relatively small size, it is jam-packed with many of the representative elements of an embedded system, all on one printed circuit board (PCB): a collection of surface-mount sensors, a form of an actuator in the on-board programmable RGB LED, and an integrated MCU-BLE module, where BLE is an acronym for a branch of the Bluetooth standard called Bluetooth Low Energy. Further still, the PCB that the MCU is soldered to serves to 'break out' additional IO from the controller to provide interfaces for external, off-board modules and connections to daughter boards. So, while in many ways, the Nano 33 BLE Sense is representative of an embedded system, it is separated from commercial implementations in that it serves no particular purpose, other than the open-ended development of an application that is. Put another way, a development board has the advantage of enabling many potential use cases. Further, you don't have to go through the process of designing circuitry, capturing schematics, or physical layout for such a board to get started prototyping a concept that could go on to be manufactured with specific intent later. The tradeoffs you make in employing a development board often involve board size and cost.

Board Size

Board size, and shape, have fairly meaningful implications for use in the field, given these simple parameters constrain where and how a system can be deployed. Some environments or contexts within which we'd like to embed a system are forgiving (like the Google Home, say), but other manifestations (smart glasses, for example) depend wholly on the requisite hardware being quite tiny and perhaps of unique form. Somewhat obviously, board size is determined by the number and physical size, or package, of the components that live upon it. At a minimum, an embedded system must include a MCU chip alongside a power source, often a lithium polymer battery, and power circuitry. The nRF52840 MCU on the Nano 33 BLE Sense lives within a MCU-BLE module, the U-Blox NINA-B306, which spans 10 by 15 mm, dimensions that include a trace antenna. The MPM3610 step-down converter used to down-regulate the 5V delivered to the board over USB occupies 3 by 5 mm, alongside small SMD passives. The entire board, meanwhile, spans 18 by 45 mm. While impressive, clearly a purpose-designed PCB could remove some of the sensors and IO breakout unnecessary for a specific application to reduce the overall scale even further, perhaps by about 60 to 70%.

Cost

Unsurprisingly, the cost of a development board scales with its MCU's compute capability and general feature set. In selecting the appropriate MCU for an application, it is important to remember that MCUs are often deployed to complete specific tasks, where the complexity they will face is predetermined or constrained. For very simple tasks, we highlight the ATTINY85 (an 8-bit processor with 8 kB of flash) that can, depending on the package selected, cost well less than $1 EA and even less at scale, perfect for simple computing requirements. In the context of TinyML, the AI-capable NINA-B306 module featuring the Nordic Semiconductor nRF52840 chip (an ARM Cortex M4) and all-in-one BLE hardware (including antenna) costs about $10 EA and is more generally representative. In view of the on-board sensors and design costs, the Arduino Nano 33 BLE Sense costs just over $30, not even an hour's time of an electrical engineer who might design such a board — a tremendous value. In general, the boards we originally considered for this class span from $2 (BluePill) to $54 (Disco-F746NG).

You can find a list of the boards we considered in the table below. Ultimately, our staff selected the Arduino Nano 33 BLE Sense for its versatility, in providing a large selection of on-board sensors, accessible IO via breakout pins, and a BLE module for projects that involve wireless communication, with reasonably representative compute specifications for resource-constrained hardware. We'll explore the compute specifications in a bit more detail in the next video and reading.

TinyML Development Board Comparison ◽ **officially TFLM supported** ▫ **unofficially compatible boards**

Board	MCU	CPU	Clock	Memory	IO	Sensor(s)	Radio
Arduino Nano 33 BLE Sense	Nordic nRF52840	32-bit ARM Cortex-M4F	64 MHz	1 MB flash 256 kB RAM	x8 12-bit ADCs x14 DIO UART, I2C, SPI	Mic, IMU, temp, humidity, gesture, pressure, proximity, brightness, color	BLE

Espressif ESP32-DevKitC	ESP32 D0WDQ6	32-bit, 2-core Xtensa LX6	240 MHz	4 MB flash 520 kB RAM	x18 12-bit ADCs x34 DIO** UART, I2C, SPI	Hall effect, capacitive touch***	WiFi, BLE
Espressif EYE	ESP32 D0WD	32-bit, 2-core Xtensa LX6	240 MHz	4 MB flash* 520 kB RAM	SPI via surface pads	Mic, camera	WiFi, BLE
Teensy 4.0	NXP iMXRT1062	32-bit ARM Cortex-M7	600 MHz	2 MB flash 1 MB RAM	x14 10-bit ADCs x40 DIO** UART, I2C, SPI	Internal temperature, capacitive touch	None
MAX32630FTHR	Maxim MAX32620	32-bit ARM Cortex-M4F	96 MHz	2 MB flash 512 kB RAM	x4 10-bit ADCs x16 DIO UART, I2C, SPI	Accelerometer, gyroscope	BLE

*This board also features 4 MB flash and 8 MB of PSRAM external to the MCU,
**shared programmable functions, ***with external touchpads

Board	ASIC	DSP	Clock	Memory	IO	Sensor(s)	Radio

Himax WiseEye WE-I Plus EVB	HX6537-A	32-bit ARC EM9D DSP	400 MHz	2 MB flash 2 MB RAM	x3 DIO I2C	Mic, accelerometer, camera	None

We include the Himax WiseEye as an officially supported example of hardware optimization for TensorFlow Lite that calls on an application-specific integrated circuit (ASIC).

Embedded Microcontrollers

In this reading, we will explore the diversity of embedded microcontrollers.

Microcontrollers sit at the bottom of the computing hardware hierarchy, at least in terms of computational capability. You might be wondering what differentiates an MCU from the CPU in the machine in front of you. Well, in some ways an MCU and CPU are quite similar, namely because an MCU is the integration of a low-performance CPU and other peripherals within a single chip. In this way, MCUs are also similar to, albeit less sophisticated than, a System-on-a-Chip (SoC). Typical peripherals integrated into an MCU include memory, analog-to-digital converters (ADC), digital-to-analog converters (DAC), timers, counters, general-purpose IO (GPIO), pulse-width modulation (PWM), direct memory access (DMA), interrupt managers, and serial protocol controllers. We can look at this integration and draw two conclusions: MCUs emphasize connections to the environments they live within and require that all on-board hardware be as tightly packaged as possible. Much like a CPU is seated upon a motherboard, MCUs are chips with packages soldered to much smaller PCBs that, at a minimum, must have supporting power circuitry and a programming interface. The latter prerequisite can be avoided if the MCU is programmed prior to assembly, but this is unusual, at least in development. Beyond this, the specifics of a custom embedded system are contingent upon the needs of the application. In leveraging a development board, you'll want to center your selection on the meeting, if not exceeding, the same requirements. We'll walk you through a comparison of MCU development boards and specifications in a moment.

To return to the computing hardware hierarchy: MCUs are clearly outstripped by the performance of your average personal computer, as well as computing clusters. They do fall somewhat adjacent to, if not arguably short of, what we'll call 'intermediate' computing in the form of single-board computers (SBCs), like the Raspberry Pi. SBCs and MCU development boards are similar enough conceptually and in terms of computing power but are differentiated in that SBCs aim to be low-power, single board computers with proper operating systems and IO typical of personal computers, but

often lack the peripherals we've listed above, making them ill-suited in the context of embedded systems design. Looking at the computing hierarchy we've put forward (clusters > computers > intermediate computing > microcontrollers), you may wonder why exactly so much emphasis is placed on developing advanced software solutions for MCUs, like deploying deep learning models at the edge. Conventional wisdom would suggest that increasingly powerful computing hardware is required for and enables increasingly complex software applications. In recent years, however, there is an incentive for us to develop solutions that allow microcontrollers to take on such complex tasks to realize visions of distributed sensing and computation, where we need to strike a balance between cost (which can beget ubiquity), performance, and power efficiency. To put a name to such an incentive: we can compare the power required to transmit a stream of data from some remote controller to a server uplink and find that this energy requirement greatly outstrips the power we would need to perform a similar analysis on board. Further, there are complicated ethical and policy issues enveloping the deployment of distributed sensors that capture and then share streams of possibly identifying information, rather than de-identified reports of model returns. As such, we have seen MCUs take on increasingly complex tasks through the clever implementation of software, coupled with ever-advancing architectures that enable more powerful processing in even tinier packages.

Compute Capability

Microcontroller units have varied CPU architectures. Of these, chipmakers can license the 32-bit ARM Cortex M architecture, making modifications from this common starting point to introduce features and differentiate their products. For instance, the Nano 33 BLE Sense calls on a U-Blox NINA-B306, which is a stand-alone MCU-BLE module, that integrates the nRF52840 MCU from Nordic Semiconductor, which licensed the underlying architecture from ARM, as the nRF52840 is an ARM Cortex-M4. While this is generally representative of the modern controller, it's worth noting that there is a great deal of variation in compute capability, with processing cores that step all the way down to 8-bits, carrying modern relevance in simple applications.

As with other computing hardware, a faster MCU clock enables greater temporal performance, at the cost of diminishing power efficiency. Typical clock speeds sit between 50 and 500 MHz. For example, the Nano 33 BLE Sense is clocked at 64 MHz. We've deliberately said 'MCU clock,' here, rather than CPU clock, since the core clock of an MCU enables functions beyond just the CPU and trickles down to other peripherals (ADCs, DACs, et cetera) via scalars so that a faster clock can not only enable faster processing but also create headroom (bandwidth) for non-compute functions as well. For instance, while a less than 0.5 GHz CPU clock might not sound impressive, this can translate to theoretical sample intervals for analog-to-digital

conversion on the order of tens of nanoseconds, which is more than sufficient for most physical phenomena of interest. Furthermore, lower speeds may actually be beneficial, somewhat counterintuitively, for timing longer intervals accurately.

Ultimately, the best processing core for your application will depend on the complexity of the task, its performance requirement, and temporal dynamics. One thing that MCUs are known for is their role in facilitating real or near real-time responses to events via interrupts and event systems. The latency of these responses will of course be tied to the performance specifications for the controller. Another thing that MCUs are known for is lying largely dormant in the environment until they are called upon. Dynamic clocking can ensure that an MCU sips as little current as possible until an interrupt wakes the MCU from this low-power mode to respond to whichever event. Consider a TV remote that sits at home while you're at work that can respond to a button press, wake its controller, and begin emitting the necessary sequence of electromagnetic pulses from its IR transmitter. The ability to adjust the system clock to reduce power consumption is a powerful technical feature — pun intended.

Memory & Storage

An important consideration to make when approaching MCU memory and storage for the first time is that the scales you may be used to working with on computers will likely not apply. Microcontrollers call on flash as a non-volatile option for program memory so that the very same program and initialization will occur at each reset. Typically, MCUs feature about 1 MB of program memory. Microcontrollers call on random access memory (RAM) to store working data, but this is of course volatile, so data from one power cycle cannot persist to the next. In the context of this course, it is interesting to consider the size of various models. One of the primary constraints for TinyML is ensuring that MCUs, as resource-constrained hardware, can fit a desired model in the first place. Compact speech recognition models, for instance, require 20-30 kB, whereas more complex models like those required for visual feature detection, require hundreds of kilobytes, really at the extreme of what most MCUs provide.

We want to highlight here that some microcontroller boards interface with SD cards as a way to extend the storage capacity of their system as well as to create a non-volatile record of data, for applications that require it. For what it's worth, EEPROM is another form of on-board or on-chip memory that some embedded applications (smart cards, for example) leverage to store programmable, persistent data.

In the end, you should consider the scale of your application and the role that volatility might play in limiting or enabling the function you require.

***