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 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 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 a MCU sip 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 in-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.