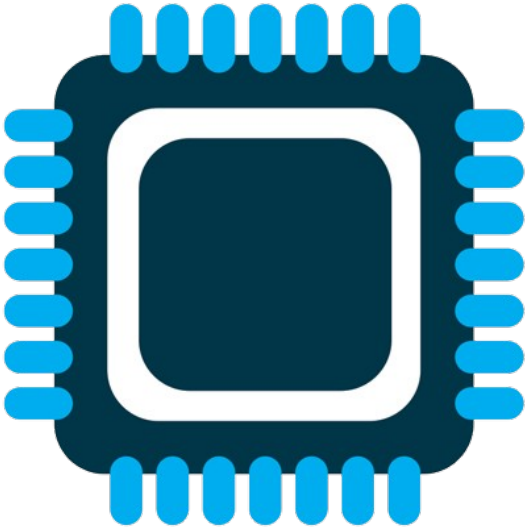
Technology Corner

Microcontroller versus System-on-Chip in Embedded System Designs



Rudolf J Streif
CTO, ibeeto
September 2019

Abstract

Microcontrollers (µC) have been dominating embedded system designs for decades. With 8, 16, and 32 bit variants and many different on-chip peripherals and memory configurations embedded system designers have many options to choose from. Semiconductor vendors often provide development tools, software libraries and code examples for common problems free of charge and free of licensing royalties, making it easy and straight forward to get an embedded project off the ground quickly.

However within the last couple of years, Systems-on-Chip (SoC) started “infringing” on the µC turf for embedded designs. SoC commonly provide more processing power, multiple CPU cores, support for high-level operating systems, access to mass storage devices and more. Affordable embedded development boards such as Beaglebone, Beagleboard, Minnowboard, Raspberry Pi and many others are sparking embedded engineers’, hardware and software developers alike, imaginations. Why not use an SoC for our next design?

Whether you should use a µC or SoC for your next design for the most part depends on your application. However, there are some additional considerations you should take into account before making your decision.

Hardware Design

If it is a µC or a SoC you ultimately have to integrate it into your hardware design. Electrical and mechanical characteristics of a µC are much different from those of SoC. Let’s talk about some of the more demanding aspects.

Clock Speed

The typical µC clock speed is in the low to high tens of MHz and therefore orders of magnitude lower than the GHz clocks that are common for SoC. Of course, higher clock speeds mean faster execution but they also put higher much more demand on the electrical design of the board to provide the necessary clock stability and also avoid radio frequency emissions from the oscillator possibly causing interference.

Processing Power

With higher clock speeds come shorter cycle time and therefore more processing power. True parallel processing can only be enabled with multiple processing cores. While there are several multi-core microcontrollers, single-core designs are dominant. For SoCs it is pretty much the other way around, multi-core (or at least multi-threading) SoCs are the norm while single-core is the exception. Even the SoC used on the $35 Raspberry Pi 3B has four cores.

An alternative to more processing power on an embedded device is cloud processing of data. The embedded device only performs data collection and eventually data pre-processing tasks and then sends the data to a cloud server for more compute-intensive processing. The server then sends back instructions to the embedded device based on the processing result. That can save on processing power and eventually power consumption of the embedded device, but of course, at the expense that the device has to be permanently connected. A common example are popular voice assistants that send audio streams to cloud servers for language processing and receive back the command to be executed.

Power Consumption

When it comes to power consumption less is always more: less processing power, less peripherals, less RAM, etc. SoCs provide a lot of functionality. For instance, if your embedded device does not have a screen then an on-chip GPU does not add any value but might consume power if you cannot turn it off. Microcontrollers are in general more power efficient than SoCs. If power consumption is of importance for your embedded application a microcontroller might be the better choice given that it can perform all of the tasks.

RAM

A standard µC has RAM and flash memory on-chip while SoCs require external RAM. On-chip microcontroller RAM is typically is static RAM and limited to a couple of megabytes which sometimes can be increased by connecting more static RAM externally. Since SoCs are derived from general purpose CPUs they use standard dynamic RAM chips which can provide many gigabytes of capacity. While dynamic RAM is cheaper than static RAM, it requires a memory controller for addressing and refreshing the RAM cells. Generally, the RAM controller is part of the SoC. Since RAM uses parallel address and data lines, it is critical that hardware designers pay careful attention to the board layout when connecting external RAM. That is in particular true for RAM with high data rates. Connecting traces on the PCB have to be at similar lengths to avoid timing problems and free from interference. Memory controllers may require tuning tables to adjust for signal timing on different traces.

Peripheral Buses

Peripheral buses on µC include low-speed serial buses such as I2C and SPI. Microcontrollers designed for industrial and automotive applications often also include CAN (Controller Area Network) and LIN (Local Interconnect Network). Some microcontrollers may also have USB interfaces, albeit that is less common. SoCs also provide the low-speed serial interfaces and very much always have USB ports. For high-speed peripheral devices they offer either PCIe or increasingly SerDes (Serializer-Deserializer) interfaces with multiple configurable lanes to which disk controllers (SATA), Ethernet bridges and other devices requiring higher data rates

can be connected. Higher data rates once again mean higher demands on the board design, albeit serial interfaces are less critical than parallel ones.

Power Management

Many SoC require different voltages for the various components such as CPU cores, memory controllers and peripherals to operate correctly. It is not uncommon that a chip needs to fed 1.8V, 3.3V and 5V at different power levels that sometimes need to be applied using specific power-up and power-down sequences for the SoC to operate correctly. The more complex SoC have companion chips, so called Power Management IC (PMIC), which simplify providing the correct power configuration.

Packaging

Almost any microcontroller is available in flat-pack packaging with connecting leads on the side of the device package. The number of connections ranges from the low tens to up to 300+ (302 pin QFP (Quad Flat Package). SoCs on the other side often have much higher pin count and require BGA (Ball Grid Array) and other high-density packaging. Such packaging has much higher demand on board layout and design often requiring six or more layers with vertical interconnects (VIA).

Hardware design for SoC can often be much more demanding and therefore time consuming than for µC. However, using a System-on-Module (SoM) may provide the advantages of both worlds: SoC performance with µC hardware design simplicity. SoM are complete compute modules with SoC, power management, RAM, peripheral devices such as Ethernet, SATA, etc. They are designed to be integrated with a motherboard through a standardized connector such as COM Express, Qseven, SMARC, etc. Using an SoM lowers hardware design complexity with the added benefit of interchangeability. Simply by shipping a different SoM with your product you can provide different functionality and performance levels without having to change your motherboard design.

Temptation might be great to use a development board such as a Raspberry Pi for production designs. After all they are ready to go with common peripherals and software support at a low cost. We strongly advise against it. These boards are not intended for production use. And as development boards they also do not have (nor need) proper certification such as FCC, CE etc. Sometimes designers of development boards also implement watchdog timers forcing a reset after a period of continuous operating time to discourage production use.

Software Design

Much can be said about the differences in software design for microcontroller and SoC. We are focusing on a few major aspects.

Architecture and Instruction Set

Nowadays, unless you are a chip designer or assembly programmer, you do not have to worry too much about architecture and instruction set anymore as software development tools have come a long way essentially abstracting machine details. However, it is helpful to understand the basic concepts as they apply to microcontrollers and SoCs and at least to some extend determine parts of the overall software design.

Microcontrollers commonly employ what is known as *Harvard* architecture. Harvard architecture separates instruction and data flows and uses separate memory for code and data. In most cases microcontroller code is stored in non-volatile memory that allows in-place execution such as ROM, EPROM or NOR flash. The CPU directly fetches the instructions from the non-volatile memory and executes them.

On the other hand, SoC typically use *von Neumann* architecture. This architecture is based on the stored-program computer concept where code and data essentially share the same memory. Programs are loaded from storage media into RAM before their code can be executed. A memory management unit (MMU) separates code from data and protects code and data regions in RAM form access by other programs. While von Neumann architecture is more flexible it also adds complexity for loading and MMU setup.

As far as instructions sets are concerned, microcontroller vendors typically implement their own for their products and provide development tools such as compilers and debuggers which are often also embedded into an Integrated Development Environment (IDE). An example is Atmel Studio for the Atmel/Microchip AVR and SAM devices which are used by the popular Arduino boards.

SoC instruction sets follow the mainstream architectures for general purpose CPUs: ARM, ARM64, PowerPC, x86, and x86-64. Virtually all SoC vendors provide board support packages (BSP) for Linux and often also for Microsoft Windows and QNX. For embedded Linux development the Yocto Project (www.yoctoproject.org) and OpenEmbedded (www.openembedded.org) have established themselves as a de-facto standard, superseding Buildroot and Crosstools-NG. Additionally, Debian Linux package feeds are gaining interest among embedded Linux developers for building Linux operating system stacks.

Operating Systems

In short, for the typical microcontroller application you probably do not need an operating system. Your program can run right away after some basic initialization of the hardware. Due to the linear architecture and a lack of a memory management unit, concurrency is pretty much limited to interrupt service routines next to the main loop or your program. Software libraries, provided by the µC vendor, that you can easily integrate into your own programs simplify dealing with peripherals.

To use the full potential of a SoC you most likely will want to use an operating system. That will also require a boot loader to set up the core hardware functionality necessary for the operating system kernel to execute. Modern SoC commonly use a multi-stage boot process starting with a first stage running from internal ROM or flash memory. Subsequent stages can then be loaded from various boot media, often also including USB devices next to disk drives and others. Once the operating system kernel is loaded and executing, it takes over control of the hardware. The two main jobs of the operating system are to manage the system resources and to provide an interface for applications to utilize the system resources (Application Binary Interface (ABI)).

Since you probably won't be developing an operating system yourself, the amount of third-party code used in your system, and thus system complexity, will increase by orders of magnitude when using a SoC over a microcontroller. Given, you will get much more functionality with an SoC but that is only of value if you are actually going to use it for your application.

Startup/Boot Time

This is where the microcontroller truly outshines the SoC. Embedded systems with SoCs running high-level operating systems can easily take 30 seconds or more to boot. Just look at your smartphone. Since the microcontroller boot process is much simpler, startup times of less than 10 ms can be achieved. In all fairness of course, systems with SoC often have a plethora or peripheral devices that need to be initialized during the boot process which can be time consuming.

Optimizing the boot process of a device with an SoC is possible but takes a lot of experience and time. You clearly need to understand hardware and software dependencies to squeeze the last bit out of the system startup time. High-level operating systems such as Linux are commonly built to support different hardware configurations with the same kernel. That potentially means hardware drivers being loaded and hardware being probed that does not exist. Removing those driver is a first step.

System Integration or Own Code versus Third-party Code

We think that the Pareto rule pretty much applies to own code versus third-party code: 80/20 own code versus third-party code for a microcontroller project and 20/80 for a SoC project. This is where organizations accustomed to designing microcontroller-based systems struggle the most when switching to SoC: software development all of a sudden becomes more of an integration task than a coding task. Furthermore, these organizations are used to be in full control over all of the source code that eventually comprises their device's software stack. Hence the majority of them look to Linux and open source as the solution. Unfortunately, it is a common oversight that the *free* in free software refers to *freedom of use* not to *free of cost*. Yes, you do not have to buy the software but you have to make substantial investments into your organization's software engineering capabilities to fully take advantage of open source software.

Considering the size of your team and the amount of code produced during a regular work day, you might still be getting away with the old paradigm of nightly builds and weekly test cycles. That will definitely not be sufficient anymore when using Linux and open source software but even if you are using a commercial OS or a commercial distribution of Linux you will eventually have to switch to continuous integration and test or ultimately Development Operations (DevOps). Think of it as if your software development team all of a sudden has grown to thousands of developers distributed around the globe. Each day there are tens of thousands of new lines of code written, new functionality integrated, bugs fixed, Common Vulnerabilities and Exposures (CVE) resolved and more.

The common head-in-the-sand approach is to freeze the third-party software to a known and working state and only integrate your own software on top of it. However, that only gets you that far and eventually you will have to upgrade the third-party software for your device to newer versions. If you waited too long that easily becomes a major upheaval.

Consequently the demand for *long term support* is only too understandable. For the Linux kernel there is the Long Term Support Initiative (LTSI) managed by the Linux Foundation (https://ltsi.linuxfoundation.org). But the Linux kernel is only one of the about 1,000 software packages that make up a typical Linux operating system stack. What about those?

Even if there were long term support for all of those software packages it would only postpone the problem to a future point in time. Yes, ideally you would want long term support for the expected life of your product. But how long that actually is depends on the product. For a consumer electronics product that might be two or three years; for an industrial control it might be 7 to 10 years; for a car 10 to 15 years.

Continuous integration and DevOps cannot entirely solve the problem either. In particular it cannot solve the obsolescence problem at which point your hardware in the field simply will

not be able to run the latest software anymore. However, it can move the mark at least for some time. If you continuously integrate software updates during the development of your product and test it then once you start shipping your product, you are shipping it with the latest software and not with software that has already been outdated during the time of development. If you continue the practice after the product started shipping you can provide current software updates to your existing customers while preparing for the next release of your product. Many manufactures of mobile phones are already practicing this process: until hardware obsolescence is preventing further updates they are delivering new software versions to their customers. A side effect of this service is that it does not go unnoticed with customers and thus supports brand loyalty.

Security

Neither microcontroller nor SoC are inherently secure or not secure. It is the software running on them that exposes vulnerabilities that can be exploited for attacks. It is standard for current devices to have Internet connectivity for data collection and remote access. Connectivity adds another dimension to vulnerability as attacks can easily be scaled across thousands if not millions of devices.

Nowadays, embedded developers have to design their devices with security in mind. Even if your device does not have network connectivity chances are that somebody might use a serial-to-Ethernet converter to attach your device to a network through the serial port that you only intended for system maintenance.

The more software you have running on a device the greater the attack surface might become. Embedded developers love remote terminals, network file systems, network boot capabilities etc. as they can greatly accelerate round-trip development. Operating systems such as Linux might automatically run terminal services on a serial port leaving an unnoticed backdoor. Network services are often started listening for incoming connections on common network ports. If they are not needed it is best to remove them or at least disable them. If they are needed follow best practices to secure them.

However, it does not stop there. Eventually your device needs to be provisioned and tested as part of the production process. Your contract manufacturer might tell you that they need certain capabilities and remote access built into the system software to run their tests. Ideally, such functionality should be removed once the production tests have been completed.

Of course much more can be said about security and hardening your device. If you use a SoC with an operating system make sure that you fully understand the entire stack and what the software components do to assess if you actually need them. Then tailor your system accordingly. Less is more when it comes to security.

Development Environments

With a high-level operating system typically comes the support for many different development environments and programming languages. You often have the choice of multiple different Integrated Development Environments (IDE) such as Eclipse (https://www.eclipse.org) and of course virtually any programming language: C/C++, Go, Java, Python, Rust and many more. Choices are much more limited for microcontrollers. C is pretty much the de-facto standard but it is also customary that microcontroller provide IDEs with built-in compiler, linker, assembler, debugger, target deployment, etc.

For team leads and project managers finding developers with the right skill set is of utmost importance. There is a good reason for semiconductor companies providing and supporting the development of inexpensive development boards together with tools and tutorials. Embedded system development used to require expensive development tools such as in-circuit emulators, EPROM programmers and erasers, custom compilers and more. And to use them properly training classes often were necessary too. Readily available hardware at low cost and tools at no cost with online documentation and tutorials have changed the landscape enlarging the pool of developers with the right skills.

There are plenty of options to choose from when deciding on a CPU for an embedded project. Narrowing down the choices through diligent analysis of the required functionality is mandatory. The final decision might always be a compromise but knowing your options and understanding the trade-offs upfront avoids delivery delays, redesign efforts, waived functionality and more down the road of product development.

Contact Information

We would like to hear from you! If you have feedback to this ibeeto Technology Corner article please contact us:

ibeeto
Rudolf J Streif
rudolf.streif@ibeeto.com
+1.855.442.3386

ibeeto is a hardware and software consulting firm providing development and engineering services as well as a variety of training courses. We are happy to assist you with getting your next embedded project off the ground.