Microcontrollers & SBCs Cheatsheet

A quick reference guide to microcontrollers and single-board computers (SBCs), covering essential terminology, architectures, popular platforms, and key considerations for selection and use.

Fundamentals

Definitions

Microcontroller (MCU)	A self-contained system-on-a-chip that includes a processor core, memory, and programmable input/output peripherals. Designed for embedded applications.
Single-Board Computer (SBC)	A complete computer built on a single circuit board, typically including a microprocessor, memory, I/O, and other features required for a functional computer. Often runs a full operating system.
Embedded System	A specialized computer system designed to perform a dedicated function, often with real-time constraints. Microcontrollers are commonly used in embedded systems.
SoC	System on Chip, integrates all components of a computer or other electronic system into a single integrated circuit.
GPIO	General Purpose Input/Output pins. Configurable pins on a microcontroller or SBC that can be used for digital input or output.
UART	Universal Asynchronous Receiver/Transmitter. A serial communication protocol.

Key Differences

Processing Power	SBCs generally have significantly more processing power than microcontrollers, featuring faster processors and more memory.
Operating System	SBCs typically run a full operating system (e.g., Linux, Windows IoT), while microcontrollers often use real-time operating systems (RTOS) or run bare-metal code.
Complexity	SBCs are more complex to set up and manage due to the OS and software dependencies. Microcontrollers are simpler for basic tasks.
Power Consumption	Microcontrollers generally consume less power than SBCs, making them suitable for battery-powered applications.
Cost	Microcontrollers are usually cheaper than SBCs, especially for high-volume production.
Use Cases	Microcontrollers are suited for dedicated tasks like controlling sensors and actuators. SBCs are better for applications requiring complex processing, networking, or user interfaces.

Architectures

RISC (Reduced Instruction Set Computing)	Emphasizes simplified instruction sets, leading to faster execution and lower power consumption. ARM architecture is a prominent example.
CISC (Complex Instruction Set Computing)	Features a more extensive set of instructions, allowing for more complex operations. x86 architecture is a common example (used in many SBCs).
ARM	A widely used RISC architecture, particularly in microcontrollers and mobile devices. Known for its energy efficiency and versatility.
x86	A CISC architecture commonly found in desktop and laptop computers. Also used in some higher-end SBCs.
Harvard Architecture	Separate memory spaces for instructions and data, enabling simultaneous access and faster execution.
Von Neumann Architecture	Single memory space for both instructions and data, simpler but may lead to performance bottlenecks.

Popular Platforms

Microcontroller Platforms

Arduino	An open-source electronics platform based on easy-to-use hardware and software. Ideal for beginners and rapid prototyping. Uses AVR microcontrollers.
ESP32	A low-cost, low-power system-on-a-chip (SoC) series with Wi-Fi and Bluetooth capabilities. Popular for IoT applications.
STM32	A family of 32-bit microcontrollers based on the ARM Cortex-M core. Known for their performance and versatility.
PIC Microcontrollers	A family of microcontrollers from Microchip Technology, widely used in embedded systems due to their low cost and ease of programming.
Teensy	A line of microcontroller boards designed for hobbyists and developers, offering a balance of performance, size, and ease of use.
AVR	A family of microcontrollers developed by Atmel (now Microchip Technology), commonly used in Arduino boards and other embedded applications.

Single-Board Computer Platforms

Raspberry Pi	A series of small, affordable SBCs widely used for education, hobbyist projects, and industrial applications. Runs Linux-based operating systems.
NVIDIA Jetson	A family of SBCs designed for AI and machine learning applications, featuring powerful GPUs and optimized software libraries.
BeagleBone	A series of open-source SBCs known for their flexibility and extensive I/O capabilities. Often used in industrial automation and robotics.
ODROID	A line of SBCs offering a range of performance options and features, suitable for various applications including gaming, media centers, and embedded systems.
Intel NUC	A small form factor computer that can be used as a single board computer alternative with more processing power. Usually runs Windows or Linux.
Rock Pi	A high-performance single board computer offering excellent performance and rich interfaces.

Comparison Table

Feature	Arduino Uno	Raspberry Pi 4	ESP32
Processor	AVR	ARM Cortex-A72	Dual-core ESP32
Clock Speed	16 MHz	1.5 GHz	240 MHz
Memory	2 KB SRAM	1-8 GB RAM	520 KB SRAM
Operating System	None	Linux	RTOS/None
Use Case	Simple tasks	Complex apps	IoT
Cost	Low	Medium	Low

Selection Criteria

Performance Requirements

Consider the processing power, memory, and clock speed required for your application. SBCs are preferable for computationally intensive tasks, while microcontrollers are sufficient for simpler control applications.

Evaluate the need for real-time processing. Microcontrollers often excel in real-time applications due to their deterministic behavior.

Assess the complexity of algorithms and data processing involved. SBCs are better suited for complex algorithms and large datasets.

I/O and Connectivity

Determine the number and type of I/O interfaces required (e.g., GPIO, UART, SPI, I2C, USB). Microcontrollers offer a wide range of I/O options, while SBCs provide more connectivity options like Ethernet and HDMI.

Consider the need for wireless connectivity (e.g., Wi-Fi, Bluetooth, Cellular). Some microcontrollers and SBCs come with integrated wireless modules.

Evaluate the need for analog input and output capabilities (ADC/DAC). Microcontrollers are commonly used for analog sensor interfacing.

Power Consumption

For battery-powered applications, prioritize low power consumption. Microcontrollers generally consume less power than SBCs.

Consider power management features such as sleep modes and voltage scaling. These features can help minimize power consumption when the device is idle.

Evaluate the power requirements of peripherals and external components. Choose components that are energy-efficient.

Software and Development Environment

Consider the availability of software libraries, development tools, and community support. Arduino and Raspberry Pi have large communities and extensive resources.

Evaluate the ease of programming and debugging. Some platforms offer user-friendly IDEs and debugging tools.

Consider the operating system requirements. SBCs typically run Linux, while microcontrollers often use RTOS or bare-metal programming.

Programming Languages

Languages for Microcontrollers

C/C++	The most common languages for microcontroller programming. They provide low-level control and efficient memory usage.
Assembly Language	Provides the most direct control over the hardware, but is more complex and time-consuming to write. Used for performance-critical sections of code.
MicroPython	A lean and efficient implementation of the Python 3 programming language that is optimized to run on microcontrollers.
Arduino Programming Language	A simplified dialect of C++ designed for use with the Arduino IDE. Makes microcontroller programming more accessible.

Languages for SBCs

Python	A high-level, general-purpose programming language that is widely used on SBCs due to its readability and extensive libraries.
C/C++	Also used on SBCs, especially for performance-critical applications and system-level programming.
Java	A platform-independent language commonly used for developing applications on SBCs, particularly in enterprise environments.
JavaScript	Used for web development and Node.js applications on SBCs. Useful for creating user interfaces and network services.

Debugging Tips

- **Use a debugger:** Tools like GDB or platform-specific debuggers can help step through code and inspect variables.
- **Print statements:** Insert print statements (e.g., `printf` in C/C++, `print` in Python) to trace program execution and check variable values.
- **Logic analyzers:** Use a logic analyzer to monitor digital signals and identify timing issues.
- **Oscilloscopes:** Use an oscilloscope to visualize analog signals and diagnose signal integrity problems.
- **Serial communication:** Use serial communication to output debugging information from the microcontroller or SBC.
- **LEDs:** Use LEDs as visual indicators to signal specific events or states in the program.