Embedded Systems

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

The course delves into embedded systems, covering their characteristics, requirements, and constraints. It explores hardware architectures, including various types of software executors, communication methods, interfacing techniques, off-the-shelf components, and architectures suited for both prototyping and large-scale production.

In terms of software architectures, the course examines abstraction levels, real-time operating systems, complex networked systems, and the tools and methodologies used for code analysis, profiling, and optimization.

Students will also learn to analyze and optimize hardware/software architectures for embedded systems, focusing on managing design constraints and selecting appropriate architectures. Key topics include estimating and optimizing performance and power at various abstraction levels, project management, and designing for reuse.

Additionally, the course addresses run-time resource management and includes case studies to illustrate trade-offs based on application fields and system sizes.

Contents

CHAPTER 1

Introduction

1.1 Introduction

Embedded systems are characterized by their ubiquitous presence, low power consumption, high performance, and interconnected nature. These systems are commonly utilized in four main application contexts:

- *Public infrastructures*: safety is a primary concern to prevent potential attacks, and in some cases, latency is also crucial. Examples include highways, bridges, and airports.
- *Industrial systems*: reliability and safety are the predominant concerns. Key industries utilizing embedded systems include automotive, aerospace, and medical.
- *Private spaces*: this includes control systems for houses and offices.
- *Nomadic system*: these systems involve data collection related to the health and positions of animals and people, requiring both security and low latency.

In the future, embedded systems will evolve in several key ways:

- *Networked*: transitioning from isolated operations to interconnected, distributed solutions.
- *Secure*: addressing significant security challenges that impact both technical and economic viability.
- *Complex*: enhanced by advancements in nanotechnology and communication technologies.
- *Low power*: utilizing energy scavenging methods.
- *Thermal and power control*: implementing runtime resource management.

Usually, we process useful data locally and send the relevant data to the cloud less frequently. This approach conserves energy, thereby extending battery life. Due to time constraints, developing a device from scratch is often infeasiblebecause the design process involves a multidisciplinary team.

1.1.1 Technological problems

Applications are expanding rapidly, pushing the need for mass-market compatibility and integrating into all aspects of life, which in turn drives up volumes. However, finding a balance in this progress is complicated by several factors. CMOS technology is reaching its physical limits, and the cost of developing new foundational technologies can often be unaffordable. Additionally, the power and energy demands create significant barriers, as does the exponential proliferation of data.

1.2 Modern applications

Key characteristics of modern applications include:

- *Compute-intensiveness*: they will demand significant computational resources, necessitating optimized hardware and software regardless of their application domain.
- *Connectivity*: these applications will be interconnected, either wired or wirelessly, and will often be online—globally interconnected through the Internet.
- *Physical entanglement*: they will be embedded within and capable of interacting with the physical world, not only observing but also controlling their environment.
- *Intelligence*: these applications will possess the capability to interpret noisy, incomplete, analog, or remote data from the physical world, allowing for smarter interactions and decision-making.

The ongoing integration of the digital and physical worlds will be driven by advances in cognitive computing, big data analytics, and data mining.

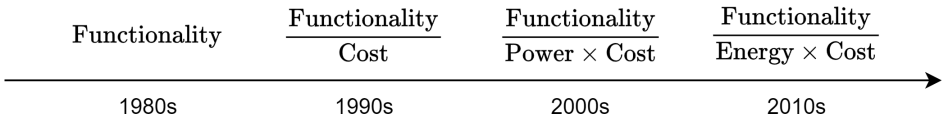

Figure 1.1: Industry requirements

While the number of transistors in modern systems can continue to increase, a key challenge is that we cannot power all of them simultaneously. This limitation drives the need for innovative approaches to using extra transistors more efficiently. Additionally, heterogeneous processing combined with aggressive power management is crucial for optimizing performance across various tasks.

Transistors As transistors shrink in size, several challenges emerge. Process variation, physical failures, and aging mechanisms can degrade device performance over time. With very-large-scale integration, packing more transistors into smaller spaces increases power density, which in turn creates thermal issues. Furthermore, traditional communication subsystems struggle to provide adequate power-performance trade-offs in these densely packed systems.

Silicon Despite silicon being a relatively good thermal conductor, large chips can still experience significant temperature gradients. One practical solution involves dividing the chip into concentric rings and applying dynamic voltage and frequency scaling to each region. By fine-tuning voltage and frequency for each ring, it becomes possible to optimize performance while maintaining thermal balance.

Frequency Modern systems address power management by partitioning chips into independent islands that operate at different voltage and frequency levels. This approach allows sections of the chip to be dynamically turned off when not in use, a process known as power gating.

1.2.1 Cooling

Air cooling, while common, suffers from a low heat transfer coefficient, poor chip temperature uniformity, and requires large heat sinks and air ducts. It is also noisy and expensive to maintain. Water cooling offers an improvement, more uniform chip temperatures, smaller heat sinks, and fewer fans. Water cooling also allows for potential heat recovery, though it requires large pumps to function effectively.

Two-phase cooling systems present an even more efficient solution. They provide better chip temperature uniformity, and smaller pumps, along with isothermal coolant. However, two-phase cooling systems suffer from low pump efficiency and reliability issues.

1.3 Productivity and planning

The revenue model can be visualized simply as a triangle, where the product's life is represented by a span of $2W$, peaking at W. Any delay in market entry results in a loss, which is the difference between the areas of the on-time and delayed triangles.

In theory, increasing the number of designers on a team should reduce project completion time. In practice, though, productivity per designer tends to decrease due to the complexities of team management and communication. At a certain point, adding more designers can even extend project timelines.

To enhance productivity, the design methodology must support reuse, particularly at higher abstraction levels, and this should be backed by standardization (International Technology Roadmap for Semiconductors). Two nain technology development trends are:

- *System on Chip* (SoC): focuses on full integration and achieving the lowest cost per transistor.
- *System in Package* (SiP): focuses on lowering the cost per function for the entire system.

1.3.1 Micro Electro Mechanical systems

Micro-Electro-Mechanical Systems (MEMS) are structures with integrated circuit and specialized micromachining (transducers, microsensors, and microactuators). Integrated microsystems combine circuitry and transducers to perform tasks autonomously or with the assistance of a host computer.

Wireless sensor nodes Wireless sensor nodes are small, battery-powered devices that monitor local conditions. These devices typically have limited resources and form nodes within a wireless network that covers a region or object of interest.

CHAPTER 2

Microprocessors and microcontrollers

2.1 Microprocessors

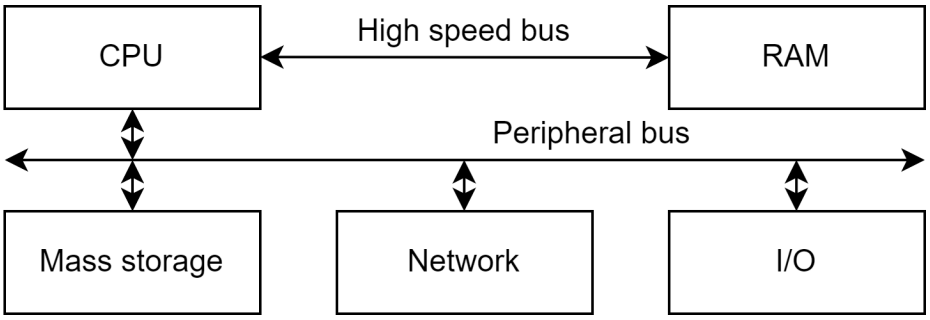

Figure 2.1: Microprocessor architecture

The key features of microprocessor-based applications include:

- *Flexibility*: this encompasses maintainability and the ability to evolve the application.
- *Time To Market*: this refers to the duration required to bring a product to the customer.
- *Easy upgrade*: software solutions can be updated and improved with minimal disruption.
- *Cost*: the overall cost is volume-dependent. Design costs are incurred only once, while production expenses are primarily linked to the cost of silicon.

This approach is cost-effective since embedded systems often do not require the full processing power of a microprocessor.

While an equivalent hardware solution may generally provide better performance, microprocessor architectures are optimized for flexibility. While software implementations are typically easier to develop than hardware solutions, modifying hardware to meet new requirements is often simpler, with performance advantages remaining on the hardware side.

Designing a software solution necessitates a comprehensive understanding of the microprocessor architectures available in the market. The characteristics of the problem (algorithm) should guide the designer, alongside non-functional constraints such as performance, cost, development time, and power consumption. The initial step before selecting a specific processor is determining the appropriate class of processor and the form in which it will be acquired.

2.1.1 Taxonomy

Processors can be categorized into two main types:

- *Application Specific Processors* (ASP): these processors are tailored for specific classes of applications that require high performance.
- *General Purpose Processors* (GPP): these processors are not optimized for any particular application.

In typical embedded systems, a multi-processor solution often employs GPPs for supervising and controlling the activities of one or more ASPs.

Availability Processors can further be classified as follows:

- *Components Off The Shelf* (COTS): these are standard chips purchased and mounted onto a printed circuit board along with the necessary interfaces to integrate with the rest of the system.
- *Intellectual Property* (IP): this involves purchasing the design specifications of a microprocessor. In this case we may classify the microprocessors depending on the description as: soft macro (HDL) or hard macro (layout).

Selection The selection process is based on:

- *Class*: the nature of the algorithm and the operations are the primary drivers.
- *Form*: the target architecture of the system.
- *Performance*: a key metric for performance is the average number of Instructions Per Clock (IPC). Million Instructions Per Second (MIPS) serves as an absolute measure of throughput but can be misleading when comparing processors with different Instruction Sets. For floating-point operations, MFLOPS is commonly used, while specialized architectures like Digital Signal Processors (DSPs) often use Million Multiply Accumulate operations Per Second (MMACS), and Network Processor (NP) typically measure the average number of processed packets per time unit.
- *Power*: both average power and peak power are important metrics for estimating the maximum or average power consumption of the overall system. A combined measure of power and speed is often used.

Other important considerations include:

- *Memory*: the bandwidth and size requirements of the application can impose hard constraints. In some cases, only internal memory may be available, making external memory utilization impractical.
- *Peripherals*: embedded systems often process external signals and control physical devices. Designers can either choose an architecture focused on computation and design the rest of the system accordingly or opt for a single-chip solution that integrates computing, interfacing, and peripherals.

- *Software*: he availability of libraries can simplify both the design and validation processes. Differences among SDKs are significant, and factors such as the software compilation flow, code quality, and flexibility offered to the designer are important.
- *Packaging*: packages differs in size, pinout, and materials.
- *Certifications*: some certifications are tailored for specific fields, such as MISRA rules for automotive code.

2.1.2 General Pupropose Processor

General Purpose Processors (GPPs) are characterized by a generic architecture that is applicable across a wide range of fields. They are commonly found in PCs, and for embedded systems, they are suitable for low-end applications where energy efficiency and performance are not critical, and the application nature is quite heterogeneous. GPPs handle tasks such as controlling and managing slow interfaces with sensors and enabling interactivity through graphical user interfaces. The possible architectural styles for GPPs are:

- *Complex Instruction Set Computer* (CISC): allow each arithmetic or logic operation to access data and write results using any available addressing mode. However, in practice, the number of combinations of operations and addressing modes is often limited. Despite the complexity, CISC instructions are encoded in variable-length formats, which necessitates more complex fetch and decode units and a higher speed for memory access. Some architectures even support vector instructions, enabling operations on multiple registers simultaneously. To enhance throughput, CISC processors often employ intricate pipelining techniques. Additionally, they may alter the order of execution for assembly instructions without changing the program's semantics (out-of-order execution). This complexity makes it difficult for compilers to predict instruction execution status, potentially leading to suboptimal instruction decoding.
- *Reduced Instruction Set Computer* (RISC): utilize a limited number of straightforward instructions. RISC instructions typically have fixed lengths, facilitating balanced pipeline stages and higher clock speeds. RISC employs a load and store architecture for input and output operations. While the limited number of addressing modes can aid performance, effective use of memory hierarchy is crucial for minimizing data access time. When comparing RISC and CISC using the same high-level source code, RISC generally requires a higher number of instructions. RISC architectures do not have hardware mechanisms to resolve pipeline conflicts, but their predictable execution allows compilers to generate efficient code. Furthermore, RISC architectures typically exhibit lower power consumption due to their inherent simplicity compared to CISC architectures.
- *Superscalar*: multiple execution units, enabling the simultaneous execution of more than one instruction per clock cycle. This parallelism enhances throughput but also increases the complexity of control logic. However, a significant portion of the processor's resources is dedicated to implementing complex scheduling mechanisms and maintaining consistency.
- *Very Long Instruction Word* (VLIW): these processors support explicit parallelism of RISC instructions. Once fetched from memory, the VLIW word is decoded in parallel, with individual instructions dispatched to various execution units. This fixed structure

shifts the complexity of scheduling onto the compiler. Power consumption for VLIW architectures falls between that of simpler RISC and superscalar designs.

- *CISC RISC*: combine out-of-order execution with superscalarity while maintaining compatibility with legacy code. This architecture is more costly, but achieves higher performance.
- *CISC VLIW*: aims for high performance while ensuring backward compatibility. CISC instructions are broken down into basic instructions that can be executed by a VLIW core, which is both simple and efficient.

Comparison The comparison between GPPs architecture is outlined in the following table:

	RISC	**CISC**	**Superscalar**	**VLIW**
Instruction set	simple	complex	complex	simple
Addressing modes	few	many	many	few
Instruction size	fixed	variable	variable	fixed, large
Register file	single	single	single	multiple
Number of registers	high	low	medium	very high
Instruction scheduling	static	dynamic	dynamic	static
Conflict management	static	dynamic	dynamic	static
Compiler complexity	medium	high	low	very high
Parallelism	absent	absent	implicit	explicit
Hardware complexity	low	high	very high	high
Pipeline complexity	low	high	very high	medium
Power consumption	very low	high	very high	high
Typical IPC	≈ 1	< 1	> 1	≥ 1

2.1.3 Application Specific Processor

Embedded systems often require high specialization and a limited set of functions, making GPPs less suitable for certain applications. To address these specific needs, Application Specific Processors (ASPs) have been developed. The main ASPs architectures are:

- *Digital Signal Processor* (DSP): microprocessors designed specifically for numerical computing tasks based on the multiply accumulate (MAC) operation:

$$z_{t+1} = z_t + x \cdot y$$

To achieve high efficiency, DSPs are designed to optimize the execution of loops. This is possible due to certain characteristics of loop structures, such as having small bodies—typically around ten assembly instructions—where the control loop variable remains unchanged. To combat this, DSP architectures often include high-bandwidth interfaces and sophisticated memory hierarchies for enhanced memory access speed. Many modern ASPs processors are based on VLIW architectures to optimize performance.

- *Network processor* (NP): specialized microprocessors tailored for processing network packets, which use high levels of parallelism to handle the demands of network applications. Designing an NP can be quite complex due to the ever-evolving networking functionalities and protocols. To maximize performance, NPs incorporate dedicated hardware for high-speed I/O interfaces, queue management, cryptographic cores, and multiple RISC cores (for supervision and management). In some cases we may have specialized NP called Packet Processors (PPs) or Channel Processors (CPs) to manage multiple independent data channels.
- *Microcontrollers*: these devices integrate peripherals and interfaces onto a single chip, making them ideal for applications that do not demand high computational power but do require careful management of hardware resources and development time. The lack of memory interfaces can increase pin counts, prompting some architectures to employ multiplexing techniques to limit this increase. Microcontrollers typically provide a variety of peripheral interfaces. The SDKs available for microcontrollers can vary significantly, ranging from basic C compilers to comprehensive frameworks that assist in performance analysis and configuration management.

2.2 Microcontrollers

Microcontrollers are specifically designed to perform targeted computational tasks. These devices incorporate small, integrated memory solutions.

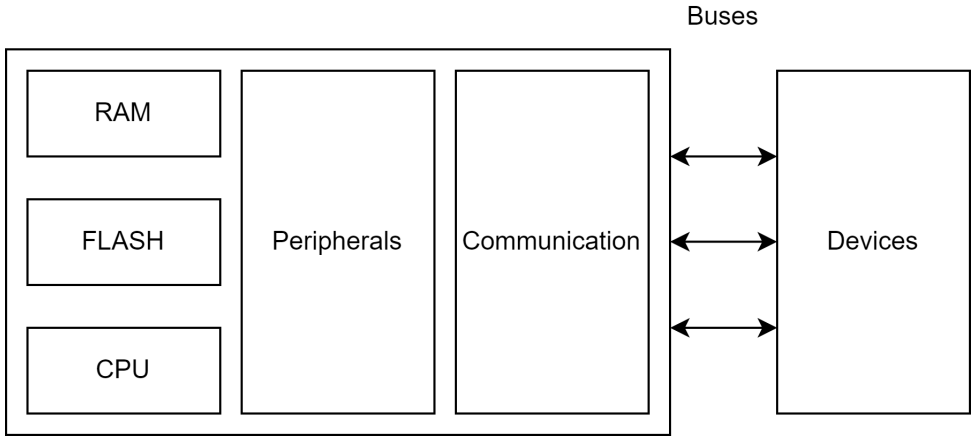

Figure 2.2: Microcontroller architecture

2.2.1 Subsystems

The main subsystems in microcontrollers are:

- *Clock*: fundamental component that synchronizes all subsystems (core, memories, and peripherals). Usually, we have two types of clock:
  - *Main clock*: drives core, memories, and peripherals (from 2 to 144 MHz).
  - *Real Time Clock* (RTC): used for timekeeping (usually $\approx$ 32 Hz). It serves two primary functions timekeeping and periodic alarm generation. It is designed for high

accuracy over extended periods. To achieve accurate timing, periodic resynchronization with external sources is necessary.

RTC can be used to count periodically (timers) or check periodically some subsystems (watchdogs). Watchdogs are configured once during the boot process, it is set to trigger an interrupt or reset at a fixed time interval. There are two primary modes of operation for watchdog timers: non-windowed (cleared before a specified time) and windowed (cleared within a specific time).

The clock could be originated by internal oscillators (imprecise due to heat, djusted through a Digital Phase Locked Loop) or external oscillators (accurate, frequency adjusted with a DPLL or a crystal). The main clock can be modulated by dedicated clock generation and distribution logic, which enables various clock feeds to supply different groups of elements (called domains). Clock management consists in mantaining low drift, low temperature, and high accuracy. To lower the power consumption techniques such as frequency scaling and clock gating are employed.

- *Memory*: usually, it is characterized by a single addressing space (with 16 or 32 bit addresses). Various types of memory are mapped to different regions within this same addressing space, eliminating the need for a cache in many cases.

  When the addreesing space is limited we may use banked memory (that composes a subset of the global memory). Thus, in this case we have differences between local and global addresses.

  In case of wide address we need some specialized registers that necessistates special memory to be handled.

- *General Purpose Input Output* (GPIO): GPIO pins are capable of serving multiple functions such as digital and analog input, and analog input output. Those pins are organized into groups known as ports. To manage these functionalities effectively, several registers are utilized:

  - *Data Direction Register* (DDR): defines the direction of each pin.
  - *Port Data Register* (PDR): holds the data for the pin when it is used in digital mode.
  - *Port Pin State Register* (PPSR): checks the current state of the port pins (used for real-time monitoring of statuses).
  - *Edge Port Control Registers* (EPCR):configure pins for various detection modes (level detection and detection of rising or falling edges). They also enable or disable interrupts related to specific pins and manage internal pull-up or pull-down resistors.
  - *Edge Port Status Register* (EPSR): maintains status flags associated with pins that have interrupt capabilities.

- *Analog comparators*: specialized differential amplifiers designed to compare two input voltages, saturating the output to either the supply voltage or ground. Given two inputs voltages A and B, and an ouptu Y, the behavior of the analog comparator is straightforward:

$$Y = \begin{cases} 1 & \text{if } A > B \\ 0 & \textit{otherwise} \end{cases}$$

  This behaviour could be inverted through polarity selection.

  Analog comparators can be used for various purposes:

  - *Polling*: based on the output value, the software can perform slow operations.
  - *Interrupt*: the comparator's output is connected to an interrupt controller. When a comparison event occurs, an interrupt service routine is executed.
  - *Hardware*: the output of the comparator is fed directly to an output pin.
  - *Hardware count or capture*: the output is connected to a control input of a timer. In count mode, every change in the output generates a front that is counted In capture mode, the system measures the time interval between two fronts.

- *Analog to digital converter* (ADC): encodes an input voltage signal into a numeric value. The steps performed by the ADC are:

  1. *Sampling*: the input signal is observed based on a periodic time base.
  2. *Quantization*: rounds the real valued physical quantity of the input signal to fixed discrete intervals determined by the power supply voltage. This process converts the continuous signal into a finite set of values.

  Accurate timing is essential for the proper functioning of an ADC (with the main clock).

  The perfromance is dependent on the stability of the power supply. To mitigate this issue, ratiometric measurement techniques are employed, which require a fixed absolute voltage reference to maintain accurate readings.

  One important function of the ADC is input multiplexing (multiple analog inputs are connected and converted cyclically).

- *Communication*: there are multiple buses available:

  - *Serial Peripheral Interface* (SPI): synchronous communication protocol used to transfer data between a single master and multiple slave devices. The communication is synchronized through a clock signal generated by the master, allowing high-speed data transfer rates. SPI uses three primary lines: MISO (Master In, Slave Out), MOSI (Master Out, Slave In), and SCK (Serial Clock). A separate line, SS (Slave Select), is used to choose which slave device the master communicates with.

    In multi-slave configurations, connecting N slave devices requires $3 + N$ lines, as each device is selected by its dedicated SS line. One of the strengths of SPI is that devices can operate at different speeds, making it adaptable for various systems.

    During transmission, the master asserts the SS line to select the target slave, generates clock pulses, and sends data to the slave on the MOSI line. In reception, the master selects the slave, sends clock pulses, and transmits dummy data on the MOSI line while receiving actual data from the slave on the MISO line.

    In full-duplex data transfers, SPI allows the master to send data to the slave on MOSI while simultaneously receiving data from the slave on MISO. To account for delays between read and write operations, dummy bytes can be added at the beginning or end of the data frames.

  - *Universal Asynchronous Receiver Transmitter* (UART): point-to-point asynchronous communication protocol with medium data transfer rates that involves a transmitter (Tx) and a receiver (Rx). Hardware flow control can be implemented in UART to allow devices to negotiate when to start or stop the transmission of data. The Ready To Send (RTS) signal indicates that the transmitter is ready to send data, while the

Clear To Send (CTS) signal is used by the receiver to inform the transmitter when it is ready to receive data.

In UART, data transmission is organized into frames. A typical frame consists of a start bit, followed by 7 or 8 data bits, an optional parity bit (used for error detection), and one or two stop bits.

In operation, the transmitter sends bits over the Tx line, while the receiver reads them from the Rx line. If hardware flow control is used, the transmitter first asserts the RTS signal, indicating it is ready to send. When the receiver is ready, it asserts the CTS signal, allowing the transmission to proceed. The transmitter then sends bits over the Ex line, and the receiver reads them over the Rx line. If the receiver needs the transmitter to pause, it de-asserts the CTS signal, causing the transmitter to suspend sending until the receiver re-asserts CTS.

– *Inter Integrated Circuit* (I^2C): synchronous communication protocol designed to allow communication between multiple devices, including multiple masters and slaves. The protocol uses two lines: a bidirectional data line (SDA) and a clock line (SCL), both shared by all devices on the bus.

I^2C supports multiple slaves without needing additional lines for each slave, as required in SPI. It has lower data rates compared to SPI

In a typical I^2C configurations, a single master communicates with multiple slave devices. The master initiates all data transfers and generates the clock signal. Both the master and the slaves can control the data line depending on whether the operation is a read or write.

In a write operation, the master generates a start condition and sets up to write by sending the slave address with the read and write bit set to 0. The master then sends the data chunks, with the slave acknowledging each chunk, and the master finally ends the operation by generating a stop condition.

In a read operation, the master begins similarly by generating a start condition and setting up to read, sending the slave address with the read and write bit set to 1. The slave acknowledges and then sends data chunks, which the master acknowledges. Once the read is complete, the slave does not acknowledge, and the master generates a stop condition.

I^2C also supports combined transfer operations, where a write and a read occur in a single transaction. In this case, after the initial write operation, the master generates a repeated start condition and then initiates a read operation. Data is exchanged in the same way, with the master acknowledging until the transfer is complete. After receiving all the data, the master sends a stop condition to terminate the communication.

2.3 Real Time Clock

A Phase-Locked Loop (PLL) operates by synchronizing the frequency of a Voltage-Controlled Oscillator (VCO) with that of an input signal.

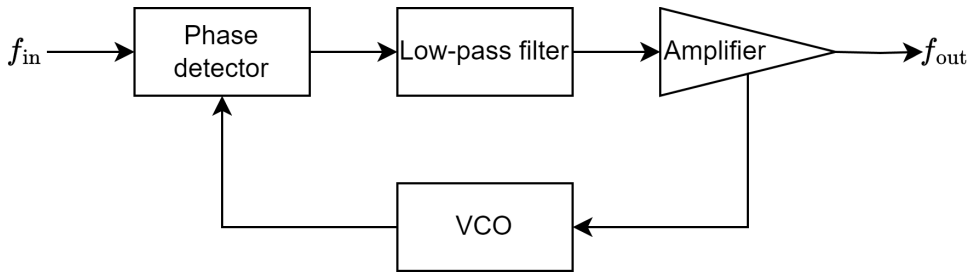

Figure 2.3: Phase-Locked Lopp

In normal operation, when the loop is locked, the input and VCO output frequencies are identical. The phase detector generates a fixed phase difference between these signals, which is converted into a DC control voltage applied to the VCO. As the input signal frequency fluctuates, the phase detector adjusts this DC voltage to maintain synchronization. We have two useful frequencies in a PLL:

- *Capture range*: the frequency range around the VCO's free-running frequency within which the PLL can initially acquire lock with the input signal.
- *Lock range*: once the PLL is locked, it can maintain synchronization with the input signal over a wider frequency range than the capture range.

The main applications of a PLL include:

- *Clock multiplier* or *clock generator*: generates a clock signal that is a multiple of the input frequency or provides multiple clock outputs.
- *Frequency synthesizer*: produces a clock signal with an arbitrary frequency by dividing or multiplying a reference clock. A frequency divider is placed between the VCO output and the phase comparator. The output frequency is divided down before being feedback to the phase detector. As long as the loop remains locked, the VCO output will be an integer multiple of the input frequency.

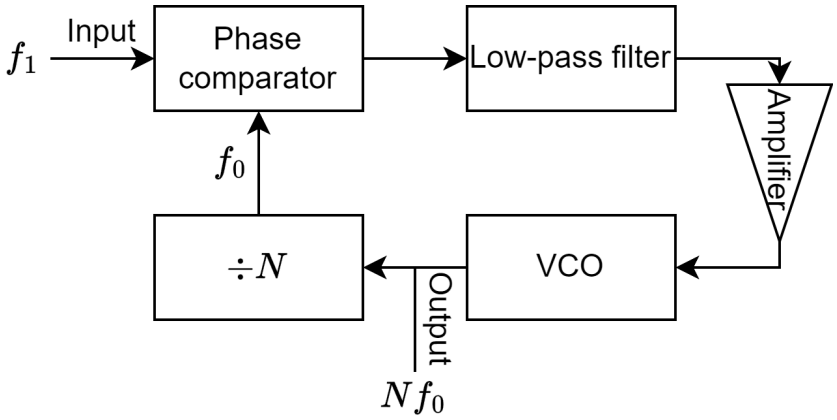

Figure 2.4: Frequency synthesizer

- *Clock and data recovery*: recovers digital data and a synchronized clock signal from a serial data stream, using a specialized phase detector.
- *Frequency and module demodulation*: demodulates a radio signal by tracking the frequency modulation and converting it into a usable signal.

Integer synthesizer An integers-N synthesizer is created by using a stable, high-frequency crystal oscillator and divide its output. The output frequency f_o can be expressed as:

$$f_o = f_{\text{ref}} \frac{n}{r}$$

Here, f_{ref} is the reference frequency, n and r are programmable integer values that can be selected based on system requirements.

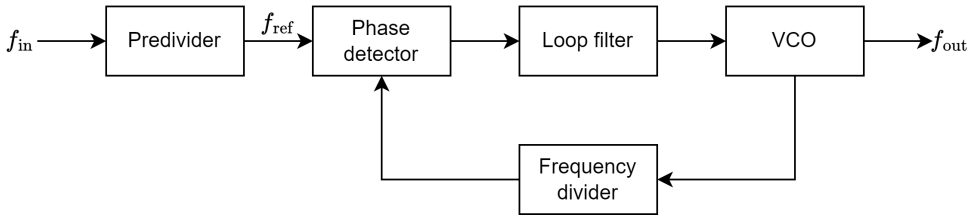

Figure 2.5: Integer-N synthesis

Prescalers Prescalers are frequency dividers that reduce the frequency of the VCO before it reaches the phase detector and extend the frequency range of a synthesizer. A four-modulus prescaler provides four scaling factors and uses two control signals to select one of the available factors.

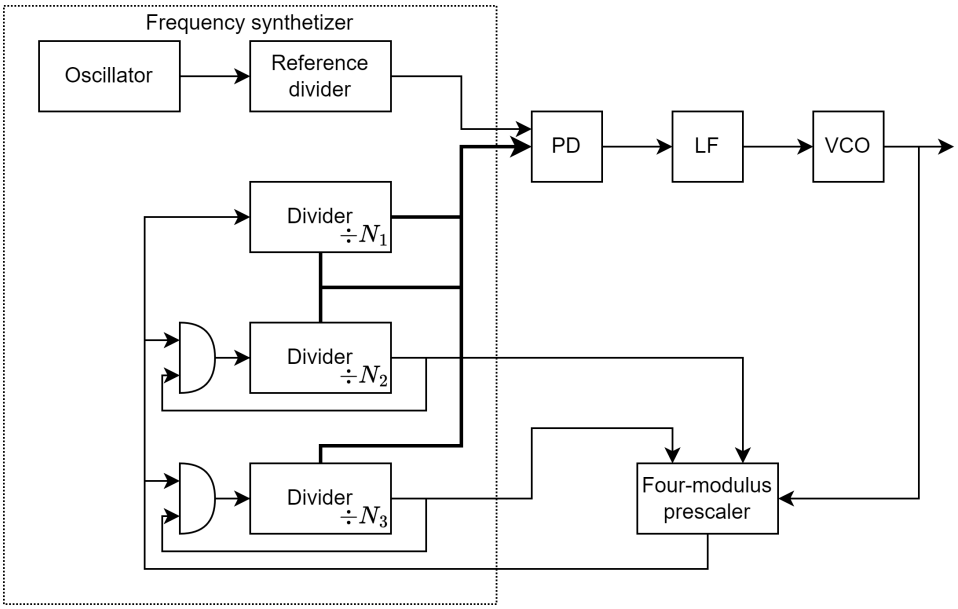

Figure 2.6: Integer-N synthesis with prescalers

This design allows dynamic adjustment of clock speeds to optimize performance and power consumption. By combining prescalers with PLLs, systems can adapt to variable computational loads while maintaining efficient power use.

2.3.1 Timers

A programmable timer is a specialized clock used to measure time intervals or count events.

	Timer	Counter
Increment	The register is incremented for every machine cycle	The register is incremented based on a 1 to 0 transition at an external input pin (T_0, T_1)
Maximum count rate	$\frac{1}{12}$ of the oscillator frequency	$\frac{1}{24}$ of the oscillator frequency
Source of signal	A timer uses the frequency of the internal clock to generate delay	A counter uses an external signal to count pulses

Counters are similar to timers but are designed to count external events instead of clock cycles.

Registers Timers are controlled through some special function registers, such as:

- *Gate*: the timer only runs while an external interrupt is active.
- *Start and stop control*: timers can be started and stopped via software or hardware control.
- *Counter or timer select bit*: this bit determines whether the module operates as a timer or a counter.

Reading There are two primary values to read from a timer:

- *Timer value*: read the actual count value stored in the timer registers.
- *Timer overflow*: monitor the timer overflow flag, which is set when the timer reaches its maximum value.

Settings Timers can be used to perform two main tasks:

- *Polling*: continuously reading the status registers to check for timer events or the current counter value. Polling can consume significant processing time and may introduce variability in response times, especially in complex programs.
- *Interrupts*: generate an interrupt when a specific event occurs. This method ensures fast and predictable responses to timer events, without requiring the main program to check for them continuously.

Structure The basic structure of a timer in an microcontroller consists of several essential components:

- *Clock source*: multiple clock sources may be available, with the possibility of selecting an external clock if needed.
- *Prescaler*: divides the input clock by a factor.
- *Main counter*: after the prescaler, the clock feeds into a main counter that increments or decrements depending on the timer mode.
- *Modulus value*: the main counter's range is controlled by a modulus value, which can be programmed into a register.

- *Control logic*: the control logic defines the operational mode of the timer.

Based on the usage, the timers can be:

- *Periodic*: generate repetitive events or ticks with a fixed period (determined by the modulus value).
- *Delay*: execute actions after a specific time interval has passed. This function works by resetting the timer to zero and waiting for a defined number of ticks before triggering the desired action.

Timers and counters are some of the most critical peripherals in microcontroller designs, enabling a wide variety of functions that improve performance, reduce power consumption, and simplify designs. Timers and counters can be used in virtually any application to enhance performance, reduce power usage, or streamline design by replacing CPU-based operations with interrupt-driven tasks. Many COTS microcontroller devices include built-in support for programmable timers. Manufacturers provide development kits and reference designs tailored to specific applications that rely heavily on programmable timers.

2.3.2 Watchdog timer

A watchdog timer detects software anomalies and takes corrective actions to recover from system faults, ensuring reliability. Watchdog timers can be implemented as either external chips or integrated within the microcontroller itself. They may offer fixed or programmable timeout intervals.

In normal operation, the system regularly kicks or resets the watchdog timer to prevent it from reaching its timeout. If a hardware fault or software error occurs, the system may fail to reset the timer and the watchdog will time out and generate a corrective action. Corrective actions may vary depending on the fault.

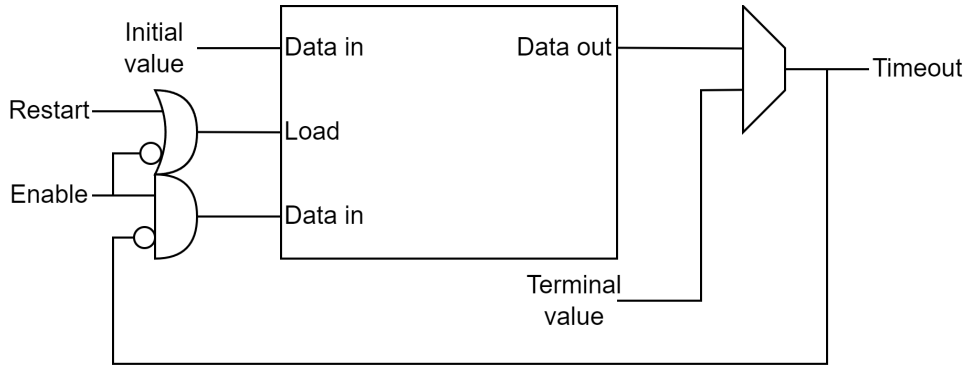

Figure 2.7: Watchdog architecture

The watchdog timer operates as a countdown timer, which is periodically reset by the system. When the countdown reaches zero, a timeout signal is generated. While the program can modify run mode states at any time, safe mode states can only be changed through a special write mechanism. When the watchdog times out, the data selector switches to safe mode states. Since the control circuitry remains operational, it retains vital state information, which may be needed for fault recovery.

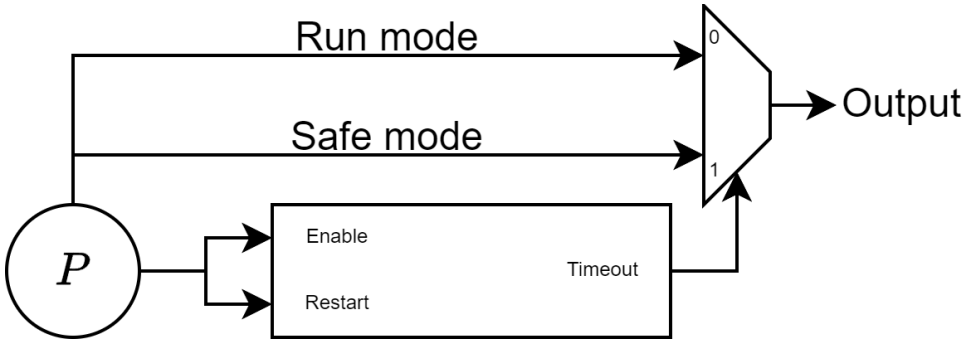

Figure 2.8: Watchdog mode

A system may enable the watchdog only when it is needed, while in other cases, watchdogs are automatically activated upon system boot and cannot be disabled (boot fault detection). When a system resets, watchdog timers are typically disabled until the control program explicitly enables them. When the application is terminating, it should disable the watchdog after safely shutting down all outputs and stopping control operations.

When a fault is detected, the watchdog performs the following actions actions:

1. *Safe state transition*: the system immediately sets all control outputs to safe levels as soon as the fault is detected.

2. *System recovery*: normal operation can be restored.

Single stage watchdog A single-stage watchdog timer is a design where an immediate system restart is triggered upon timeout. The system relies on this reset to bring the control outputs to their safe states. Both the microcontroller and the watchdog timer may share the same clock signal for synchronized operation.

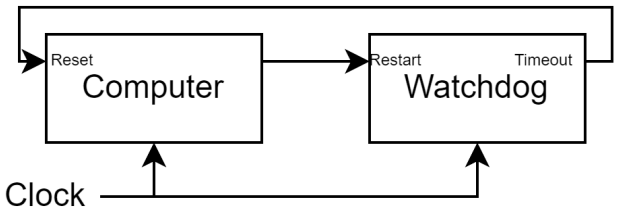

Figure 2.9: Single stage watchdog

Multiple stage watchdog A multi-stage watchdog consists of two or more timers arranged in a cascade. Only the first stage is reset by the processor, and upon its timeout, subsequent stages are initiated, each triggering corrective actions before starting the next stage. Upon the timeout of the final stage, no further stages are initiated, and typically a full system restart is triggered. Multi-stage watchdogs enable a series of corrective actions that happen in stages before restarting the system.

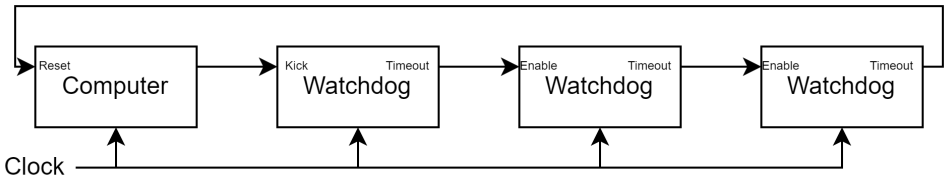

Figure 2.10: Multiple stage watchdog

A common technique in multiple stage designs is using a dedicated control reset signal, which resets the control circuitry without resetting the computer itself. Watchdog timers can trigger various corrective actions, including maskable interrupts, non-maskable interrupts, processor resets, fail-safe state activations, power cycling, depending on the system architecture. Abrupt system restarts due to watchdog timeouts can be costly, both in terms of downtime and the potential loss of important state information. A multi-stage watchdog mitigates this by providing a more controlled recovery process:

1. The watchdog switches control outputs to safe states immediately.
2. The watchdog schedules a deferred restart and signals the computer, allowing the program time to attempt recovery or log critical state information.
3. If recovery is successful, the scheduled restart can be canceled.

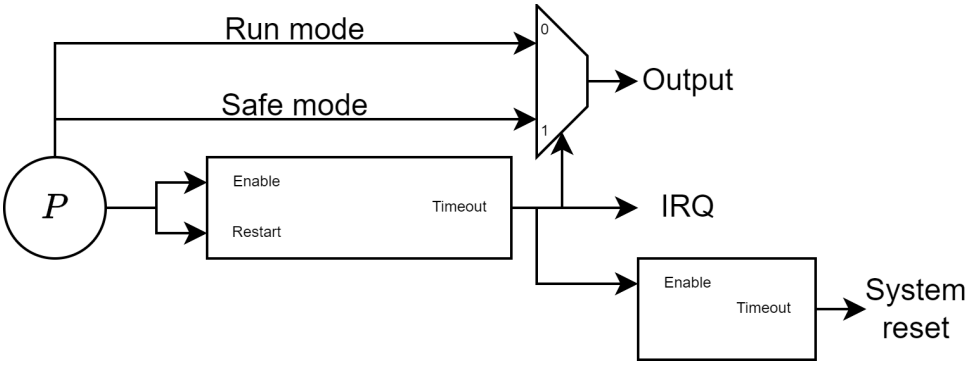

Figure 2.11: Watchdog recovery

Taxonomy The main types of watchdogs are:

- *AVR*: this watchdog is equipped with a special timer (WDT), which operates independently of the main system clock. It acts as a counter that increments with each clock cycle from the oscillator, forcing an interrupt or a system reset once the counter reaches a pre-configured timeout value. In normal operation, the application code must issue a reset (WDR) instruction to reset the counter before it reaches the timeout value. If this reset does not occur, the system will either generate an interrupt or trigger a system reset, depending on the selected watchdog mode. There are three primary modes in which this watchdog can operate:
  - *Interrupt* (WDTIE bit): when the watchdog timer expires, it generates an interrupt.
  - *System reset* (WDE bit): when the watchdog timer expires, it forces a system reset.

  - *Interrupt and system reset*: when the watchdog timer expires, an interrupt is generated first, allowing the system to save critical parameters or execute a graceful shutdown. Afterward, a system reset follows to restart the system safely.

- *Smart*: function as a supervisory microcontroller that monitors also system communications. In some cases, the microcontroller might stop responding to network commands, while still resetting the watchdog timer as usual. A smart watchdog addresses this by monitoring communication lines.

- *External*: offers several advantages that contribute to system robustness. This kind of watchdog provides an autonomous process to monitor the system's health, significantly increasing system reliability. However, the additional hardware required increases both the complexity of the system and the overall cost.

Best practice To maximize the effectiveness of watchdog timers, several best practices should be followed. Disabling the watchdog for any reason should be avoided, as it opens the system to vulnerabilities. Additionally, clearing the watchdog timer in periodic interrupts without first performing functionality checks is a common mistake that can lead to runaway software overriding the watchdog's safety mechanism.

It's important to use an independent watchdog timer with a separate clock, allowing it to detect when the system clock has halted. Implementing a windowed watchdog is also recommended. A windowed watchdog enforces a minimum time delay before the watchdog can be reset. This ensures that if runaway software tries to clear the watchdog too quickly, the system will still be reset.

2.4 System boot

Definition (*System boot*)**.** System boot refers to the sequence of steps that transitions a system from a powered-off state to a fully operational state.

The complexity of the system boot process can vary depending on the type of system involved, as it requires the coordination of both hardware and software components. These components typically include the hardware platform, the microcontroller or microprocessor, the BIOS, the bootloader, the operating system, and the application layer.

A critical phase within this sequence is the point when the BIOS begins execution. At this moment, the hardware initialization has completed, and the first instruction of the system firmware is executed.

2.4.1 Microcontroller boot sequence

we consider a typical system with a microcontroller, flash memory to store the binary executable, and RAM to hold data during execution. We assume the binary contains the complete code required for operation. In our case, the code is executed directly from flash memory. The scenario where code is executed from RAM, which is common in general-purpose microprocessor-based or larger embedded systems, follows a similar process.

Flash memory is typically organized into different sections, including:

- *Text*: stores the executable code. Here, we have three logically distinct types of information:

1. *Interrupt Vector Table* (IVT): a fixed-size table that contains the initial stack pointer, the address of the reset handler, and the addresses of the interrupt service routines.
2. *Startup code*: a small portion of assembly code (usually provided by microcontroller vendor) responsible for initializing the system.
3. *Application code*: the main executable code of the application.

- *Rodata*: holds constants.
- *Data*: contains initialized variables.
- *Bss*: holds uninitialized variables.

Boot process When the microcontroller exits the reset state, it begins by loading the program counter with the address from the second entry in the IVT, which points to the reset handler. The microcontroller then starts executing the code at this address. In more general cases, the location of the IVT base may not be fixed but is stored in a specific register in the microcontroller. While the exact position of the reset handler in the table may vary by microcontroller model, having it as the second entry is quite common.

Once the reset handler begins execution, it performs the following key operations:

1. *Stack pointer initialization*: it loads the initial stack pointer address into the stack pointer register. From this point on, the stack is operational, allowing functions to be called.
2. *System initialization*: the reset handler invokes a system initialization function, which usually sets up the system's clocks and waits for the PLL to stabilize.
3. *Memory preparation*: the memory is prepared by: copying the data section from flash to RAM (based on linkler directives), and zeroing out the bss section in RAM.
4. *Jump to* `main`: the reset handler jumps to the `main` function, which serves as the entry point for the application or the application combined with the OS. This jump is not a function call, meaning execution transfers directly to main without returning to the reset handler.

2.4.2 Bootloader

A bootloader is essential for firmware updates in embedded systems. Unlike typical development environments where the application is loaded into the microcontroller's flash memory using a programmer, the bootloader allows the system to update its firmware in the field without external programming tools. The bootloader is a small program designed to handle three main tasks:

- *Receiving a firmware binary*: the bootloader obtains the new firmware through a communication interface.
- *Writing the firmware to flash memory*: the new firmware is written into flash, replacing the old version, often after validating its integrity.
- *Executing the updated firmware*: once the new firmware is successfully loaded, the bootloader transfers control to the new application.

When a bootloader is present, it is the first software to execute upon system startup. It occupies its own dedicated area in flash memory, coexisting with the main firmware. However, the bootloader and the application each maintain separate IVTs, which is crucial for switching between the two. The bootloader's IVT is located at the default address, and its role ends once it successfully transfers control to the new firmware.

Suppose that the bootloader has already perfromed the firmware update. At this point, the bootloader must hand over control to the new firmware. This involves switching to the application's IVT and starting the new firmware execution. How this is done depends on whether the microcontroller has an IVT offset register:

- *With IVT offset register*: first, it prepares for the handover by disabling all interrupts and clearing any pending ones. This is crucial to prevent any interrupt from being serviced using the bootloader's IVT while the bootloader itself is still executing. The system must also ensure that all memory operations are fully completed before proceeding, to avoid any issues with uncommitted data. Once these preparatory steps are completed, the bootloader executes the new firmware by updating the IVT offset register with the address of the new IVT. The stack pointer is then loaded with the value found in this updated IVT, ensuring the system is ready to handle interrupts and function calls as intended by the new firmware. Finally, control is transferred by jumping to the reset handler address specified in the new IVT, which starts the execution of the new firmware.

- *Without IVT offset register*: in systems lacking an IVT offset register, the IVT may be stored at two fixed addresses: one in flash memory and the other in RAM. Typically, the bootloader operates with its IVT in flash. Here, the bootloader begins by disabling all interrupts and clearing any pending ones. The next step is to copy the new IVT from flash memory to RAM, as this will be the new location for interrupt handling during the firmware's operation. To execute the new firmware, the bootloader enables memory remapping, which switches the system to use the newly copied IVT in RAM. The stack pointer is then loaded from the new IVT, ensuring the system is properly configured for the upcoming firmware execution. Finally, the bootloader jumps to the reset handler in the new IVT, thereby initiating the execution of the new firmware.

2.5 Microcontrollers IDE

Microcontrollers are primarily designed to host the final application, making them less suitable for code development processes compared to traditional PC-based environments. Given the limitations of microcontrollers, there arises a need for an Integrated Development Environment (IDE). An IDE is not just a simple code editor but a comprehensive suite that supports editing, versioning, cross-compilation of code, debugging, profiling, and integration with evaluation boards. Modern IDEs offer additional features, such as debugger integration, device libraries, support for real-time operating systems (RTOS), and code templates to simplify and accelerate development.

2.5.1 MDK Professional

MDK Professional is a comprehensive development suite that supports software development for a wide range of devices. It caters to both secure and non-secure applications. MDK Professional is well-suited for Internet of Things applications requiring secure network connectivity

to the cloud, as well as projects that rely on proven middleware components. The suite includes the Arm C/C++ Compiler, which is optimized for small code size and performance, and features highly optimized runtime libraries. Additionally, MDK supports the installation of Software Packs, that include device support, CMSIS libraries, middleware, board support, code templates, and example projects.

2.5.2 MuVision

The μVision IDE integrates project management, build tools, a runtime environment, source code editing, and program debugging into a single environment to accelerate embedded software development. It supports multi-screen configurations, allowing users to create individual window layouts on their visual workspace. The μVision Debugger provides a unified environment to test, verify, and optimize application code. It includes features such as simple and complex breakpoints, watch windows, and execution control, while offering full visibility into device peripherals. Software applications can be created using pre-built software components and device support from Software Packs, which may include libraries, configuration files, source code templates, and documentation.

2.5.3 Parasoft C/C++

Parasoft C/C++ test is a complete quality testing solution that enhances software development team productivity and improves the quality of C and C++ applications. Its key features include static code analysis, coding policy enforcement, automated code reviews with graphical progress tracking, unit and regression testing, and code coverage analysis. Additionally, Parasoft C/C++test uses the high-speed streaming trace capabilities of the ULINKpro adapter to capture performance and code coverage data, which is then analyzed using the MDK-ARM development kit. It also includes advanced features such as value tracking of variables and strong typedef-based type checking to detect issues early in the development cycle.

2.5.4 ULINK Debug Adapters

The ULINK family of USB Debug Adapters connects a PC's USB port to a target system, enabling developers to debug and analyze embedded programs running on target hardware. The ULINKpro provides advanced features like execution profiling and code coverage through direct streaming trace to a PC. In addition to supporting real hardware targets, developers can use Fixed Virtual Platforms (FVPs) for software development without the need for physical hardware. FVPs simulate an entire Arm system, including processors, memory, and peripherals, allowing developers to start bare-metal coding and Linux application development at speeds comparable to real hardware.

2.5.5 Code Composer Studio

Code Composer Studio (CCS) is an integrated development environment designed for Texas Instruments' microcontroller and embedded processors portfolio. It combines the advantages of the Eclipse software framework with advanced embedded debugging capabilities, providing a feature-rich environment for embedded developers. For high-performance processors the highly-optimizing C/C++ VLIW compiler can perform various optimization techniques, to accelerate algorithms. The compiler is regularly validated against industry-standard benchmarks, ensuring stability and performance across releases.

Many high-performance Texas Instruments processors offer the capability to perform processor trace, providing a detailed historical account of code execution, timing, and data access patterns. Trace data can be captured in dedicated on-chip memory or exported for external analysis.

System analyzer The system analyzer tool provides visibility into the application, operating system, and hardware by correlating software and hardware instrumentation across multicore systems. It consists of two core components:

- *Unified Instrumentation Architecture* (UIA): a software package for logging, runtime control, and data movement.
- *Analysis displays*: tooling for data collection, decoding, analysis, and visualization.

System analyzer can capture data from the on-chip embedded trace buffer or stream it off-device via the system trace receiver. It offers real-time insight into system behavior, allowing developers to optimize and debug their applications.

Linux development CCS supports both Linux kernel and application development. The Linux kernel can be debugged via JTAG, while application development can be conducted using GDB. Additional functionality, such as Linux Trace Tools (LTTng), can be installed to enhance the development environment.

2.5.6 IAR Systems

IAR Embedded Workbench is a versatile development toolchain supporting multiple platforms and architectures, including RISC-V. IAR Systems offer a complete development environment encompassing all aspects of embedded software development with powerful functionality. IAR Embedded Workbench targets multiple architectures and offers a dedicated team of experts to assist developers.

The IAR Evaluation Kit includes a 30-day evaluation license for IAR Embedded Workbench for RISC-V, a GigaDevice Evaluation Board, and the I-jet Lite debug probe, as well as access to IAR Academy's online course for RISC-V development.