Department of Information Engineering and Computer Science

Master's Degree in

Computer Science - ICT Innovation - Embedded Systems

FINAL DISSERTATION

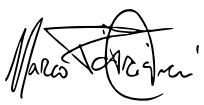
Xtrooder - An embedded firmware for 3D printers, written in Rust

Supervisor

Marco Patrignani

Student

Federico Pezzato



Academic year 2023/2024

Acknowledgements

I would like to thank my parents, Alessandra and Maurizio, for allowing me to follow my dreams, supporting me in the most difficult times and accompanying me on this long journey. I love you, you are the best.

I would like to thank my fiancée, Matilde, for comforting me, bringing patience, and standing by me in every difficult situation. I love you, baby.

I would like to thank my brother, Alessandro, for influencing me with his passion for computer science since I was little, involving me in stimulating activities, giving me insights, and always judging my crazy ideas with a critical eye. I love you, big brother.

I would like to thank Alberto, Andrea, Davide and Sebastiano for keeping me company since the first year of my three-year degree, for all the good times we had together, for the laughs and nights spent gaming. I love you, guys.

I would like to thank Francesco for the long walks and random car rides to listen to newly released albums. I love you, slime.

Contents

1 Abstract

In recent years, 3D printing has emerged as a life-changing technology in most industries, enabling rapid prototyping and replicating almost every kind of component. Its versatility and accessibility allow on-demand production, reduced waste, and the creation of highly customized solutions tailored to specific needs. This paradigm shift has opened up new possibilities in fields such as medicine, where "*surgeons, for example, can use 3D-printed models derived from CT scans to plan intricate surgeries*"[21] and in aerospace engineering, where *"companies like Airbus are now leveraging this technology to produce complex components and high-temperature alloys, and due to the precision of 3D printing processes like selective laser sintering (SLS) and fused deposition modeling (FDM), manufacturers can make aircraft parts lighter, stronger, and more cost-efficient."*[21].

At the core of every 3D printer lies its firmware — a critical layer of software that controls the hardware, orchestrates movements and interprets commands from user applications. It acts as the brain of the printer, managing everything from precise stepper motor coordination to temperature regulation and communication with the host computer. The reliability, performance, and capabilities of a 3D printer heavily depend on the efficiency and robustness of its firmware.

Traditionally, most firmware for 3D printers has been developed with native and relatively old programming languages like C or C++. These languages have dominated embedded systems development for decades, offering fine-grained hardware access, extensive libraries, and compatibility with various platforms. Their longevity has also led to the establishment of standards and best practices, allowing developers to write functional and efficient code.

However, as 3D printing applications grow more diverse and complex, the limitations of these traditional approaches become increasingly apparent, and the demand for safe, maintainable, and expressive firmware has grown increasingly urgent. Open-source firmware for CNC and 3D printers presents common challenges:

- Code base - These firmware systems are predominantly written in C or C++, which are inherently unsafe, from the manual low-level memory management to limited built-in error handling. Such characteristics make the code prone to bugs like segmentation faults, buffer overflows, and memory leaks. Furthermore, undefined behavior and unchecked runtime errors can result in severe issues, including malfunctions or safety risks, particularly in production environments where stability is critical. The lack of memory safety not only makes debugging difficult but also complicates efforts to maintain and scale the code base over time.

- Concurrency - Even though a 3D printer has numerous components requiring cooperative execution—such as temperature control, motion planning, communication, and sensor feedback — many, popular firmware solutions lack true concurrency. Most firmware does not operate on an RTOS (Real-Time Operating System) or employ concurrent execution models, which are essential for efficient resource utilization and code modularity. This limitation arises partly from legacy support for older boards (e.g., Atmel boards with 8-bit architectures), which are unable to meet the demands of modern embedded RTOS. As a result, these systems rely on a main loop architecture where all tasks are sequentially executed. For instance, Marlin[14] (the most used firmware in the market of general-purpose 3D printers) processes tasks like temperature monitoring, motion planning, and serial communication in a single loop, leading to significant delays under heavy load and potential performance bottlenecks. By contrast, Klipper[12] (another firmware for 3D printers, which is less popular but really powerful) adopts a more modular approach, using two separate systems: a host computer, typically a Raspberry Pi, to handle computationally intensive tasks like motion planning, and a micro-controller to execute low-level operations such as stepper motor control and temperature monitoring. These components communicate via a binary protocol over a serial connection. While this approach alleviates some performance issues, it introduces complexity and hardware dependencies, requiring additional components and setup.

Nowadays, the cost and size of a 32-bit micro-controller have dramatically dropped compared to 15 years ago, and more modern and powerful general-purpose programming languages have arisen, opening up new paths to cope with those problems, which is mandatory in order to provide a reliable and power-efficient system that just runs.

This thesis proposes a novel approach: Xtrooder, an asynchronous, efficient, and memory-safe firmware written in safe Rust. Xtrooder aims to address the limitations of traditional 3D printer firmware by leveraging the modern features of the Rust programming language, such as memory safety, zero-cost abstractions, and a strong emphasis on concurrent execution. It is built on top of the Embassy[3] (Embedded-Async) framework and introduces a paradigm shift in firmware design, replacing the existing monolithic and sequential architectures with a more modular and maintainable solution.

At its core, Xtrooder adopts a multi-tasking environment combined with asynchronous HALs (Hardware Abstraction Layers), enabling peripherals to be set up and used seamlessly with intuitive async-await syntax sugar. This approach allows for more efficient utilization of micro-controller resources, reducing latency and improving overall system responsiveness. Furthermore, the usage of Embassy[3] provides additional benefits such as precise task scheduling, minimal runtime overhead, and compatibility with modern embedded hardware. This allows Xtrooder to fully leverage the capabilities of contemporary 32-bit microcontrollers, making it both future-proof and adaptable to various hardware configurations. Integrating these features, it not only simplifies the development process for firmware engineers but also enhances the reliability and safety of 3D printers, reducing the risk of runtime errors and undefined behavior.

The use of Rust guarantees zero-cost abstractions, enabling the modeling of different functional units of the printer without introducing any additional overhead. This ensures that even with a high level of encapsulation and modularity in the code base, there is no compromise on performance. Each abstraction is designed to translate directly into efficient, low-level operations, making Rust an ideal choice for embedded systems where resource optimization is paramount.

Furthermore, the memory architecture in this firmware has been carefully designed to operate on a stack-only model, eliminating the need for dynamic memory allocators typically provided by a classical OS. By avoiding heap-based allocations, the system achieves a higher level of predictability in memory usage, which is crucial for real-time applications like 3D printing. This approach ensures consistent behavior under all operational conditions, reducing the risk of memory fragmentation or unpredictable runtime errors.

In addition, all tasks required for printer operation—such as driving stepper motors, reading and controlling temperatures, managing heaters, and handling external communication—are statically allocated at compile time. These tasks are transformed into efficient state machines using Embassy[3] macros, which are automatically scheduled for execution by a runtime orchestrator, called "futures executor" in Rust lingo. The orchestrator optimizes task execution, ensuring that no resources are wasted and tasks are run in an order that minimizes power consumption and avoids contention. This approach also eliminates the complexities of per-task stack size tuning.

The reliability of this firmware has been rigorously tested through an extensive suite of automatic host and on-board tests, ensuring robustness and stability in both development and production environments. The testing framework leverages Rust's built-in unit and integration testing capabilities, enabling thorough validation of core logic and functional units directly within the development process. Additionally, specialized crates designed for runtime testing on micro-controllers were employed to verify the behavior of the firmware in real-world conditions, further enhancing confidence in its correctness.

The remaining part of the thesis is organized as follows:

- Background: This section begins with an overview of embedded systems and their main peripherals, such as GPIO, timers, and ADC, followed by an explanation of the different physical components of a 3D printer and how they interact. The discussion then transitions to the Rust no-std environment, explaining which are the main differences from the standard environment, along with an overview of the Rust Future trait, explaining its design principles and why it has become a cornerstone of modern Rust development. A detailed overview of the Embassy[3] framework follows, with an emphasis on how its runtime leverages Direct Memory Access (DMA) to optimize the Future trait's execution within the HAL (Hardware Abstraction Layer), and how its asynchronous task scheduler works.

- Software developement: This section provides an explanation of the architecture of the project, including a breakdown of the key dependencies and custom-developed crates, and an explanation of the implementation of the various functional units of a 3D printer. The section also discusses how these abstractions are integrated into independent asynchronous tasks, leveraging the runtime provided by Embassy[3] for scheduling and execution, and how those tasks share data. The final part of the session talks about the error handling.

- Test and validation: This section evaluates the system's performance by comparing it against Marlin[14] firmware across various metrics, such as binary size, platform support, and ease of development. It then details the testing methodologies employed, including unit tests, integration tests, and on-device runtime validations. These tests ensure that the firmware meets its design objectives and functions reliably under real-world conditions.

- Related work: This section examines existing projects, frameworks, and guides that inspired and supported the development of this firmware. It provides an overview of open-source firmware projects, like Marlin[14] and Klipper[12], and analyzes their architectures and limitations.

- Conclusion and future work: The thesis concludes with a summary of the main achievements, emphasizing the firmware's key features. It reflects on the challenges encountered during the project and how they were addressed. Additionally, this section identifies areas for future work, highlighting functionality that the current firmware does not yet implement.

2 Background

Writing firmware for 3D printers requires a non-trivial set of information to know, from the hardware generally used to drive every component, to the different features of the programming language that has been used. That's why the following sub-sections will guide the reader across these topics:

- **Embedded systems**: contains an explanation of the most common peripherals used by an MCU to control a 3D printer

- **3D Printer**: contains an explanation of the main components of a 3D printer and how they interact with peripherals

- **Asynchronous programming in Rust**: shows the asynchronous features of the Rust programming language along with a comparison with the well-consolidated thread programming

- **The Embassy framework**: lists the main features of the Embassy[3] framework, which has been used as a building block for Xtrooder

2.1 Embedded systems

The control unit of a 3D printer is typically a microcontroller, often integrated into a development board. Due to the constrained nature of their design, these units are significantly more resource-limited than general-purpose computers. Their compact size and minimal power consumption make them ideal for tasks like 3D printing, but this comes at the cost of reduced computational capabilities and storage. For instance, microcontrollers commonly found in most control boards for 3D printers may offer RAM ranging from 256 KB to 1 MB, non-volatile flash memory around 1 MB, and clock frequencies in the range of tens to hundreds of MHz. These constraints directly affect the system's ability to handle complex computations or large datasets, requiring highly optimized code and lightweight binaries. The following subsections will show which are the main peripherals that a 3D printer generally uses and how they interact with the external environment.

2.1.1 GPIO (General-Purpose Input/Output)

The GPIO peripheral provides configurable pins that serve as the primary interface between a microcontroller and the external world. These pins can be set as either input or output, making them versatile for various digital signal operations. In input mode, it reads digital signals (high/low) from external devices like buttons, sensors, or switches. In output mode, it outputs digital signals to control devices like LEDs, relays, or motors. Pins are often grouped in ports (8 to 16), defined by letters (e.g. as we see in fig. 1, on the top and bottom borders of the MCU)

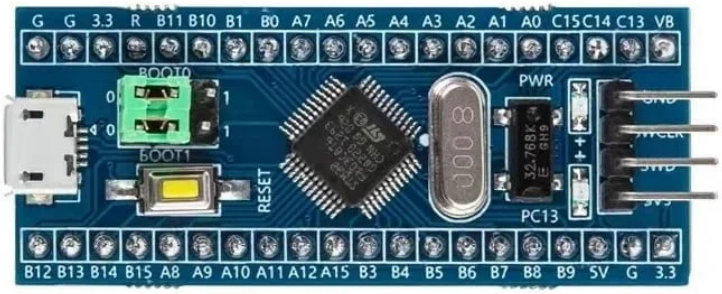


Figure 1: STM32 Blue Pill

2.1.2 Timer

Timers are critical peripherals in microcontrollers, designed to measure time intervals, count events, or generate precise timing signals. In a firmware for 3D printers, timers are used to generate pulses to drive stepper motors and heaters, and to perform periodic operations efficiently. They works by incrementing a counter at a rate determined by their clock source, often derived from the microcontroller's main system clock. This versatility enables them to perform tasks such as adjusting timing precision with a pre-scaler, detecting or generating events when the counter reaches specific values, and triggering actions at predetermined intervals or conditions. Most of the time, different peripherals are driven by different timers. In STM32 microcontrollers, for instance, the primary clock source, referred to as the system clock (SYSCLK), can originate from different oscillators, such as an HSI (High-Speed Internal Oscillator) which is a reliable built-in RC oscillator typically running at 16 MHz, or an HSE (High-Speed External Oscillator) which uses an external quartz crystal to achieve greater stability and accuracy. The SYSCLK serves as the base clock and is distributed to different subsystems after being processed through pre-scalers. As we can see in fig. 2 The HCLK is derived from SYSCLK and it is used to drive the CPU and the AHB (Advanced High-Performance Bus), the main backbone for high-speed data transfers between memory, CPU and DMA. From the HCLK, another pre-scaler generates the clock signals for the APB (Advanced Peripheral Bus). The APB is divided into two branches: APB1 and APB2. APB1, often referred to as the low-speed bus, drives peripherals such as general-purpose timers, while APB2, the high-speed bus, powers advanced peripherals like SPI.

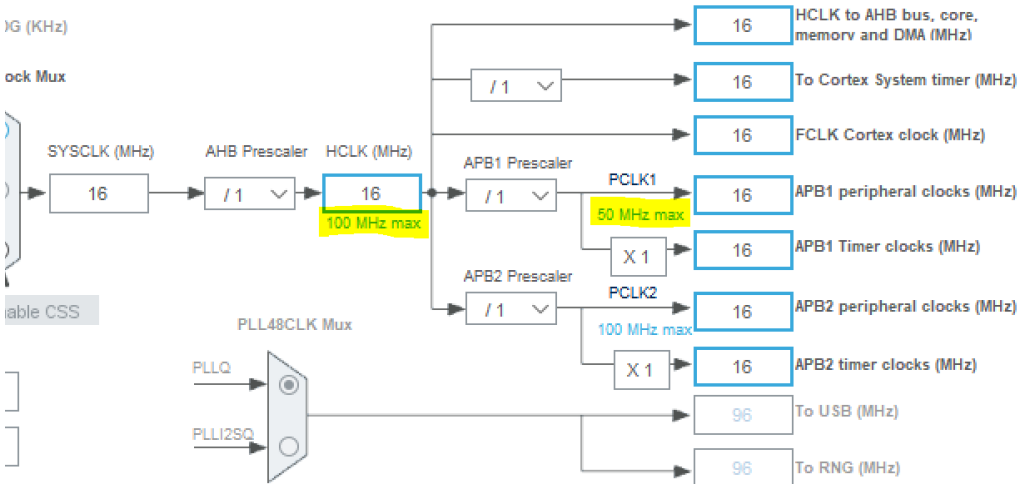


Figure 2: Timers in a generic STM32 MCU

PWM Pulse Width Modulation (PWM) is a method for generating analog-like signals using digital outputs. This technique works by rapidly toggling a signal between high and low states at a specific frequency, with

the proportion of time spent in the high state (known as the duty cycle) determining the effective output signal strength. For example, a higher duty cycle corresponds to a stronger or higher average signal level. By modulating the duty cycle, PWM enables precise control over devices such as motors, LEDs, and other peripherals that respond to variable power levels. In STM32 microcontrollers, PWM is implemented using timers, which are highly efficient and hardware-accelerated. General-purpose timers in STM32 typically support up to four PWM channels, corresponding to four independent counters. Each channel can produce a PWM signal by counting down from a predefined value to zero. When the counter value matches a specified compare value, the signal on the corresponding GPIO pin toggles between high and low states. If the switching is performed at a high frequency (between 1KHz and 10KHz), the output signal can be perceived as an analog signal with a voltage equal to

$$V_{out} = \frac{d}{d_{max}} \cdot V_{in}$$

where d is the duty cycle, d_max is the maximum duty cycle, V_in the input voltage and V_out the voltage of the produced analog signal.

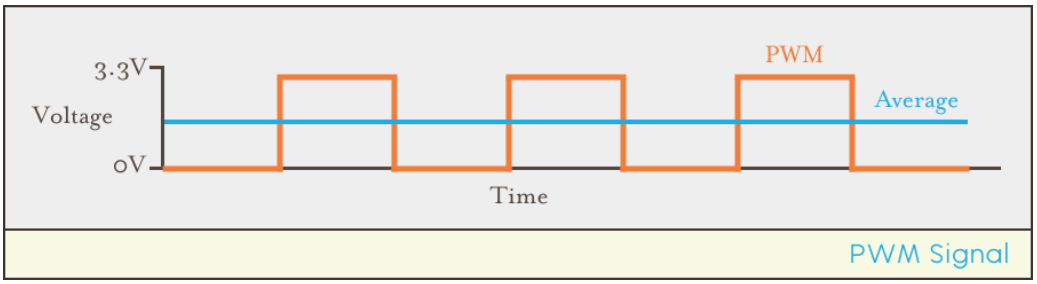


Figure 3: Analog signal using PWM

2.1.3 ADC (Analog-to-Digital Converter)

An Analog-to-Digital Converter (ADC) is responsible for transforming continuous analog signals into discrete digital values that can be processed by digital systems. It achieves this by sampling the input signal at regular intervals and quantizing the sampled values into binary form based on a specified resolution and reference voltage. The resolution of an ADC, typically expressed in bits (e.g., 8-bit, 10-bit, 12-bit), determines the number of discrete levels into which the input signal can be divided. A higher resolution provides greater precision, allowing the ADC to distinguish finer variations in the input signal. For instance, a 12-bit ADC can resolve the input signal into an integer range between 0 and

$$2^{12} = 4096$$

values, while an 8-bit ADC between 0 and

$$2^8 = 256$$

The ADC operates relative to a reference voltage, which defines the upper limit of the input signal range. This reference is often 3.3V or 5V, depending on the operating voltage of the microcontroller. The digital output is proportional to the input signal's magnitude relative to the reference voltage. For example, if the reference voltage is 3.3V and the ADC resolution is 10 bits, an input of 1.65V would yield a digital value close to

$$O = \frac{1.65}{3.3} \cdot (2^{10} - 1) = 512$$

2.1.4 UART (Universal Asynchronous Receive/Transmit)

The UART (Universal Asynchronous Receiver-Transmitter) peripheral is a commonly used hardware interface that enables serial communication between devices, facilitating the exchange of data one byte at a time. Unlike synchronous communication protocols, UART does not require a shared clock signal between the transmitting and receiving devices. Instead, it relies on predefined timing parameters, such as the baud rate, which determines the rate at which data is transmitted (measured in bits per second, bps). This approach ensures that the

data is correctly interpreted by both devices despite the lack of a synchronized clock signal. UART supports full-duplex communication, meaning data can be transmitted and received simultaneously, through two separate lines: one for transmission (TX) and the other for reception (RX). The basic UART communication setup only requires two wires for data transfer (TX and RX), in addition to a ground (GND) line, (as shown in fig. 4) which serves as a common electrical reference for both devices. This simplicity in wiring is one of the key reasons UART is widely adopted in embedded systems, microcontrollers, and serial communication interfaces.

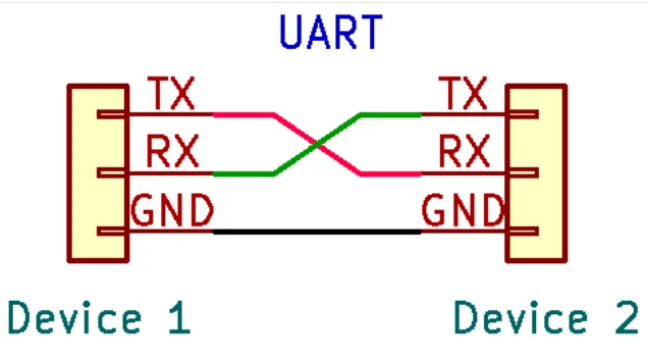


Figure 4: Wiring between two devices that communicate via UART

2.1.5 SPI (Serial Peripheral Interface)

SPI (Serial Peripheral Interface) is a synchronous serial communication protocol commonly used for data transfer between microcontrollers and peripheral devices. It is often used for its simplicity, high-speed capabilities, and support for full-duplex communication. The SPI protocol operates in a master-slave architecture, where one master device controls the communication, and one or more slave devices respond. The master generates a clock signal to synchronize the data transfer with the slaves, ensuring proper timing and alignment of bits. SPI uses four primary communication lines to facilitate data exchange. These lines include:

- MOSI (Master Out, Slave In): This line is used to send data from the master to the slave device
- MISO (Master In, Slave Out): This line is used for sending data from the slave to the master device
- SCK (Serial Clock): The master generates this clock signal to synchronize data transfer between the master and the slave
- CS (Chip Select): This signal is used by the master to select which slave device will participate in the communication at any given time

The protocol requires minimal hardware, making it an attractive option for embedded systems. Additionally, SPI supports communication with multiple slave devices by using separate CS lines for each slave (as shown in fig. 5), allowing the master to select which slave to communicate with during each data transfer cycle.

In 3D printers, SD cards are commonly used as non-volatile memory to store G-code files, which contain the instructions for the printer's movements and operations. These files can be large, making SD cards an ideal storage solution due to their ability to retain data without power. The SD card typically interfaces with the microcontroller using the SPI protocol or the SDIO native protocol (which will not be shown in this thesis). As shown in fig. 6, every SD card exposes up to 9 pins to directly communicate with other devices.

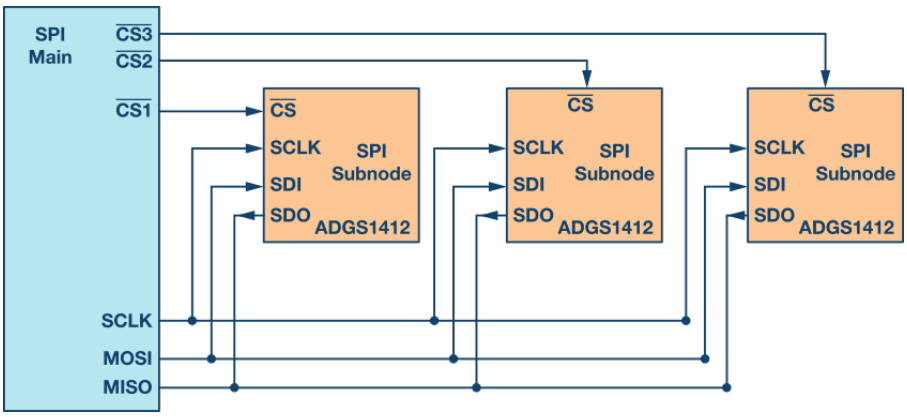


Figure 5: Multi-slave communication using SPI protocol

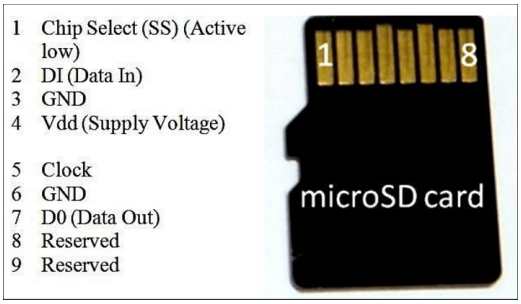


Figure 6: Pin-out of a micro SD-card

To interface an SD card with a microcontroller, it is typically paired with a breakout board. This board exposes the necessary pins that allow for easy connection to the GPIO pins of the microcontroller, as shown in fig. 7

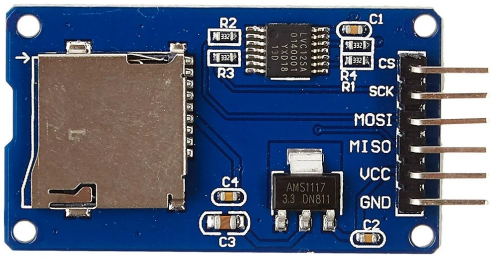


Figure 7: SD-card breakout board

2.1.6 DMA (Direct Memory Access)

Direct Memory Access (DMA) is a peripheral that allows data transfer between memory and other peripherals (or between two memory locations) without involving the CPU (as shown in fig. 8). This capability significantly improves efficiency and performance, especially in systems requiring high-speed or large-volume data movement. It is commonly used with peripherals like UART, SPI, ADC, DAC, and timers to handle data transfer without CPU intervention. In STM32, there are 2 DMA controllers with 14 channels each, which are dedicated to managing memory access requests from one or more peripherals. Each channel has an arbiter to handle

priority between DMA requests, and each peripheral can trigger the DMA for specific events, like a UART receiving data. To offload the CPU using the DMA, the procedure goes as follows:

1. The CPU configures the DMA controller to prepare it for a specific data transfer. This perhaps involves selecting the source and the destination address for the data, the transfer size, and the direction
2. The CPU configures the peripherals that want to offload and links it to a channel of the DMA, telling which event will trigger the DMA
3. Once triggered by an event, the DMA controller sends a handshake to the CPU, informing that it will handle the signal
4. If the CPU accepts the handshake, the DMA proceeds with the transfer.
5. The DMA signals the CPU using an interrupt in case of successful or erroneous transfer.
6. Finally, the CPU can process the transferred data (e.g. in case of an analog signal read from an ADC), clear and reset the DMA or reconfigure it to perform another transfer

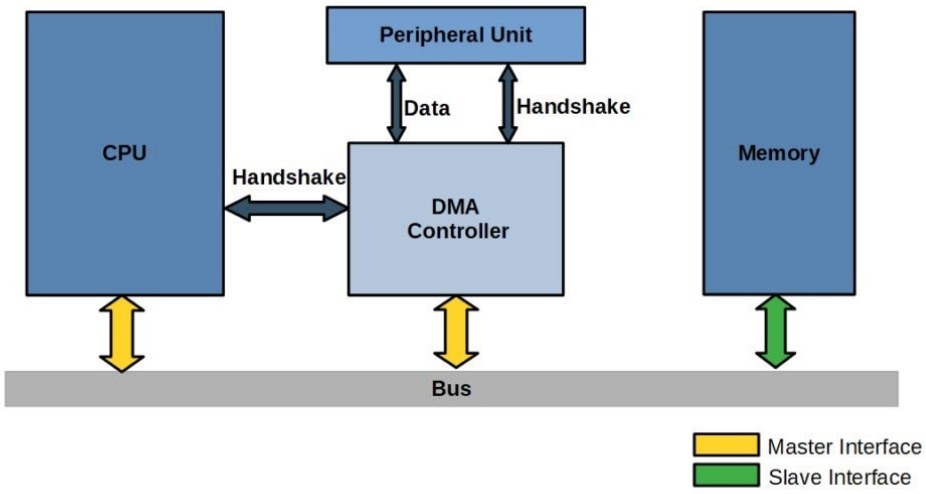


Figure 8: Schema of communication between CPU, memory and DMA

2.2 3D printer

After an overview of the most common peripherals that can be found inside an MCU, this section will show the reader which are the main physical components that compose a 3D printer and how they interact with the underlying hardware and the user:

- Structure: shows how a stepper motor, a hot-end, and a heated bed work, and how they can be controlled using hardware peripherals
- Language: shows how GCODE language is used to control a CNC machine along with a set of the most useful commands

2.2.1 Structure

A 3D printer (shown in fig. 9) works by building objects layer by layer using a process called additive manufacturing. It starts with a digital 3D model of the object, which is converted into instructions for the printer to follow by a software called “slicer”. The printer then deposits material, such as plastic, resin, or metal, in successive layers to create the physical object. Here’s a breakdown of its main components.

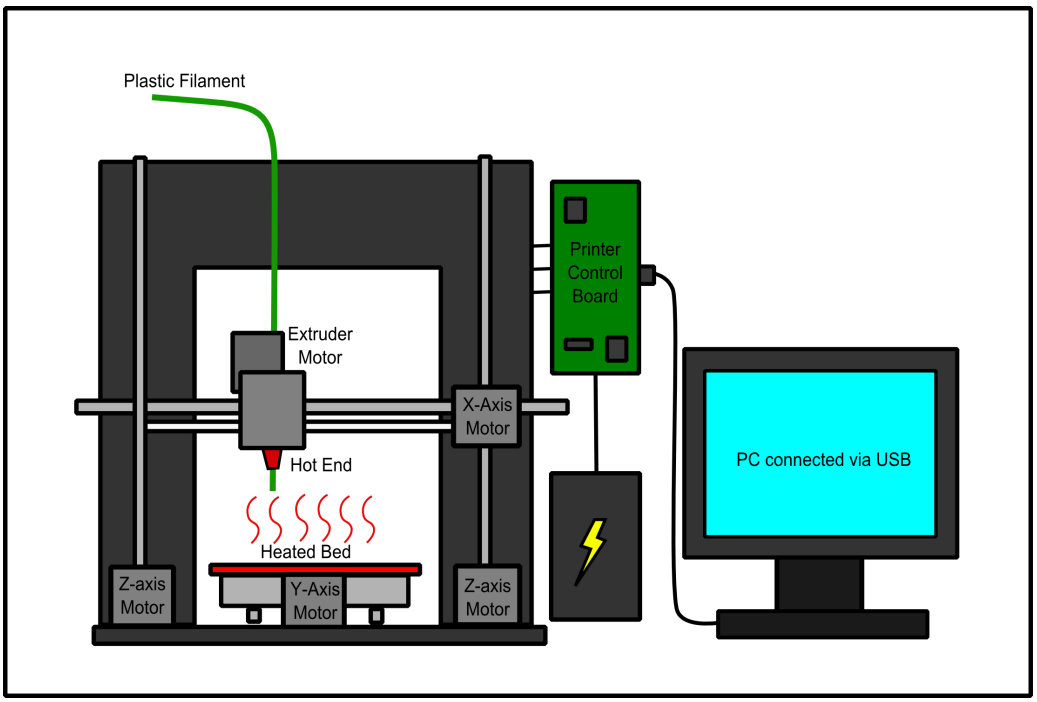


Figure 9: Drawing schema of 3D printer

Stepper motor A stepper motor is an electric motor whose main feature is that its shaft rotates by performing steps, that is, by moving by a fixed amount of degrees. This feature is obtained thanks to the internal structure of the motor and allows one to know the exact angular position of the shaft by simply counting how many steps have been performed, with no need for a sensor. Like all electric motors, stepper motors have a stationary part (the stator) and a moving part (the rotor). On the stator, there are teeth (grouped in pairs, one on the opposite side of another, composing a single phase) on which coils are wired, while the rotor is either a permanent magnet or a variable reluctance iron core (as shown in fig. 10). The basic working principle of the stepper motor is the following: by energizing one or more of the stator phases, a magnetic field is generated by the current flowing in the coil and the rotor aligns with this field. By supplying different phases in sequence, the rotor can be rotated by a specific amount to reach the desired final position

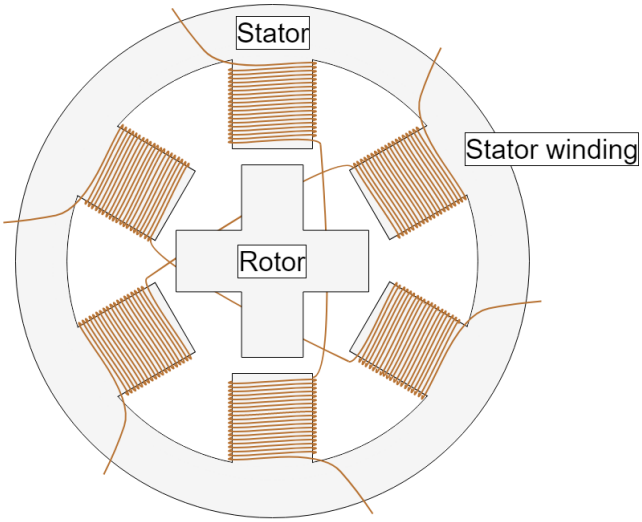


Figure 10: Section of a 3-phase stepper motor

Setting up a circuit and developing the firmware to directly manipulate the single coils could be cumbersome, due to the electrical noise and the working voltage of the steppers (usually between 12V and 24V, depending on the maximum reachable speed and the torque). That's why the main board often drives the stepper using an external stepper driver. This piece of hardware provides a chip that can commutate signals received

from 2 pins (one for the step, one for the direction) directly to pulses to the different phases of the stepper motor.

One of the most used stepper motor drivers for 3D printing is the A4988 (shown in fig. 11, initially manufactured by Polulu. On the right side of the chip, there are used to drive with the stepper. The A4988 supports 2-phase motors and provides the following pins:

- VMOT/GND: provides current to the stepper (up to 35V, 2A per phase)
- VDD/GND: provides current to the chip (3.3V to 5V)
- 1A/1B: output for the first phase of the motor
- 2A/2B: output for the second phase of the motor
- MS1, MS2, MS3: input for configuring the microstepping.

MS1	MS2	MS3	Microstep resolution
Low	Low	Low	Full step
High	Low	Low	Half step
Low	High	Low	Quarter step
High	High	Low	Eighth step
High	High	High	Sixteenth step

- ENABLE: enables/disables the motor
- RESET/SLEEP: TODO
- STEP: perform a step
- DIR: change the rotation direction. When high, the stepper rotates in a clockwise direction, otherwise in counter-clockwise

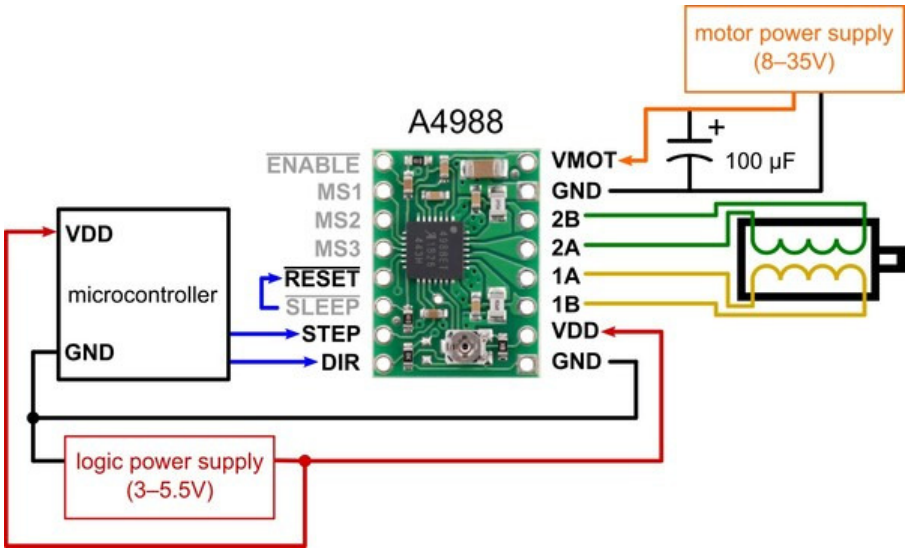


Figure 11: Wiring of an A4988 driver with a microcontroller and a stepper motor

To make the motor step, the STEP pin must be set to high and immediately low. The angular velocity of the stepper is dictated by the delay applied between 2 subsequent steps. The formula to compute the duration of the delay as a function of the RPM ω and the steps per revolution s of the motor is

$$\sigma(\omega, s) = \frac{\frac{1}{\frac{\omega}{60}}}{s}$$

A common stepper motor performs 200 steps per revolution. Given an angular velocity of 60 RPM (one revolution per second), the delay is:

$\omega = 60$

$s = 200$

$\sigma(60, 200) = \frac{\frac{1}{60}}{\frac{60}{200}} = 0.005s = 5ms$

However, in the 3D printing world, the speed (often called feed rate) is represented as mm/s, which dictates how much filament per second is extruded. The average feed rate for a normal model is between 50mm/s and 100mm/s. So how much delay has to be applied between steps to achieve these feed rates? In this case, the formula needs the distance covered for each step, which implies that the stepper has to be attached to something like a pulley or a rod. For a pulley of 10mm in diameter, the circumference is:

$C(d) = d \cdot \pi$

$d = 10mm$

$C(10) = 10 \cdot \pi = 31.4mm$

If the stepper performs 200 steps per revolution, each step equals to:

$s = 200$

$C(10) = 10 \cdot \pi = 31.4mm$

$d = \frac{31.4}{200} = 0.157mm$

This means that a step duration (or delay between steps) must be:

$d = 0.157mm$

$s = 50\frac{mm}{s}$

$t = \frac{d}{s} = \frac{0.157}{50} = 0.003s = 3ms$

$s = 100\frac{mm}{s}$

$t = \frac{d}{s} = \frac{0.157}{100} = 0.0015s = 1.5ms$

Thermistor The thermistor (shown in fig. 12) is a type of resistor whose resistance changes with the temperature change, and it's a combination of the words "Thermal" and "resistor". It is a passive component that does not require an extra power source to operate, they are inexpensive and accurate in measuring temperature, but they do not operate well in extreme conditions. A thermistor is made of semiconductor material whose resistance greatly depends on the surrounding temperature. If the resistance increases with the rising of the temperature, the thermistor is defined as PTC (Negative Temperature Coefficient), otherwise NTC (Positive Temperature Coefficient). This change in resistance can be calibrated and measured to calculate the exact temperature of the environment in a circuit. For accurate measurement, it is necessary to have maximum surface contact with the component or equipment, often using a special thermal paste.

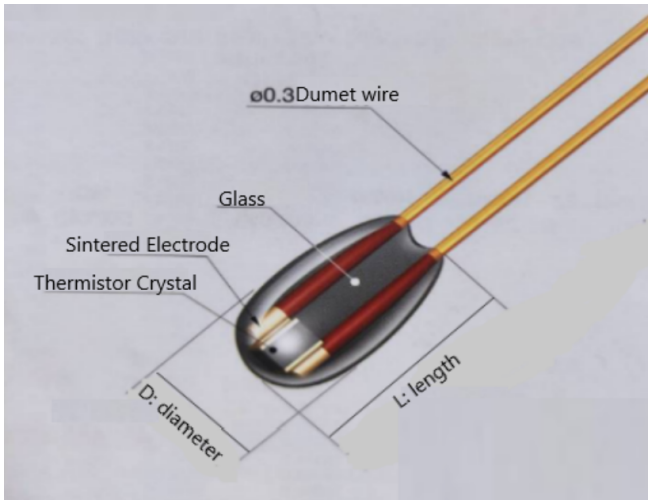


Figure 12: A generic thermistor

To determine the measured temperature from the instantaneous resistance of a thermistor, a direct resistance measurement by a microcontroller is not possible since microcontrollers cannot measure resistance directly. Instead, this challenge is addressed by leveraging the ADC (Analog-to-Digital Converter) peripheral to measure voltage changes using a voltage divider circuit (as shown in fig. 13), where the thermistor forms part of the divider. As the resistance of the thermistor changes with temperature, the voltage across it changes correspondingly. This voltage is then sampled by the ADC, allowing the microcontroller to compute the resistance indirectly and convert it into a temperature value through appropriate calculations.

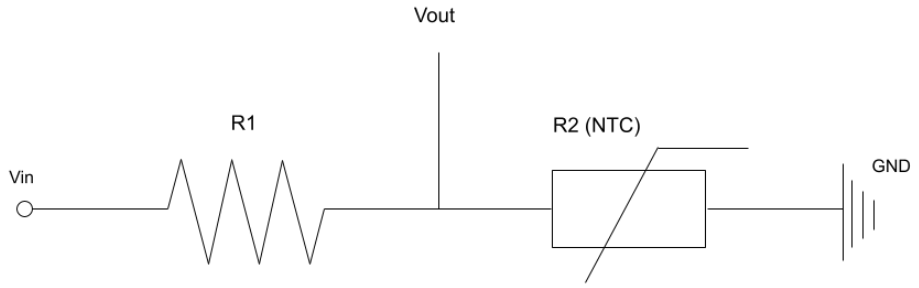


Figure 13: Circuit for voltage divider and thermistor

In the case of a voltage divider, the voltage between 2 resistances in series is:
$V_{out} = V_{in} \cdot \frac{R_2}{R_1+R_2}$
We need to derive R2 from the formula, which gives:
$R_2 = \frac{V_{out}*R_1}{Vin-Vout}$
To convert the resistance to an actual temperature, the simplified Stehinhart-Hart equation for NTC thermistors can be used:
$T = \frac{1}{\frac{1}{T_0}+\frac{1}{B}\cdot(ln(\frac{R_2}{R_1}))}$
where T is the temperature, T_0 is the reference temperature (room temperature, 25°C), B is the beta coefficient of the thermistor, R_1 is the reference resistance of the thermistor (the resistance at the reference temperature) and R_2 is current resistance of the thermistor.

Cartridge Heater A cartridge heater (shown in fig. 14) is a tube-shaped, industrial heating element that can provide localized and precise heating, and it is typically inserted inside a metal block. It consists of a resistance coil wound around a ceramic core which is surrounded by a dielectric material and encased in a metal sheath. Powered heat transferred through the coil to the sheath causes the sheath to heat up. This heat is then transferred to the inside metal part requiring heat.

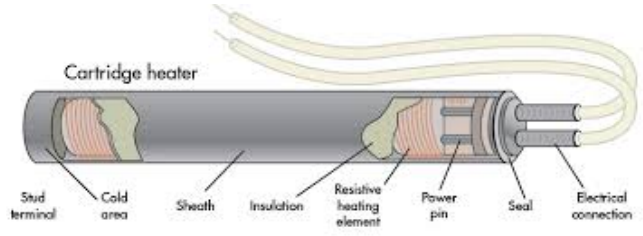


Figure 14: Circuit for voltage divider and thermistor

In order to control this device, as well as fans or other heating elements, a common solution is to utilize a MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor), (shown in fig. 15) which can act as an electronically controlled switch, allowing current to flow between its two main terminals, called source and drain, when a specific voltage is applied to its third terminal, the gate. A common configuration (as shown in fig. 16) is to connect a PWM-capable GPIO pin to the MOSFET, and the device that is being controlled is connected in

series with the source and drain terminals, typically with one terminal of the device linked to the power supply and the other to the MOSFET. When the PWM signal is applied to the gate, the MOSFET switches on and off rapidly, enabling or disabling the current flow through the device. The duty cycle of the PWM determines how much power is delivered to the device, effectively controlling its speed, intensity, or temperature. MOSFETs are well-suited for this purpose because they can handle high-current and high-voltage loads (ranging from 10V to over 50V or more) with minimal gate power requirements. Modern fast-switching MOSFETs can be driven with a low gate voltage, often below 2.5V, making them compatible with standard microcontroller outputs.

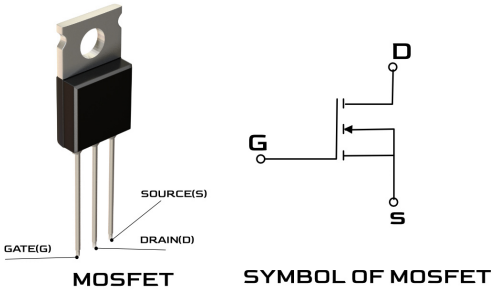


Figure 15: Structure of a generic MOSFET

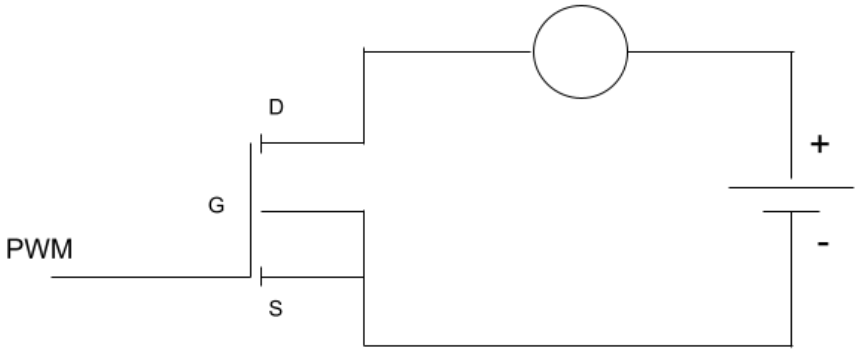


Figure 16: Circuit for PWM control using a MOSFET

Hot end The hot end is responsible for melting and extruding filament, and it is composed of several parts:

- Heat block: a block of metal that houses the heating cartridge and the thermistor. Often made of aluminum or copper for effective thermal conductivity. This block is heated by cartridge, maintaining a uniform temperature to melt the filament, while the thermistor monitors the current temperature
- Heat sink: a dissipator, used to dissipate the heat coming from the heat block
- Nozzle: the final exit point for the molten filament. It comes in various sizes (e.g., 0.4mm, 0.6mm) to control extrusion width and layer precision.

The filament is pushed by an extruder, typically a stepper motor, inside the tube. It is then melted by the heat block and it exits from the nozzle. In commercial 3D printers, the target temperature of the hot end stays between 180 and 220 degrees Celsius, depending on the type of filament melted.

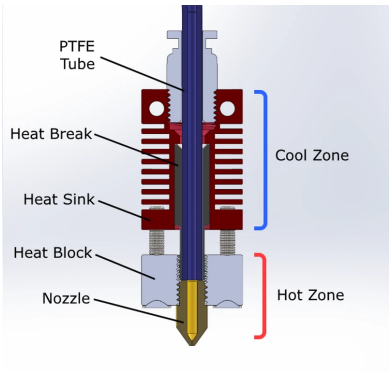


Figure 17: Cross-section of an hot-end

Heatbed bed The hotbed (or heated bed) ensures that prints adhere well to the build surface during the printing process and prevents issues like warping or detachment. It achieves this by maintaining a consistent temperature across its surface, often through an integrated PCB heater, composed of thin heating traces that are etched into a circuit board.

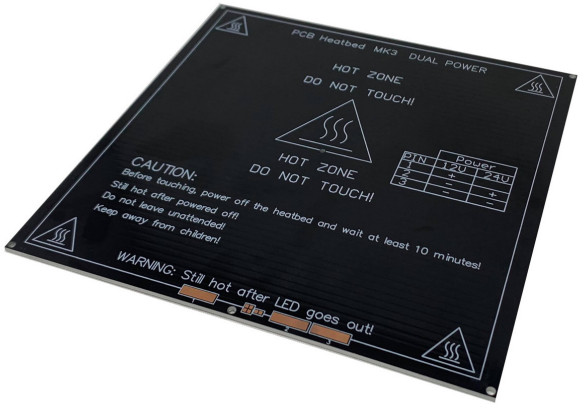


Figure 18: heated bed

2.2.2 Language

G-code (also known as RS-274) is the most widely used computer numerical control (CNC) and 3D printing programming language. It is used mainly in computer-aided manufacturing to control automated machine tools, as well as for 3D printer slicer applications. The G stands for geometry. G-code has many variants.

G-code instructions are provided to a machine controller (industrial computer) that tells the motors where to move, how fast to move, and what path to follow. Here's a snippet of some GCode for 3D printers.

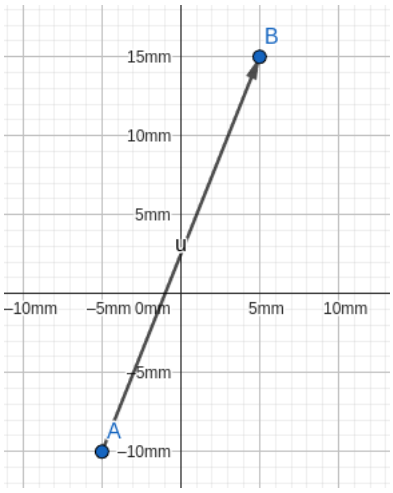
```
G92 E0 ; set extruder stepper to position 0
G28 ; auto home
G1 F1500 ; set feedrate to 1500 mm/s
G1 X2.0 Y2.0 F3000 ; move to (2.0, 2.0) with a feedrate of 3000 mm/s
G1 X3.0 Y3.0 ; move to (3.0, 3.0)
```

Regarding 3D printing, there's a list of the most commonly used commands where a 3D model has to be printed:

- G0 - Linear motion: tells the printing head (hot end) to move to a destination point. A G0 command is built as follows:

  ```
  G0 X2.0 Y2.0 Z0.2 F50
  ```

  where (X, Y, Z) is the destination point (it's not mandatory to include every axis, e.g. if we want only to move the head on the Z axis), in mm, and F is the feed rate in mm/s, which is the speed at which the motion will be performed.



- G1 - Linear motion with extrusion: tells the printing head to move to a destination point, while extruding. A G1 command is built as follows:

  ```
  G1 X2.0 Y2.0 E3.0 F50
  ```

  where (X,Y,Z) and F are the same as the G0 command, while E is the length of filament that will be extruded.

- G2/G3: Arc/circle move: tells the printing head to perform an arc/circle move. The difference between these two commands is that G2 performs a clockwise move, while G3 a counter-clockwise move. The G2/3 command has 2 different forms:

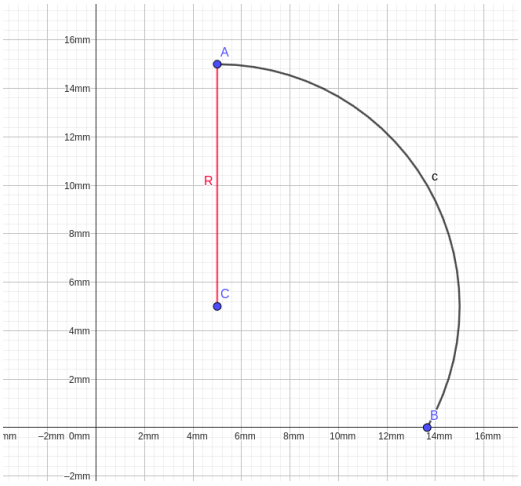
  - IJ form, where I and J and respectively the x and y offset from the center

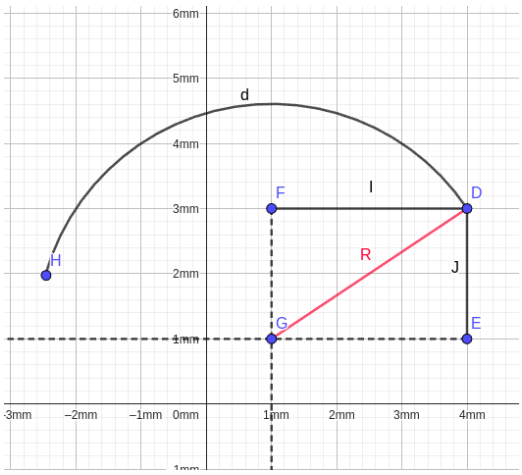
    ```
    G2 X125 Y32 I10.5 J10.5
    ```

  - R form, where R is the radius of the circle that contains the arc to draw.

    ```
    G2 X125 Y32 R3 F20.3
    ```

The arc is often approximated to a sequence of short straight lines.





- G10 - Retract: Retraction occurs when the printer's nozzle moves from one part of the model to another without extruding filament. The filament is pulled back slightly to relieve pressure in the nozzle, preventing excess material from oozing. Retraction helps avoid creating stringy residues or "spider webs" when the nozzle moves across open areas of the model, or between different layers or objects in the same bed. The retraction is performed in 2 simultaneous moves: move the z-axis and retract the filament with the extruder. Speed of motion, length of the retracted filament, and z lift are defined by another command. The G10 command is used alone:

  ```
  G10
  ```

- G11 - Recover: the recover command is used to return from a retract command. It is often used to restore the pressure on the nozzle to provide a correct printing afterwards. The G11 command is used alone, as G10:

  ```
  G11
  ```

- G90/91: set positioning to absolute or relative. Using absolute positioning, each move will use as the destination the absolute coordinate of the bed, while using relative the destination will be computed in relation with the current position.
- M20: list SD-card files
- M21: Init SD-card. This often initializes the communication with the SD card and simulates the file system mounting
- M22: Simulate ejection of the SD-card. This often unmounts the file system of the SD card
- M23: Select SD-card file.

  ```
  M23 mygcode.gcode
  ```

- M24: Start or resume SD print
- M25: Pause SD print
- M31: report print time
- M104: set hotend temperature

  ```
  M104 S210 ; set temperature to 210 units
  ```

- M105: report temperatures
- M106: set fan speed

```
M106 S215 ; set speed to 215/255
```

- M114: report print head position
- M140: set bed temperature

```
M140 S90 ; set temperature to 90 units
```

- M149: set temperature unit

```
M149 C ; set temperature unit to celsius
```

- M154: position auto-report

```
M154 S2; report position of the printing head every 2 seconds
```

- M155: temperature auto-report

```
M155 S2; report temperatures every 2 seconds
```

- M207: set firmware retraction settings
- M208: set firmware recover settings
- M220: set feed rate multiplier

```
M220 S80; set multiplier to 80%
```

- M524: abort sd print

2.3 Asynchronous programming in Rust

In concurrent programming, a system appears to perform multiple tasks simultaneously, although the exact mechanism depends on the platform. On general-purpose devices such as desktops or servers, concurrency is often achieved using threads. Each thread executes its sequential code and is scheduled by the operating system. On multi-core processors, threads can run on separate cores, enabling genuine parallel execution. However, in resource-constrained systems, typically operating on a single core, running a full operating system for thread management is impractical or inefficient. Here's a snippet of code that runs 2 threads that count up to 5 every second. The output can be seen in fig. 19

```rust
fn task(id: i64, sleep_time: std::time::Duration){
    for i in 0..5{
        std::thread::sleep(sleep_time);
        println!("[{}] {}", id, i);
    }
}

fn main() {
    let sleep_time = std::time::Duration::from_secs(1);
    let h1 = std::thread::spawn(move || task(0, sleep_time));
    let h2 = std::thread::spawn(move || task(1, sleep_time));
    h1.join().unwrap();
    h2.join().unwrap();
}
```

Asynchronous programming offers an alternative approach where concurrency is managed entirely within the program without relying on the operating system. In Rust, this is enabled by an async runtime, which manages tasks defined by the programmer. These tasks yield control explicitly using the `await` keyword.

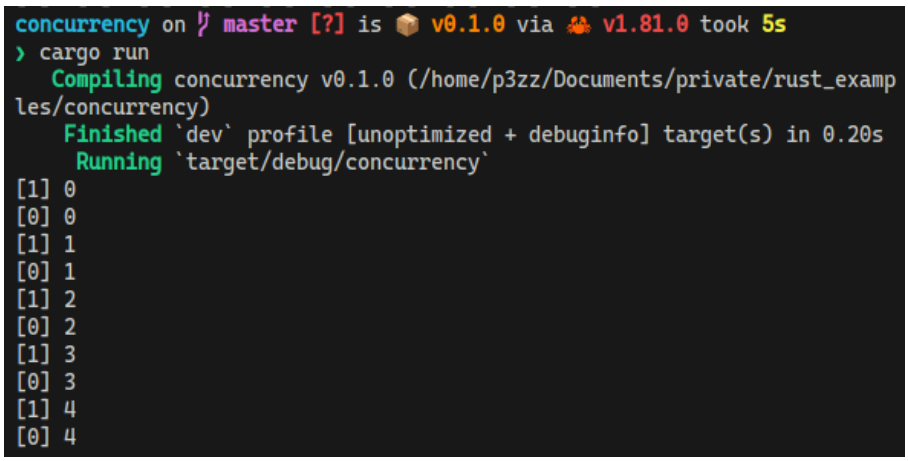


Figure 19: Output of 2 functions that count up to 5, using OS threads

Crates such as **tokio**[28] (for systems using the standard library) **embassy**[3] (for bare-metal, no-std environments) provide lightweight runtimes that include task executors, which handle the scheduling and management of asynchronous tasks, allowing smooth and efficient concurrency even on systems with minimal resources. Without operating system involvement, asynchronous task switching is extremely fast, as there's no need for the heavy context switches required by threads. Async tasks consume significantly less memory than traditional threads since they do not require storage for the entire thread state during scheduling. This efficiency makes asynchronous programming particularly suitable for scenarios with many concurrent tasks where most time is spent waiting—for instance, waiting for I/O operations or external responses. Here's a snippet of code that does the same thing as the one in fig. 19, but using the executor provided by the **tokio**[28] crate, and without using OS threads. The output can be seen in fig. 20

```rust
async fn async_task(id: i64, sleep_time: std::time::Duration){
    for i in 0..5{
        tokio::time::sleep(sleep_time).await;
        println!("[{}] {}", id, i);
    }
}

#[tokio::main]
async fn main() {
    let sleep_time = std::time::Duration::from_secs(1);
    tokio::join!(
        async_task(0, sleep_time),
        async_task(1, sleep_time)
    );
}
```

As we can see, the same result has been achieved in 2 different types of concurrency. At the core of asynchronous programming in Rust lies the `Future` trait. A `Future` represents an asynchronous computation that eventually produces a result, which can even be an empty value (e.g., `()`).

```rust
trait Future {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}
enum Poll<T> {
    Ready(T),
```

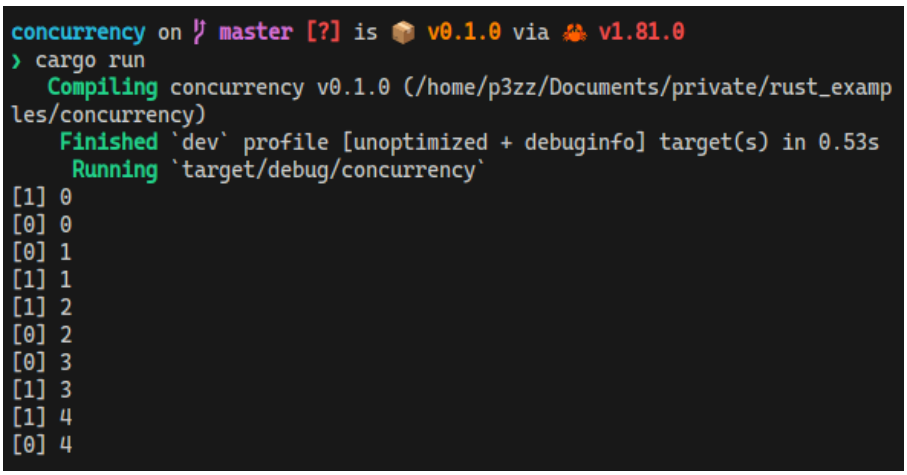


Figure 20: Output of 2 functions that count up to 5, using **tokio**[28] tasks

```
    Pending,
}
```

"Futures can be advanced by calling the `poll` *function, which will drive the future as far toward completion as possible. If the future completes, it returns* `Poll::Ready(result)`*, otherwise if is not able to complete yet, it returns* `Poll::Pending` *and arranges for the* `wake()` *function to be called when the Future is ready to make more progress. When* `wake()` *is called, the executor driving the Future will call* `poll` *again so that the Future can make more progress. Without* `wake()`*, the executor would have no way of knowing when a particular future could make progress and would have to be constantly polling every future. With* `wake()`*, the executor knows exactly which futures are ready to be polled. This is the reason why futures in Rust are defined as "lazy": they don't make progress unless someone polls it."*[24]

2.4 The Embassy framework

"Embassy is a Rust framework designed specifically for embedded systems to enable efficient, asynchronous programming. It provides a modern, non-blocking approach to embedded development, leveraging Rust's powerful type system and memory safety guarantees. Embassy is tailored for environments where resources are constrained and an operating system is either unavailable or unsuitable. Key features of Embassy include its lightweight async runtime and HAL (Hardware Abstraction Layer) implementations that allow for fine-grained control over hardware peripherals while maintaining high efficiency. Embassy's runtime is optimized for real-time constraints, ensuring predictable performance and minimal overhead, and it supports common embedded platforms like STM32 and nRF devices, making it versatile across different hardware ecosystems. Furthermore, DMA and interrupts are widely used by the embassy framework, but they're practically transparent to the developer because everything is configured under the hood."[3]

The Embassy[3] project consists of several crates that the user can use together or independently, such as:

- embassy-executor[4]: asynchronous executor with static task allocation, used by Xtrooder to seamlessly orchestrate the different tasks that drive the components of the printer
- embassy-stm32/nrf/rp[6]: synchronous/asynchronous HALs implementation, used by Xtrooder to abstract peripherals

2.4.1 Executor

"The `embassy-executor`*[4] is an* **`async`**`/`**`await`** *executor that generally executes a fixed number of tasks, often allocated at startup. The executor may also provide a system timer that the user can use for both async and blocking delays. The executor keeps a queue of tasks that it should poll. When a task is created, it is polled, so the task will attempt to make progress until it reaches a point where it would be blocked. This may happen whenever a task is .*`await`*'ing an* **`async`** *function. When that happens, the task yields*

execution by returning `Poll::Pending`*. Once a task yields, the executor enqueues the task at the end of the run queue and proceeds to* `poll` *the next task in the queue. When a task is finished or canceled, it will not be enqueued again."*[3] The executor is informed about the different functions that has to handle through the `#[embassy_executor::task]`, which represents a task that needs to be scheduled, and the `#[embassy_executor::main]`, which tells the compiler which is the main entry point of the program. The attribute gives the main function a **spawner** parameter that is used to spawn different tasks. If you use the `#[embassy_executor::main]` macro in your application, it creates the Executor for you and spawns the main entry point as the first task. You can also create the Executor manually, and you can create multiple `Executors`. Here's a snippet of code that creates a task that makes an LED blink, and spawns it using the spawner provided by the `embassy-executor`[4] crate.

```
#[embassy_executor::task]
async fn blinker(mut led: Output<'static>, interval: Duration) {
    loop {
        led.set_high();
        Timer::after(interval).await;
        led.set_low();
        Timer::after(interval).await;
    }
}
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_nrf::init(Default::default());

    let led = Output::new(p.P0_13, Level::Low, OutputDrive::Standard);
    unwrap!(spawner.spawn(blinker(led, Duration::from_millis(300))));
}
```

2.4.2 HALs (Hardware Abstraction Layers)

"HALs implement safe Rust API which lets the user use peripherals such as USART, UART, I2C, SPI, CAN, and USB without the need to directly manipulate low-level registers. Embassy provides implementations of both async and blocking APIs where it makes sense. DMA (Direct Memory Access) is an example where ***`async`*** *is a good fit, whereas GPIO states are a better fit for a blocking API."*[3] The Embassy project maintains HALs for the following MCU families:

- embassy-stm32, for all STM32 microcontroller families
- embassy-nrf, for the Nordic Semiconductor nRF52, nRF53, nRF91 series
- embassy-rp, for the Raspberry Pi RP2040 microcontroller
- esp-rs, for the Espressif Systems ESP32 series of chips
- ch32-hal, for the WCH 32-bit RISC-V(CH32V) series of chips

Here's an example of the UART HAL provided by the Embassy framework.

```
use defmt::*;
use embassy_executor::Spawner;
use embassy_stm32::usart::{Config, Uart};
use embassy_stm32::{bind_interrupts, peripherals, usart};
use heapless::String;
use static_cell::StaticCell;
use {defmt_rtt as _, panic_probe as _};
```

```rust
bind_interrupts!(struct Irqs {
    UART7 => usart::InterruptHandler<peripherals::UART7>;
});

#[embassy_executor::task]
async fn main_task() {
    let p = embassy_stm32::init(Default::default());

    let config = Config::default();
    let mut usart = Uart::new(
        p.UART7,
        p.PF6,
        p.PF7,
        Irqs,
        p.DMA1_CH0,
        p.DMA1_CH1,
        config
    ).unwrap();

    for n in 0u32.. {
        let mut s: String<128> = String::new();
        core::write!(&mut s, "Hello DMA World {}!\r\n", n).unwrap();

        usart.write(s.as_bytes()).await.ok();

        info!("wrote DMA");
    }
}

#[embassy_executor::main]
async fn main(spawner: Spawner) -> ! {
    info!("Hello World!");

    spawner.spawn(spawner.spawn(main_task()));
}
```

3 Software development

The software that controls a 3D printer has a few challenges to overcome:

- Motion planning: each movement the steppers need to perform hides a non-trivial computation under the hood. For instance, a simple linear motion from point A to point B is computed as a sequence of steps, delayed by a precise timing which is derived from the speed of the movement and the distance covered by one single step. Furthermore, if we include the micro-stepping, the out-of-bounds check, and the acceleration, the computation gets even worse. Considering that most of the time this needs to be computed for 3 steppers or more steppers (e.g. X, Y, Z, and E steppers), the algorithm must be efficient.

- Control and temperature monitoring of the hot end and the heated bed: to obtain precise and smooth printing, the hot end needs to keep its temperature as close as possible to the optimal fusion temperature of the material of the filament, while the heated bed is maintaining a temperature such that the filament adheres correctly to the surface and doesn't warp. This is often performed using a PWM and a closed feedback loop like PID, where the duty cycle is computed as the output of the PID computation, while

the frequency is fixed (about 1KHz). Due to thermal mass, causing delays in temperature changes, fine-tuning of the PID parameters is mandatory.

- GCode parsing: each GCode message has to be properly parsed and mapped to the correct operation to perform

Furthermore, writing Rust code strictly related to an embedded platform (e.g. STM32) removes the possibility of using the standard crate provided by the Rust environment by default. The following sub-sections will show how Xtrooder copes with these deficiencies and how it solves the challenges mentioned above.

3.1 Project architecture

The project contains 2 separate workspaces:

- **host**(section 3.1.1): platform-agnostic workspace, which contains abstractions of the main components of a 3D printer, such as stepper motors and thermistors, mathematical operators and utils
- **board**(section 3.1.2): board-related workspace, which contains the main script of the project along with different structs and implementations of the traits contained in section 3.1.1

3.1.1 Host

This workspace contains everything that can be compiled with a stable toolchain on every major target, such as Linux, Windows, and MacOS, and it leverages the **core** crate, which provides several functionalities that are present in the standard crate, but that can be compiled in a no-std environment. Every ”lib.rs“ of each crate starts with this configuration conditional check:

```
#![cfg_attr(not(test), no_std)]
```

This means that, if we are in a module of the crate configured with:

```
#[cfg(test)]
```

we can leverage the standard crate of Rust to perform testing on our crate without the restriction of a no-std environment. On the other side, the remaining modules of the crate can be used in a no-std environment without worries of failed compilation.

Dependencies Here's a list of useful dependencies that have helped develop the core crates:

- **heapless**: *”the crate offers implementations of common standard library structures such as* `String`*,* `Vec`*, and* `Queue`*, built on static memory allocation. These structures require you to specify their capacity explicitly during construction. Since they operate with a fixed capacity,* `heapless[10]` *data structures do not perform implicit reallocations. This design ensures that operations like* `heapless::Vec.push` *execute in true constant time, unlike dynamically allocated counterparts, which provide amortized constant time but may experience unpredictable worst-case execution times due to reallocation. This predictability makes heapless[10] structures ideal for hard real-time systems, where unbounded execution times are unacceptable. Additionally,* `heapless[10]` *structures do not rely on a memory allocator, eliminating the risk of encountering uncatchable Out-of-Memory (OOM) errors during operation. While capacity limitations may still occur, the APIs handle such cases by returning a Result, enabling developers to gracefully manage scenarios where a structure reaches its capacity.”*[10] Here's an example of the usage of `Vec` from the standard crate:

```
let mut v = std::vec::Vec::<usize>::new();
v.push(7);
v.push(8);
v.push(9);
```

In this case, the whole vector is allocated in the heap memory, and in case of OOM condition, the method will panic. Furthermore, it requires a dynamic memory allocator, that is often present in classic OS, but not in microcontrollers. Let's now see an example of `Vec` from the `heapless`[10] crate:

```
let mut s = heapless::Vec::<usize, 2>::new();
s.push(7).unwrap(); // ok
s.push(8).unwrap(); // ok
s.push(9).unwrap(); // here it fails
```

In this case, we need to specify the capacity of the `Vec`, and each time we push something, the method returns a `Result<(), T>`, `Ok(())` if the `push` has been successful, or `Err<T>` if the vector is full. This gives the caller the chance to handle the buffer overflow error. This pattern is also used for the other provided structures.

- **micromath**: *"an embedded-friendly math library offering fast and safe floating-point approximations for common arithmetic operations. It includes support for 2D and 3D vector types, statistical analysis functions, and quaternions. The library approximates many* ***`f32`*** *arithmetic operations using bitwise techniques, delivering excellent performance and small code size at the expense of precision. For applications such as graphics and signal processing, this trade-off is often acceptable, as the performance benefits outweigh the precision loss. Floating-point approximations are accessible through the `F32Ext` trait, which is implemented for* ***`f32`****. This trait provides a drop-in, standard-compatible API for seamless integration."*[18] In order to provide the **`f32`** methods in a no-std environment, the module that needs to import the trait as follows::

  ```
  use micromath::F32Ext;
  ```

  This enables the user to call the most common mathematical functions on floating point like in a standard environment:

  ```
  let f1 = 1.0f32;
  let f2 = 3.0f32;
  let res = f1.atan2(f2);
  ```

- **measurements**: *"a versatile library designed to represent physical quantities like length, mass, pressure, and more. It provides a robust set of functions for converting between common units, ensuring precise and consistent calculations. Beyond unit conversion, the crate supports arithmetic operations, allowing you to combine quantities in meaningful ways—such as calculating distance by multiplying speed and time."*[17]

  The Problem with Unconstrained Arithmetic Consider the following example, where a simple calculation determines the distance covered given a speed and a duration: Here's an example of computing the distance covered going to a specific speed for a given time

  ```
  let speed = 10.2;
  let duration = 3.1;
  let distance = speed * duration;
  ```

  This code appears straightforward, but it introduces a significant risk: without clear unit definitions, it's nearly impossible to ensure the correctness of the computation. A mistake in the units or their interpretation can lead to hard-to-detect bugs with potentially severe consequences.

  Adding Type-Safety with measurements Using the measurements crate adds a layer of type safety, ensuring clarity and precision in such calculations. Here's an attempt using explicit unit conversions:

  ```
  let speed = measurements::Speed::from_meters_per_second(300f64);
  let duration = core::time::Duration::from_secs(2);
  let distance = measurements::Distance::from_meters(
  ```

```
        speed.as_meters_per_second() * duration.as_millis()
    );
```

While this code is safer than the previous example, it suffers from verbosity and remains prone to subtle errors. For instance, the developer might use an incorrect conversion method (e.g., `duration.as_secs()` instead of `duration.as_millis()`), introducing a bug that, while easier to catch than in the first example, could still lead to inaccuracies.

```
let speed = measurements::Speed::from_meters_per_second(300f64);
let duration = core::time::Duration::from_secs(2);
let distance: measurements::Distance = speed * duration;
```

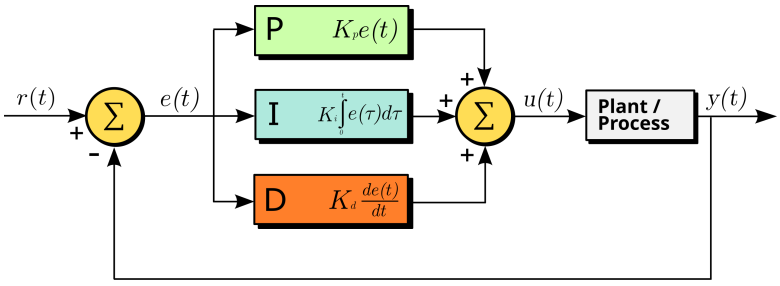
This version is both concise and robust. The crate handles unit conversions internally, ensuring correctness while maintaining readability. The ability to perform operations directly on domain-specific types enhances the reliability of the code base, significantly reducing the potential for bugs.

Crates

- **math**: provides bridges between various dependencies while also offering wrappers and a foundational implementation for 2D and 3D vector structures.

vector The vector module contains the implementation of 2D/3D vector which is designed to operate generically over any type that implements the `Measurement` trait from the `measurements`[17] crate. By leveraging the `core::ops::Add` and `core::ops::Sub` traits, these vectors seamlessly support fundamental arithmetic operations, enabling clean and intuitive manipulation of vector quantities. Additionally, the crate includes essential vector operations like magnitude, normalization, and dot products, further extending its utility in mathematical computations.

pid The PID controller is a widely used feedback control system that continuously corrects an output using 3 coefficients (K_p, K_i and K_d) to minimize the error against a target output.



The output is computed using this formula:

$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$

The PID module contains an implementation of a simple PID controller as a struct. The implementation provides setters and getters for target and output bounds, as well as an `fn update(...)` method that computes the new output using discrete steps. The output method also prevents integral windup, restricting the output between the specified bounds.

```
pub struct PID {
    kp: f64,
    ki: f64,
    kd: f64,
    target: Option<f64>,
    prev_error: f64,
    integral: f64,
```

```
    bounds: Option<(f64, f64)>,
}
```

- **parser**: introduces the `GCommand` enumeration as an abstraction for representing GCode commands. This enumeration harnesses Rust's powerful enumeration system, allowing it to encapsulate multiple variants with distinct types

```
#[derive(PartialEq, Debug, Clone)]
pub enum GCommand {
G0 {
    x: Option<measurements::Distance>,
    y: Option<measurements::Distance>,
    z: Option<measurements::Distance>,
    f: Option<measurements::Speed>,
},
...
G4 {
    p: Option<core::time::Duration>,
    s: Option<core::time::Duration>,
},
G10,
M23 {
    filename: heapless::String<12>,
},
M104 {
    s: measurements::Temperature,
},
```

The core of the parsing functionality is encapsulated within the `GCodeParser` struct. The parsing process unfolds in two stages. First, it parses an entire line, which may include comments. Comments typically begin with a ';' or '(', and end with an ')' or a ';', or nothing. Upon detecting a command, the parser extracts each option based on the specific command code. These values can represent speeds, distances, temperatures, or durations. Since each `GCommand` variant contains structured data, the parser must maintain a state to track the measurement units for distances and temperatures. These units are defined within a couple of enumerations, ensuring consistency and clarity.

```
#[derive(Clone, Copy, PartialEq, Debug)]
pub enum TemperatureUnit {
    Celsius,
    Farhenheit,
    Kelvin,
}

#[derive(Clone, Copy, PartialEq, Debug)]
pub enum DistanceUnit {
    Millimeter,
    Inch,
}

#[derive(Clone, Copy)]
enum ParserState {
    ReadingCommand,
    ReadingComment(char),
```

```
}

pub struct GCodeParser {
    distance_unit: DistanceUnit,
    temperature_unit: TemperatureUnit,
}
```

The command, represented as a `&str`, is first split into tokens. Each token is then further divided into key-value pairs, which are stored in a `LinearMap` structure provided by the **heapless**[10] crate. Finally, the parser matches the command code and interprets the associated values as the appropriate types based on the context of the command.

In this case, a G4 command (dwell) has been found, and the respective options have been retrieved. An extractor function is built as follows:

```
fn extract_duration(
    cmd: &LinearMap<&str, &str, 16>,
    key: &str,
    unit: DurationUnit,
) -> Option<Duration> {
    let value = extract_token_as_number(cmd, key)?;
    match unit {
        DurationUnit::Second =>
            Some(Duration::from_secs_f64(value)),
        DurationUnit::Millisecond =>
            Some(Duration::from_secs_f64(value / 1000f64)),
    }
}
```

- **common**: This module defines traits that are used to abstract the main peripherals. By encapsulating hardware-specific behavior, these traits enable the design of generic components for a 3D printer. This flexibility facilitates rapid on-host testing without depending on actual embedded hardware. These core traits serve as the backbone for other crates, such as those implementing the thermal actuator, stepper motor, and fan control. For instance, the abstraction of GPIO output pins allows for hardware-agnostic design while maintaining functionality. Consider the example of a stepper motor struct:

```
pub struct Stepper<P: OutputPinBase, ...> {
...
step: P,
dir: P,
...
```

The stepper is built on top of a generic type which represents an output pin, which needs to implement the `OutputPinBase`:

```
pub trait OutputPinBase {
    fn set_high(&mut self);
    fn set_low(&mut self);
    fn is_high(&self) -> bool;
}
```

Another core trait is `ExtiInputPinBase`, which abstracts a pin that is attached to an interrupt line. This enables the pin to listen for changes in its state and signal it to the DMA through an interrupt.

```
pub trait ExtiInputPinBase {
    fn is_high(&self) -> bool;
```

```
    fn wait_for_high(&mut self) -> impl Future<Output = ()>;
    fn wait_for_low(&mut self) -> impl Future<Output = ()>;
}
```

The `TimerBase` trait provides a way to abstract an asynchronous timer. This is derived from the `embassy_time::Timer` struct.

```
pub trait TimerBase {
    fn after(duration: Duration) -> impl Future<Output = ()>;
}
```

This trait is used, for example, to perform a stepper movement.

The `PwmBase` and `AdcBase` traits serve as essential abstractions for interfacing with hardware peripherals like PWM and ADC. These traits were derived from the `SimplePwm` and `Adc` structs of the embassy-stm32[6] crate, respectively. Only the methods required for the implementation of specific components — `PwmBase` for the heater and `AdcBase` for the thermistor — were extracted, ensuring a focused and efficient design tailored to their functionality.

```
pub trait PwmBase {
    type Channel: Copy + Clone;

    fn enable(&mut self, channel: Self::Channel);
    fn disable(&mut self, channel: Self::Channel);
    fn get_max_duty(&self) -> u64;
    fn set_duty(
        &mut self,
        channel: Self::Channel,
        duty_cycle: u64
    );
}

pub trait AdcBase {
    type PinType;
    type SampleTime: Copy + Clone;
    type Resolution: Copy + Clone + Into<u64>;

    fn set_sample_time(
        &mut self,
        sample_time: Self::SampleTime
    );
    fn sample_time(&self) -> Self::SampleTime;
    fn set_resolution(
        &mut self,
        resolution: Self::Resolution
    );
    fn resolution(&self) -> Self::Resolution;
    fn read(
        &mut self,
        pin: &mut Self::PinType,
        readings: &mut [u16]
    ) -> impl Future<Output = ()>;
}
```

These 2 traits have been respectively used in the `Thermistor` and `Heater` structs implementation:

```
pub struct Thermistor<'a, A: AdcBase> {
    read_pin: A::PinType,
    readings: &'a mut DmaBufType,
    config: ThermistorConfig,
}

pub struct Heater<P: PwmBase> {
    ch: P::Channel,
    pid: PID,
}
```

- **stepper**: defines the primitives for motion and a versatile abstraction over a stepper motor, leveraging generics and traits to allow customization of attachment modes and GPIO pin handling. Here's an overview of its components and functionality.

Stepper The `Stepper` struct has a generic type `M` that enables it to mark it as attached or not attached. This is achieved using the `AttachmentMode` trait, which allows the `Stepper` struct to enforce compile-time constraints based on attachment state using generics. Any struct that implements it can be used as a marker to specialize the stepper with a different working mode.

```
pub struct NotAttached {}
pub struct Attached {}

pub trait AttachmentMode {}

impl AttachmentMode for Attached {}
impl AttachmentMode for NotAttached {}


pub struct Stepper<P: OutputPinBase, M: AttachmentMode> {
    step: P,
    dir: P,
    options: StepperOptions,
    attachment: Option<StepperAttachment>,
    step_duration: Duration,
    steps: f64,
    _attachment_mode: PhantomData<M>,
}
```

`NotAttached` and `Attached` are marker structs that represent whether the stepper motor is attached to additional instrumentation (pulley or rod) or not. Furthermore, it has another generic type `P`, which has to implement the `OutputPinBase` trait, as seen in the **common** crate. The physical motor stepper does not come with an embedded encoder to remember the number of steps performed, so the **steps** member is used to accumulate them. A stepper is provided with different options:

```
#[derive(Clone, Copy, PartialEq, Debug)]
pub enum RotationDirection {
    Clockwise,
    CounterClockwise,
}
```

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
pub enum SteppingMode {
    FullStep,
    HalfStep,
    QuarterStep,
    EighthStep,
    SixteenthStep,
}

#[derive(Clone, Copy)]
pub struct StepperOptions {
    pub steps_per_revolution: u64,
    pub stepping_mode: SteppingMode,
    pub bounds: Option<(f64, f64)>,
    pub positive_direction: RotationDirection,
}
```

These options are used to perform correct computations. A stepper is defined as "Attached" if it is attached to a pulley or a threaded rod. At this point, at each step, something will be moved for a given distance. This unlocks the possibility to move the stepper using a Distance, that can be a destination of a distance to cover.

```rust
#[derive(Clone, Copy)]
pub struct StepperAttachment {
    pub distance_per_step: Distance,
}


impl<P: OutputPinBase> Stepper<P, Attached> {
    ...

    pub async fn move_for_distance<T: TimerTrait>(
        &mut self,
        distance: Distance,
    ) -> Result<Duration, StepperError>;

    pub async fn move_to_destination<T: TimerTrait>(
        &mut self,
        destination: Distance,
    ) -> Result<Duration, StepperError>;
}
```

Motion it deals with trajectory computation to drive the stepper motors. This provides straight and arc trajectories and ensures safety in every movement and efficient computation. The basis function that makes every movement work is the `fn linear_move_to(...)`:

This function takes a stepper, a destination, a speed, and an optional end-stop to listen to. It sets the speed of the stepper and makes it move to the given destination. If an end-stop is provided, the two futures are run simultaneously using the `select!` macro of the futures crate. This macro runs futures as soon as one of those finishes the execution. By doing that, if the end-stop is triggered before the movement is finished, the function safely exits with an error of end-stop hit. Otherwise, the stepper can finish the movement and reach the destination. The basis for 2D movements is `linear_move_to_2d_raw(...)`: The function takes a pair of steppers, a pair of end-stops, a destination (as a 2D vector), a speed (as a 2D vector), and runs the `linear_move_to` function for both the steppers and their associated end-stops using the join! macro from the `futures` crate, too. This macro runs simultaneously an arbitrary number

of futures and returns as soon as every future has been completed. In this case, the duration of the move is returned, specifically the duration of the longest movement.

A linear move command however, as seen in the background section, comes with a 1D speed (or feed rate), which tells at which speed the printer head has to reach a destination point. Knowing that the steppers need to move "simultaneously" to achieve smooth printing, we need to compute the speed at which each stepper needs to travel in order to achieve the requested feed rate. For the sake of brevity, let's consider a linear movement in which all 4 steppers (X, Y, Z and the extruder E) need to travel the print head to a destination point D for a point S at a feed rate F.

$$
\begin{aligned}
&f_e = F \\
&S_{xyz} = (S_x, S_y, S_z) \\
&D_{xyz} = (D_x, D_y, D_z) \\
&\delta xyz = D_{xyz} - S_{xyz} \\
&l = \sqrt{\delta_x^2 + \delta_y^2 + \delta_z^2} \\
&\delta_{xyz_{norm}} = \frac{\delta_{xyz}}{l} \\
&f_{xyz} = \delta_{xyz_{norm}} \cdot F
\end{aligned}
$$

For an arc movement, given the starting point S, the center of the circle C, the length of the arc l_{arc} and the rotation direction of the movement, we have:

$$
\begin{aligned}
&S_{xy} = (S_x, S_y) \\
&C_{xy} = (C_x, C_y) \\
&\delta_{xy} = S_{xy} - C_{xy} \\
&R = \sqrt{\delta_x^2 + \delta_x^2} \\
&\alpha = \frac{l_{arc} \cdot d}{R} \\
&D_x = C_x + \delta_x * cos(\alpha) - \delta_y * sin(\alpha) \\
&D_y = C_y + \delta_x * sin(\alpha) - \delta_y * cos(\alpha)
\end{aligned}
$$

The function that performs this computation is called `fn compute_arc_destination(...)`, which is then used inside `fn arc_move_2d_arc_length(...)`. The function divides the arc length is smaller parts using a unit length. After this, a for-loop iteratively computes the new destination for the current arc and moves the stepper to the destination using a linear motion. By doing this, the whole arc is approximated as a sequence of straight lines.

- **thermal_actuator**: it introduces a foundational struct designed to represent a thermal control system, combining two key components: a heater, responsible for heating a given surface, and a thermistor, which serves as a temperature sensor. These components operate in unison under the guidance of a PID controller, ensuring the system maintains a stable target temperature with high accuracy. The control process is executed in discrete intervals, each lasting a defined duration (**dt**), allowing for consistent and predictable adjustments to the thermal output.

Heater

```
pub struct Heater<P: PwmBase> {
    ch: P::Channel,
    pid: Controller,
}
```

The Heater struct is designed to avoid directly storing pin instances as its members, retaining only the channel it operates on. This approach reflects the nature of embedded systems, where PWM functionality is typically a feature of a timer, offering a fixed number of channels (e.g., 4 channels on an STM Nucleo board). In the Embassy[3] framework, PWM is abstracted using the `SimplePwm` struct, which takes ownership of the pins passed during its creation. Consequently, once the `SimplePwm` instance is initialized, those pins become inaccessible to other parts of the code. To address this limitation, any

structure, such as the Heater, that requires interaction with a specific PWM channel only stores the channel identifier. When operations on the associated pin are needed, the structure relies on the `SimplePwm` instance, which is passed as a parameter to its methods.

Thermistor The `Thermistor` struct models the thermistor itself and its associated ADC type. It contains a pin for reading data, a mutable buffer for storing ADC readings, and a `ThermistorConfig` instance to define its operating parameters. This struct provides functionality for asynchronously reading and calculating the temperature using the `fn read_temperature(...)` method, which collects multiple ADC readings based on the configured number of samples, computes their average, and uses the result along with the thermistor configuration to determine the read temperature. This struct differs from the Heater since the pin that is interacting with is owned, while the ADC struct is passed as a parameter when the temperature reading is performed.

```rust
#[derive(Clone, Copy)]
pub struct ThermistorConfig {
    pub r_series: Resistance,
    pub r0: Resistance,
    pub b: Temperature,
    pub samples: u64,
}

pub struct Thermistor<'a, A: AdcBase> {
    read_pin: A::PinType,
    readings: &'a mut DmaBufType,
    config: ThermistorConfig,
}
```

Thermal actuator The `ThermalActuator` struct make the heater and the thermistor work together, and it will be used as the abstraction for the heated bed and the hotbed. The core method is `fn update(...)`, which reads the temperature from the thermistor (using the ADC) and gives it as input to the heater PID controller, which will compute the new duty cycle to correct the temperature.

```rust
pub struct ThermalActuator<'a, P: PwmBase, A: AdcBase> {
    heater: Heater<P>,
    thermistor: Thermistor<'a, A>,
}
```

- **fan**: it provides a basic implementation of a fan controller. The working logic is similar to the `Heater` struct, where the PWM peripheral is passed as a parameter to the `fn set_speed(...)` method, and the duty cycle of the inner channel is updated using the max_speed member.

```rust
pub struct FanController<P: PwmBase> {
    ch: P::Channel,
    max_speed: AngularVelocity,
}
```

3.1.2 Board

The board workspace contains target-related files, as well as the main entry of the project.

config.toml The ".cargo" folder houses the "config.toml" file, which defines configuration directives for the toolchain. This file specifies commands and settings such as the runner to use when executing `cargo run` for a specific target, the build target, and additional flags for linking and importing files.

```
[target.thumbv7em-none-eabihf]
runner = 'probe-rs run --chip STM32H753ZITx'

[build]
target = "thumbv7em-none-eabihf"
rustflags=[
    "-C", "link-arg=--nmagic",
    "-C", "link-arg=-Tlink.x",
    "-C", "link-arg=-Tdefmt.x",
]

[env]
DEFMT_LOG = "trace"
```

In this configuration, the **`[target.thumbv7em-none-eabihf]`** section sets the runner command for the specified target, the `[build]` section defines the compilation target and linking flags (files specified here are needed by Embassy[3]), and the [env] section establishes environment variables, such as setting the log level for the `defmt` debugging framework.

memory.x The "memory.x" file tells the compiler the structure of the board memory. This enables also the possibility of declaring specific sections of the memory that will be explicitly used inside the code. For instance, in this file, the `ram_d3` has been declared.

```
MEMORY
{
    RAM      (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
    RAM_D1   (xrw) : ORIGIN = 0x24000000, LENGTH = 512K
    RAM_D2   (xrw) : ORIGIN = 0x30000000, LENGTH = 288K
    RAM_D3   (xrw) : ORIGIN = 0x38000000, LENGTH = 64K
    ITCMRAM  (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
    FLASH    (rx)  : ORIGIN = 0x8000000,  LENGTH = 2048K
}

SECTIONS
{
    .ram_d3 :
    {
        *(.ram_d3)
    } > RAM_D3
}
```

To use the section inside the code, the user can leverage the *`#[link_section]`* directive, as follows:

```
#[link_section = ".ram_d3"]
static UART_RX_DMA_BUF: StaticCell<[u8; MAX_MESSAGE_LEN]> =
    StaticCell::new();
```

In this example, the section uses the `RAM_D3` memory portion because it is shared with the DMA. This way, the DMA can read and write from and to the memory.

rust-toolchain.toml The "rust-toolchain.toml" file tells cargo to use a specific channel and target to compile the current workspace.

```
[toolchain]
channel = "nightly-2024-10-13"
targets = [
    "thumbv7em-none-eabihf",
]
```

Here, the file tells that cargo needs to use the nightly channel of the toolchain of 2024-10-13, and the target is "thumbv7em-none-eabihf", which represents a Cortex-M7 microprocessor.

Dependencies

- **embassy-stm32**[6]: this crate aims to provide a safe, idiomatic hardware abstraction layer for all STM32 families. The HAL implements both blocking and async APIs for many peripherals. Where appropriate, traits from both blocking and asynchronous versions of embedded-hal v0.2 and v1.0 are implemented, as well as serial traits from embedded-io[-async]. The user can tell the compiler which chip he is using with a feature flag:

  ```
  embassy-stm32 = {
      version = "0.1.0",
      git="https://github.com/embassy-rs/embassy",
      features = [..., "stm32h753zi", ...]
  }
  ```

  The `stm32-metapac` module generates register types for that chip at compile time, based on data from the `stm32-data` module. The `embassy-stm32`[6] HAL picks the correct implementation of each peripheral based on automatically generated feature flags and applies any other tweaks that are required for the HAL to work on that chip.

- **embassy-sync**[7]: contains synchronization primitives and data structures with async support, such as `Channel`, Multiple Producer-Multiple Consumer (MPMC) channel, and `Mutex` (for synchronizing state between asynchronous tasks)

- **embassy-executor**[4]: This is an async/await executor specifically designed for embedded systems. It operates without requiring dynamic memory allocation or a heap. When using nightly Rust, task futures can be entirely allocated statically. The executor includes an integrated timer queue, allowing you to easily handle delays with calls like `Timer::after_secs(1).await;`. There's no need for busy-loop polling, as the CPU automatically sleeps when there's no work to do, relying on interrupts or WFE/SEV instructions for efficiency. Polling is optimized to ensure that only the task that was woken is polled, rather than all tasks. The executor is also fair, preventing any single task from monopolizing CPU time, even if it is constantly woken. Other tasks are guaranteed a chance to run before the same task is polled again. When using nightly Rust, enabling the nightly Cargo feature allows the executor to utilize the `type_alias_impl_trait` feature, enabling static allocation for all tasks. Each task is assigned its own static memory region, with the required size computed at compile time-based on the task or its pool size. If the tasks collectively exceed the available RAM, this is detected at compile time by the linker, eliminating the possibility of runtime memory-related panics.

- **embassy-time**[8]: embassy-time provides functionality for timekeeping, delays, and timeouts. Timekeeping is based on the elapsed time since the system booted. Time is measured in ticks, with the tick rate defined either by the hardware driver (for fixed-rate ticks) or specified by the user via a tick rate feature. This tick rate applies globally to everything in `embassy-time`[8], setting the maximum timing resolution as $\frac{1}{tick_rate}$ seconds. The time module relies on a global "time driver" selected at build time. Only one driver can be active in a program, and all methods and structures in `embassy-time`[8] interact transparently with the active driver. This design allows libraries to utilize `embassy-time`[8]

in a driver-independent manner without needing to specify generic parameters, making it versatile and easy to integrate.

```
embassy-stm32 = {
    version = "0.1.0",
    git="https://github.com/embassy-rs/embassy",
    features = [..., "time-driver-tim1", ...] }
```

Here, the TIM1 timer is used to provide the global timing.

- **embassy-futures**[5]: Utilities for working with futures, compatible with no-std and not using `alloc`. Optimized for code size, ideal for embedded systems. Future combinators, like `join` and `select` utilities to use async without a fully fledged executor: `block_on` and `yield__now`.

- **embedded-sdmmc**[9]: provides a straightforward way to read/write files on a FAT formatted SD card, as easily as using the SdFat Arduino library.

Build dependencies The [build-dependencies] section of the "Cargo.toml" contains dependencies for the "build.rs" script.

- **serde**[25]: a framework for serializing and deserializing Rust data structures efficiently and generically. The Serde[25] ecosystem consists of data structures that know how to serialize and deserialize themselves along with data formats that know how to serialize and deserialize other things. Serde[25] provides the layer by which these two groups interact with each other, allowing any supported data structure to be serialized and deserialized using any supported data format. The crate supports several data formats, like JSON, YAML, TOML, URL, etc.

- **confy**[2]: it takes care of figuring out operating system-specific and environment paths before reading and writing a configuration. It gives the user easy access to a configuration file which is mirrored into a Rust struct via serde[25]. confy[2] uses the `Default` trait in Rust to automatically create a new configuration if none is available to read from yet.

- **quote**[20]: provides the `quote!` macro for turning Rust syntax tree data structures into tokens of source code. Procedural macros in Rust receive a stream of tokens as input, execute arbitrary Rust code to determine how to manipulate those tokens, and produce a stream of tokens to hand back to the compiler to compile into the caller's crate. Quasi-quoting is a solution to one piece of that — producing tokens to return to the compiler.

- **syn**[26]: parsing library for parsing a stream of Rust tokens into a syntax tree of Rust source code.

Crates This workspace contains only one crate, **app**, which is the core crate of the project, and it's made up of several files:

- config/config.toml: the configuration file of the 3D printer

- ext.rs: a bridge between the code generated by the build script and the main code

- lib.rs: contains implementations of the traits developed in section 3.1.1

- config.rs: contains structs used by the main script to configure the different tasks

- main.rs: the main entry-point of the project, where tasks are allocated and peripherals are initialized

config.toml The "config.toml" file provides users with a comprehensive way to customize the 3D printer's behavior through an extensive set of parameters. These parameters include critical settings such as the pins assigned to drive the stepper motors, the maximum allowable temperature for the hot end, and much more. This level of granularity is essential because each printer board may have unique pin assignments and peripheral configurations. Additionally, some pins might already be in use for other components, and the ability to change these settings without altering the actual source code ensures flexibility and reduces the risk of introducing errors. The configuration file is written in the TOML (Tom's Obvious, Minimal Language) format, which is designed to be both user-friendly and human-readable. Even individuals with no prior experience in programming or technical knowledge can easily edit this file to tailor the printer's operation to their needs.

```toml
[motion]
arc_unit_length = 0.0
feedrate = 0.0
positioning = "absolute"
feedrate_multiplier = 1

[motion.retraction]
feedrate = 0.0
length = 0.0
z_lift = 0.0

[steppers.x]
stepping_mode = "full"
distance_per_step = 1
steps_per_revolution = 200
bounds = [-10, 10]
positive_direction = "clockwise"
step.pin = "PA0"
dir.pin = "PA1"

[pwm]
frequency = 0
timer = "TIM2"
ch1 = "PA11"
ch2 = "PA13"
ch3 = "PA14"

[hotend.thermistor.adc]
peripheral = "ADC1"
input.pin = "PA3"
dma.peripheral = "DMA1_CH2"
```

build.rs The printer's configuration file contains numerous values that must be validated at compile time to ensure a type-safe environment throughout the system. This requirement arises because the use of generic structures, such as a stepper motor or other components, necessitates that the compiler determines the actual types at compile time. In embedded systems programming, each hardware pin and peripheral is often represented as a unique struct. For instance, in the embassy[3] crate, the pin PB2 is represented as the PB2 struct, and similarly, peripherals like ADC1 are defined as structs. However, these hardware-specific types are determined by the printer configuration, which specifies which peripherals and pins are used. For example, the thermistor in the hot end might utilize the ADC1 peripheral for temperature readings. Since these types need to be resolved before component initialization, they must be available during the compilation process. This is where the "build.rs" file comes into play. The "build.rs" file is a specialized Rust script that runs before any other file in the crate during the build process. It is automatically compiled for the target platform being

used, allowing it to generate or validate configuration-specific code. Notably, even if the main crate is no-std (intended for embedded systems without access to standard libraries), the "build.rs" file operates in a standard Rust environment (std) since it runs on the host machine. This file works by mirroring the printer configuration as a hierarchy of structs, which are serialized and deserialized using the **serde**[25] crate. Each struct corresponds to a specific section of the configuration, and the configuration parsing process is streamlined by deriving the `Serialize` and `Deserialize` traits provided by serde[25]. This ensures that the configuration values are properly translated into strongly typed Rust code, enabling the compiler to enforce type safety and consistency across the entire system. By leveraging "build.rs", the codebase achieves both flexibility and robustness, allowing seamless integration of hardware-specific configurations while maintaining compile-time guarantees.

```rust
#[derive(Default, Serialize, Deserialize)]
pub struct PeripheralConfig {
    peripheral: String,
}

...

#[derive(Default, Debug, Serialize, Deserialize, Clone)]
    pub struct ThermalActuatorConfig {
        pub thermistor: ThermistorConfig,
        pub heater: HeaterConfig,
}

...

#[derive(Default, Debug, Serialize, Deserialize, Clone)]
    pub struct MyConfig {
        pub steppers: StepperConfigs,
        pub pwm: PwmConfig,
        pub uart: UartConfig,
        pub adc: AdcConfig,
        pub hotend: ThermalActuatorConfig,
        pub heatbed: ThermalActuatorConfig,
        pub fan: FanConfig,
        pub sdcard: SdCardConfig,
        pub motion: MotionConfig,
    }
}
```

The primary function of the build script is to read and process the printer configuration using the **confy**[2] crate. The configuration file, which is typically located in the "config/config.toml" path, is loaded into a structured format, which represents the various parameters specified by the user. The script ensures the configuration file is up-to-date and triggers a rebuild if it is changed. Here's a simplified version of the main function:

```rust
fn main(){
    println!("cargo::rerun-if-changed=config/config.toml");
    let path = Path::new("config/config.toml");
    let conf = confy::load_path::<external::MyConfig>(path)
        .expect("Error reading config file");
    ...
}
```

After successfully reading the configuration, the script performs a series of validations to ensure the provided values are consistent, valid, and within acceptable ranges. This step is critical to catch potential errors early and prevent issues during runtime or compilation. Once the checks are complete, the script proceeds to generate Rust code dynamically. This is accomplished using the `quote!` macro from the quote[20] crate and the `Ident` struct from the syn crate. The generated code is saved as a file that becomes part of the build process. This generated file typically includes two essential components:

- alias types: Alias types are created to simplify and abstract hardware-specific details. These aliases allow developers to reference a type without needing to know its exact implementation details. For instance, if the step pin for the X-axis stepper motor is defined in the configuration, it is assigned an alias such as `XStepPin`. This way, even if the underlying type of the pin changes (e.g., from PB2 to PA3), the rest of the code can continue using `XStepPin` without modification, ensuring both flexibility and readability.
- peripherals_ini: this function plays a pivotal role in the 3D printer's setup process by bridging the gap between the hardware peripherals and the internal configuration of the printer. This function accepts the `Peripherals` struct from the `embassy_stm32`[6] crate as its parameter. The `Peripherals` struct provides access to all the hardware resources of the microcontroller, including pins, timers, ADCs, and other peripherals. Using these resources, the function constructs the internal configuration of the printer, represented by the `PrinterConfig` struct.

```
pub struct StepperConfig<S, D> {
    pub step_pin: S,
    pub dir_pin: D,
    pub stepping_mode: SteppingMode,
    pub distance_per_step: Length,
    pub steps_per_revolution: u64,
    pub bounds: (Distance, Distance),
    pub positive_direction: RotationDirection,
}
...

pub struct PrinterConfig<
    XP,
    XD,
    YP,
    YD,
    ZP,
    ...,
> {
    pub steppers: SteppersConfig<XP, XD, YP, YD, ZP, ZD, EP, ED>,
    pub pwm: PwmConfig<PWMT, CH1, CH2, CH3>,
    pub uart: UartConfig<UP, RXP, RXD, TXP, TXD>,
    pub adc: AdcConfig<ADCP, ADCD>,
    pub hotend: ThermalActuatorConfig<HOI>,
    pub heatbed: ThermalActuatorConfig<HEI>,
    pub fan: FanConfig,
    pub sdcard: SdCardConfig<SPIP, SPIT, SPIMO, SPIMI, SPICS>,
    pub motion: MotionConfig,
    pub endstops: EndstopsConfig<XEP, XEE, YEP, YEE, ZEP, ZEE>,
}

...
```

The `PrinterConfig` struct uses generic parameters to represent pins and peripherals, ensuring flexibility and type safety. These generics allow the system to work with specific hardware components while

maintaining a high level of abstraction. In addition to hardware components, the internal configuration includes more explicit and domain-specific types, such as `Distance` and `Temperature`, for various operational values. These types enhance code readability and provide additional type safety.

```
pub fn peripherals_init(p: Peripherals) -> PrinterConfig<
    XStepPin,
    XDirPin,
    ...
>{
    PrinterConfig{
        steppers: SteppersConfig{
            x: StepperConfig{
                step_pin: p.#steppers_x_step_pin,
                dir_pin: p.#steppers_x_dir_pin,
                ...
            }
        },
        ...
    }
}
```

The `quote!` macro generates a `TokenStream`, which is an abstract representation of Rust code as a sequence of tokens. This allows the program to dynamically construct and manipulate code in a structured and type-safe manner. Once the `TokenStream` is created, it can be serialized into a string representation, making it suitable for file output. In the context of this build process, the generated code is written to a file named "_generated.rs". This file serves as an intermediary between the build script and the final compiled binary, containing auto-generated code like type aliases, configurations, or setup functions. This approach is especially useful for embedding custom logic or configuration data directly into the compiled project without requiring manual edits to the source code.

```
let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());
let out_file = out_dir.join("_generated.rs").to_string_lossy()
    .to_string();
fs::write(&out_file, tokens.to_string().as_str()).unwrap();
```

ext.rs This file is used to import the generated code from the "build.rs" script in a transparent way:

```
include!(concat!(env!("OUT_DIR"), "/_generated.rs"));
```

lib.rs This file contains the implementation of the `AdcBase`, `PwmBase`, `OutputPinBase` and `InputPinBase` trait for the actual structs provided by the `embassy_stm32`[6] crate.

3.2 Task implementation

The "main.rs" file of the project puts together everything that has been described until now. Each control part of a 3D printer belongs to a separate task, which is an asynchronous function configured with the `#[embassy_executor::task]` attribute. The system is organized into 7 tasks (as shown in fig. 21) :

- Input task: receives messages from UART
- Command dispatcher task: parses incoming messages into readable GCode commands and dispatch them to other tasks
- Hotend task: handles the hot end as well as the cooling fan
- Heatbed task: handles the heated bed bed
- SD-Card task: interact with the SD-card and read GCode commands from files
- Planner: handles the steppers, as well as the end-stops
- Output task: transmit messages to UART

Every task is spawned from the main entry, configured as `#[embassy_executor::main]`.

```
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    ...
    spawner.spawn(input_handler()).unwrap();
    spawner.spawn(output_handler()).unwrap();
    spawner.spawn(command_dispatcher_task()).unwrap();
    spawner
        .spawn(heatbed_handler(printer_config.heatbed))
        .unwrap();
    spawner
        .spawn(hotend_handler(printer_config.hotend, printer_config.fan))
        .unwrap();
    ...
}
```

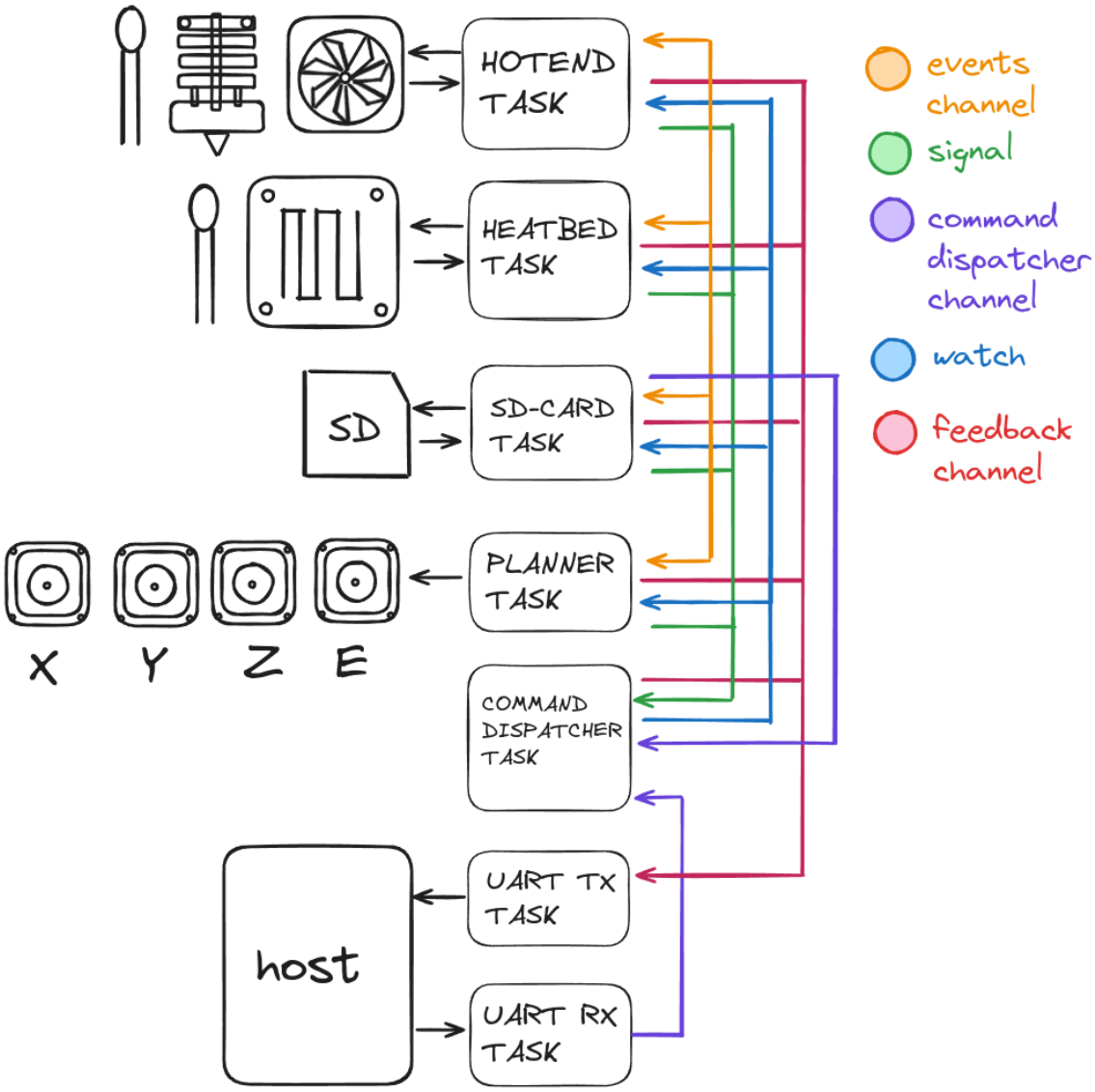


Figure 21: Xtrooder system architecture

The data between tasks is shared as shown in fig. 21. The input task (shown as UART RX TASK) waits for incoming data from UART, and as soon as something is received, it's pushed inside the `COMMAND DISPATCHER CHANNEL`, which is the input channel of the command-dispatcher task.

```
#[derive(Clone, Copy, PartialEq, PartialOrd, Eq)]
enum TaskMessagePriority{
    Low,
    Medium,
    High
}

#[derive(Clone, PartialEq, PartialOrd, Eq)]
struct TaskMessage{
    msg: String<MAX_MESSAGE_LEN>,
    priority: TaskMessagePriority
}

static COMMAND_DISPATCHER_CHANNEL: PriorityChannel<
    ThreadModeRawMutex,
    TaskMessage,
```

```
    Max,
    COMMAND_DISPATCHER_CHANNEL_LEN,
> = PriorityChannel::new();
```

The channel is a `PriorityChannel` struct, provided by the embassy-sync[7] crate, which is essentially a priority queue, where elements with higher priority are dequeued before elements with lower priority. This ensures that, for instance, messages coming from the UART (e.g. print abort or print stop) are processed before messages coming from the SD-card (e.g. linear motion or temperature calibration).

The output produced by the dispatcher task is a GCode command wrapped inside the `TaskGCommand` struct, which is exposed to the other tasks using the `Watch` primitive of the embassy-sync[7] crate. `Watch` is a single-slot signaling primitive that allows multiple receivers to concurrently await for changes in its value.

```
#[derive(Clone)]
struct TaskGCommand{
    cmd: GCommand,
    destination: u8
}

static WATCH: Watch<ThreadModeRawMutex, TaskGCommand, TASKS_N> =
    Watch::new();
```

The `TaskGCommand` provides a `destination`, which is a bit-mask that contains the ID of the tasks that need to read that value. Each destination task is saved inside the `destination` using bit-shifting.

```
#[derive(Clone, Copy)]
enum TaskId{
    Input,
    Output,
    CommandDispatcher,
    Hotend,
    Heatbed,
    SdCard,
    Planner
}

GCommand::M105 { .. } | GCommand::M155 { .. } => {
    destination = 1u8 << u8::from(TaskId::Hotend) |
                  1u8 << u8::from(TaskId::Heatbed);
}
```

When a task needs to wait for changes of the `WATCH` variable, for example the hot-end task, it performs:

```
...
let cmd = WATCH.receiver().changed().await;
if cmd.destination & (1u8 << u8::from(TaskId::Hotend)) != 0{
    \\ do something
}
```

Once a task has finished the operation related to the received command, it signals the command-dispatcher task using the `Signal` primitive of the embassy-sync[7] crate.

```
static SIGNAL: Signal<ThreadModeRawMutex, TaskId> = Signal::new();
```

The command-dispatcher task will wait until every destination task has signaled that the GCode command has been processed.

```
let task_command = TaskGCommand{
    cmd,
    destination
};
watch_sender.send(task_command);
// wait for tasks response before proceeding to parse the next command
let mut res = 0u8;
while res & destination != destination {
    let s = SIGNAL.wait().await;
    res |= 1u8 << u8::from(s);
}
```

This workflow ensures that each GCode command is correctly processed by every destination task before proceeding to the next one. If a task needs to send a message via UART (e.g. to list files inside the file-system of the SD-card), it can enqueue the message inside the `FEEDBACK_CHANNEL`, which will then be read by the output task and sent via UART to the external device:

```
static FEEDBACK_CHANNEL: Channel<
    ThreadModeRawMutex,
    String<MAX_MESSAGE_LEN>,
    FEEDBACK_CHANNEL_LEN,
> = Channel::new();
```

3.2.1 Input handler

The `embassy_stm32`[6] crate separates the UART functionality into two distinct structs: `UartRx` (for receiving data) and `UartTx` (for transmitting data). This design simplifies UART management by allowing the reception and transmission of data to be handled independently in two different tasks. However, while the UART is ultimately divided into these two structs, it is initially instantiated as a single struct and then split using the split method. To ensure safe concurrent access, the resulting `UartRx` and `UartTx` structs must be stored in static variables protected by a mutex. This approach guarantees thread safety and allows these static instances to be accessed later in their respective tasks. The splitting process is typically done in the main function, where the UART is initialized, split, and assigned to the static variables,

In order to leverage the DMA to detect data, we need to allocate the destination buffer in a section of the memory that is shared with the DMA. This is performed, as previously seen, using the `#[link_section]`:

```
#[link_section = ".ram_d3"]
static UART_RX_DMA_BUF: StaticCell<[u8; MAX_MESSAGE_LEN]> =
    StaticCell::new();
```

The task is then implemented as follows: it awaits incoming data from the UART using a mutable reference to the static UART_RX variable. It then iterates over the received data, appending everything inside an auxiliary buffer. Once a newline character is found, the `msg` buffer is sent to the command dispatcher, then it is cleared.

The correct reception of each message is backed by the use of the UART idle line detection, a mechanism used to detect when the communication line remains idle for a specific duration. This feature is often employed in protocols where communication consists of variable-length messages without explicit delimiters, enabling the receiver to recognize the end of a transmission or detect line inactivity. The UART considers the line “idle” when no data frames (start bit, data bits, or stop bits) are being transmitted for a certain period. This period is typically measured as the time it takes to transmit one or more complete data frames, depending on the UART configuration. Once the idle condition is detected, the UART generates an idle interrupt (IDLE flag in many microcontroller UART implementations). This interrupt signals to the CPU that the line has been idle for the configured period. This way the caller can await for the data while doing other running other tasks.

3.2.2 Output handler

The output handler awaits incoming messages on the feedback channel and sends them over UART. The message is awaited on the feedback channel, and as soon as a message is received, its content is copied inside the UART DMA buffer.

```
#[link_section = ".ram_d3"]
static UART_TX_DMA_BUF: StaticCell<[u8; MAX_MESSAGE_LEN]>
    = StaticCell::new();

...
```

3.2.3 Command dispatcher

The command dispatcher is the main channel of the whole system. It awaits a message from the command dispatcher channel, parses it using the `GCodeParser`, and finally matches the GCode command to decide which task is in charge of it.

3.2.4 Hotend handler

The hot end handler, as the name suggests, controls the hot end. It first initializes the different structs that compose the hot end, such as the thermistor, the heater, and the fan, using the configuration passed by parameter from the main. Then, at each iteration of the loop, it reads the temperature from the thermistor and checks if it is out of the temperature bounds. If so, the task reports an error of hot end overheating, which is pushed into the event channel, and sends a report message to the feedback channel. By doing that, the output task can forward the message to UART. Next, it checks if there are errors inside the event channel. If something is found (heated bed overheating, stepper error, or print completed), the hot end is disabled. The remaining rest of the code inside the loop handles different commands regarding the hot end and the fan.

3.2.5 Heatbed handler

The heated bed handler works very similarly to the hot end task, but it doesn't handle any fan.

3.2.6 SD-Card handler

The SD-Card handler uses the `SdCard` struct provided by the `embedded_sdmmc` crate to wrap the SPI peripheral and let the user interact with the SD-Card safely and clearly. The struct, however, accepts as argument some type that implements the blocking behavior of the trait `embedded_hal::spi::SpiDevice`. This is why we cannot use the asynchronous version of the SPI provided by the embassy[3] crate, yet. At each iteration, the task check if there's an open file to read GCode from, reads a certain amount of data, and once a newline is encountered, it enqueue the message inside the `COMMAND_DISPATCHER_CHANNEL`, with a low-priority message.

3.2.7 Planner

The planner task handles the steppers and the end-stops, and executes only motion-related operations.

3.3 Error handling

The 3D printer needs to handle several types of errors. The software handles the most basic and dangerous events that can happen:

```
#[derive(Clone, Copy, Debug)]
pub enum PrinterEvent {
    HotendOverheating(Temperature),
```

```
    HotendUnderheating(Temperature),
    HeatbedOverheating(Temperature),
    HeatbedUnderheating(Temperature),
    Stepper(StepperError),
    EOF,
    PrintCompleted,
}
```

- Hotend overheating: This can occur due to improper PID parameter configuration or a faulty thermistor. If not managed, it could damage the heating block.
- Heatbed overheating: Similar to hot end overheating, this arises from misconfigurations or hardware malfunctions, potentially causing damage to the heated bed.
- Stepper error: These include situations such as moves that are too small to execute (e.g., less than a single step), out-of-bounds movements, or collisions with activated end-stops.
- PrintCompleted: Indicates the successful completion of a print from the SD card.

To manage these events, the system utilizes a Multiple-Producer-Multiple-Subscriber (MPMS) queue. Events are shared across tasks using this queue, but no single task consumes the events exclusively. Instead, all tasks can read the latest queue element to perform their respective operations.

```
static EVENT_CHANNEL: PubSubChannel<
    ThreadModeRawMutex,
    PrinterEvent,
    EVENT_CHANNEL_CAPACITY,
    EVENT_CHANNEL_SUBSCRIBERS,
    EVENT_CHANNEL_PUBLISHERS,
> = PubSubChannel::new();
```

For example, in the event of heatbed overheating:

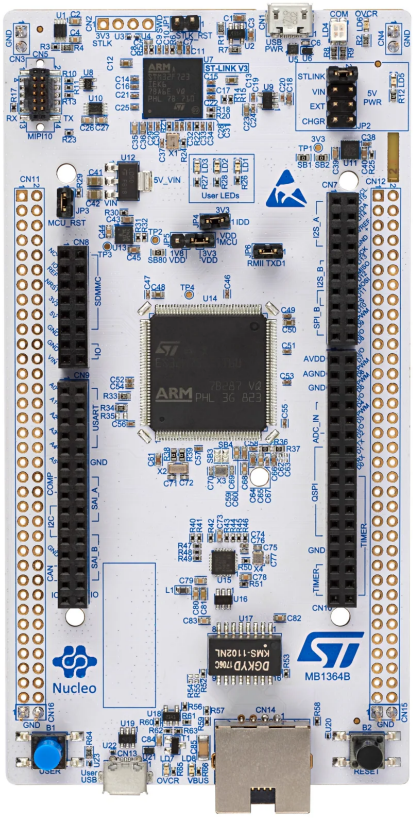
- The Hotend/Heatbed tasks deactivate the PWM channels.
- The Planner task clears the commands queue.
- The SD-Card task stops reading data and safely closes all open files and folders.
- The Output task reports the issue with a detailed UART message for user awareness.

4 Test and validation

The project has been thoroughly tested and validated on the NUCLEO-H753ZI development board, which is powered by the STM32H753ZI microcontroller. This microcontroller is equipped with a 32-bit Arm Cortex-M7 RISC core, delivering high performance with a clock speed of up to 480 MHz. The Cortex-M7 core features branch prediction and a 6-stage superscalar pipeline, enabling it to handle demanding real-time applications with exceptional speed and efficiency. The MCU provides these main features:

- Floating Point Unit (FPU): Supports both single- and double-precision calculations, critical for high-precision computations in applications like signal processing and control systems.
- 2 MB of Dual-Bank Flash Memory
- 1 MB of SRAM: Includes tightly coupled memory (TCM) for low-latency access, enhancing performance in time-sensitive operations.

- 4 DMA Controllers: Enable efficient data transfers between memory and peripherals without burdening the CPU, ideal for high-throughput tasks.
- 3 ADCs (16-bit resolution): Deliver precise analog-to-digital conversion, supporting up to 36 input channels, suitable for advanced sensor integration and data acquisition systems.
- 6x SPI
- 4X UART / 4x USART
- 10x General-purpose timers (PWM included): Provide extensive timing and control capabilities, including motor control, input capture, and signal generation.



4.1 XTrooder vs Marlin

Xtrooder is developed in Rust, whereas Marlin[14] is developed in C++. For compilation, Marlin[14] relies on the g++ compiler, part of the GNU Compiler Collection (GCC), while this project uses rustc, the official compiler for Rust.

4.1.1 Compiler

The *rustc* compiler is built on top of LLVM (Low-Level Virtual Machine), which acts as its backend. This architecture allows *rustc* to leverage LLVM's optimization capabilities while implementing Rust's unique features like ownership model, borrow checking, and lifetimes, ensuring memory safety and preventing errors such as null pointer dereferences and data races without relying on garbage collection. During the compilation process, *rustc* parses Rust source code into an Abstract Syntax Tree (AST), then converts the AST into LLVM Intermediate Representation (IR). LLVM applies advanced optimizations to the IR and translates it into target-specific machine code for architectures like x86 and ARM. *rustc* also integrates seamlessly with *Cargo*, Rust's build system and package manager, streamlining dependency management, testing, and builds. Additionally, it supports profile-guided optimization (PGO) and link-time optimization (LTO) to enhance performance further.

On the other hand, the *g++* compiler is a widely used and versatile tool for compiling C++ code. It supports multiple C++ standards as well as cross-compilation for various architectures, including x86, ARM, and AVR. G++ provides extensive optimization options to fine-tune performance, along with link-time optimization (LTO), enabling whole-program analysis and improvements during the linking stage.

rustc emphasizes safety and modern programming paradigms, delivering optimized and reliable code while avoiding common pitfalls of manual memory management. In contrast, *g++* offers fine-grained control and flexibility, making it well-suited for applications requiring low-level optimizations and compatibility with the extensive C++ ecosystem. Both compilers reflect the strengths of their respective languages and are tailored to their unique design philosophies.

4.1.2 Binary size

To ensure a fair and consistent comparison between the two solutions, both have been compiled with the minimal set of features necessary to operate a 3D printer while providing a safe and smooth user experience. The features include:

- Support for three stepper motors controlling the X, Y, and Z axes.
- One extruder motor for filament feeding (E-axis).
- Serial communication via UART
- Three end-stops for positional feedback on the X, Y, and Z axes.
- Safety mechanisms to prevent overheating of the heated bed and hot-end.
- Linear and arc motion capabilities, complete with acceleration control for precise and smooth movements
- Filament retraction and recovery functionality for cleaner prints and reduced stringing
- SD-card support to enable offline printing from external storage

Both binaries are compiled for the *thumbv7em-none-eabihf* target, which corresponds to Arm Cortex-M architectures with hardware floating-point support.

The release build of Xtrooder is performed using the following configuration:

```
...
# cargo build/run --release
[profile.release]
codegen-units = 1
debug = false
debug-assertions = false # <-
incremental = false
strip = true # strip symbols from the binary
lto = true # enable link time optimization
opt-level = "z" # optimize for binary size
overflow-checks = true
panic = "abort" # abort on panic without stack trace
...
```

- codegen-units = 1: Reduces the number of parallel code generation units to one. This can improve optimization at the cost of longer compile times, as the compiler has a more global view for optimizations.
- debug = false: Disables debug information in the release binary, making it smaller.
- debug-assertions = false: Disables runtime checks for debug assertions in release builds. This slightly improves runtime performance.

- incremental = false: Disables incremental compilation for release builds, ensuring fully optimized builds every time but potentially increasing compile times.
- strip = true: Removes symbols (e.g., function names, variable names) from the compiled binary, further reducing size and preventing reverse-engineering of certain details.
- lto = true (Link-Time Optimization): Enables link-time optimization, which allows cross-module optimizations. It may increase compile times but generally improves runtime performance and reduces binary size.
- opt-level = ”z”: Sets the optimization level to prioritize binary size reduction. This might sacrifice some runtime speed compared to opt-level = 3.
- overflow-checks = true: Enables runtime checks for arithmetic overflow. This helps detect and prevent potential bugs related to overflows, improving runtime safety.
- panic = ”abort”: Configures the program to abort on panic instead of unwinding the stack. This reduces binary size and eliminates the need for the panic stack-trace machinery, which is unnecessary in many embedded or size-critical environments.

```
du -h target/thumbv7em-none-eabihf/release/app
152K    target/thumbv7em-none-eabihf/release/app

du -h .pio/build/BTT_SKR_SE_BX/firmware.bin
115K    .pio/build/BTT_SKR_SE_BX/firmware.bin
```

The binary for Xtrooder, written in Rust, is larger at 152 KB compared to Marlin[14]’s binary size of 115 KB, compiled in C++. This difference of approximately 37 KB highlights certain distinctions between the two approaches. Rust includes additional metadata and runtime checks to uphold safety guarantees, such as bounds checking and the ownership system. While these contribute to code safety and robustness, they may increase binary size compared to C++. Rust’s compiler (*rustc*), built on LLVM, performs aggressive optimizations but may not always achieve the same low-level binary size as *g++* for simpler codebases. C++ binaries can be fine-tuned with targeted use of templates and direct memory management, often leading to smaller code when safety is not a primary concern. While the Rust binary is slightly larger, the trade-off comes in the form of safety guarantees, maintainability, and modern abstractions that Rust provides out-of-the-box. C++ offers smaller binaries and potentially more control over resource usage but requires greater care to avoid safety pitfalls.

4.1.3 Platform support

The platform support of both projects has been checked against the most used development boards and MCUs in the 3D printing and hobbies environment.

- ATMega: family of microcontrollers developed by Atmel Corporation, based on the AVR architecture, which combines the RISC (Reduced Instruction Set Computer) principles with efficient instruction execution. AVR is a 8-bit RISC-based architecture that operates at clock speeds up to 20 MHz, and uses a flash of about 32KB for program storage. Mostly found in Arduino boards.
- nRF: family of microcontrollers developed by Nordic Semiconductors, based on ARM cortex architecture. Optimized for low-energy applications, extensive support for BLE, 32-bit architecture with around 1MB of flash and 256KB or RAM
- STM32: family of microcontrollers developed by STMicroelectronics, well known for performance, scalability and feature-rich designs. Sits on ARM architecture, and has a large set of different series for different purposes (STM32F are general-purpose, while STM32U are ultra-low power ith advanced peripherals).

- ESP32: family of microcontrollers developed by Espressif Systems, based on XTensa architecture. It features a RAM between 320KB and 520KB and clock-speed up to 240MHz
- Teensy: family of microcontrollers developed by PJRC, known for their high performance, compact size and easy-to-use interface. Based on ARM architecture, clock speed ranges from 72MHz to 1GHz, RAM between 2KB to 1MB and flash from 23KB to 2MB.
- RP: family of microcontrollers designed by the Raspberry Pi Foundation. Based on ARM architecture, up to 133MHz of clock-speed, 256KB of RAM and relies on external memory
- CH32: family of microcontrollers, developed by WCH (Nanjing QinHeng Microelectronics), built on ARM cores with similar performances to STM32 family

The following table will provide a compatibility comparison for the previously listed platforms, between Marlin[14] and Xtrooder. The forth column shows also the support provided by the embassy[3] framework regarding each platform, to show which HAL has already been implemented.

MCU	XTrooder	Marlin	embassy
ATmega1280		X	
ATmega1281		X	
ATmega2560		X	
ATmega2561		X	
ATmega644P		X	
ATmega1284P		X	
nRF51			X
nRF52			X
nRF53			X
nRF91			X
STM32C	X		X
STM32F	X	X	X
STM32G	X		X
STM32H	X	X	X
STM32L	X		X
STM32U	X		X
STM32W	X		X
RP2040			X
RP235XA			X
RP235XB			X
RP2040			X
ESP32C2		X	X
ESP32C3		X	X
ESP32C6		X	X
ESP32H2		X	X
ESP32P4		X	X
ESP32S2		X	X
ESP32S3		X	X
CH32V0			X
CH32V1			X
CH32V2			X
CH32V3			X
CH32X0			X
CH32L1			X
TEENSY31_32		X	
TEENSY35_36		X	
TEENSY40_41		X	

Table 1: Comparison of HAL support

These days, Xtrooder firmware supports most of the STM32 MCUs, while Marlin[14] supports almost all the listed platforms. However, *embassy*[3] provides a great set of developed and well-tested HALs, and thanks to its easy-to-use API, Xtrooder has a good chance to mature and provide support to many more platforms in a small amount of time.

4.1.4 Ease of development

The ease of development has been validated in 3 steps:

- Tools: tools provided by the toolchain
- Configuration: ease of parameters tuning
- Peripheral access: API for low-level peripheral access

Tools Marlin[14] firmware enhances the traditional *make* build process by incorporating additional tools to simplify and streamline tasks, particularly in managing configurations and dependencies. The primary tool it leverages is *PlatformIO*[19], an integrated development environment and build system tailored for embedded projects. *PlatformIO*[19] significantly reduces the complexity of using make by providing a consistent build and configuration process across various hardware platforms. It also automates dependency management, including board definitions, libraries, and toolchains, while integrating seamlessly with popular editors like VS Code for a more user-friendly development experience. Additionally, Marlin[14] employs Python and shell scripts to automate tasks, such as generating firmware configurations, further easing the setup and build process.

cargo, on the other hand, is the official build system and package manager for Rust projects, designed to streamline the development process and provide robust tooling. It excels in integrated dependency management, automatically handling the downloading, building, and management of crates (Rust libraries) from sources like *crates.io*. With its ease of use, Cargo requires minimal setup—developers define dependencies and build settings in a "Cargo.toml" file, leaving Cargo to manage the rest. *cargo* offers seamless tooling integration, including built-in support for testing with cargo test, benchmarking, and generating project documentation with cargo doc. These tools are readily accessible, enhancing productivity without requiring additional configurations.

Configuration The configuration file enables the user to edit a large set of parameters used by the program to perform computation or to include/exclude specific pieces of code to reduce the size of the binary file. The Marlin[14] environment provides a "config.ini" file, a human-readable configuration which is a wrapper around the 2 main configuration files, implemented as header files: "Configuration.h" and "Configuration_adv.h".

```
[config:minimal]
motherboard                             = BOARD_RAMPS_14_EFB
serial_port                             = 0
baudrate                                = 250000

[config:advanced]
arc_support                             = on
auto_report_temperatures                = on
autotemp                                = on
autotemp_oldweight                      = 0.98
max_cmd_size                            = 96
```

In the example above, the motherboard is configured, along with the serial port of the UART communication and the related baud-rate. But considering the fact that most of the time the user won't change the configuration of the pins, because each printer comes with their custom pins, this type of configuration does not provide the possibility to manually map each pin of the board to a given functionality. The mapping, in fact, can be found inside the "pins" folder of the project, and every board has its specific pin mapping.

```
//
// Steppers
//
#define X_STEP_PIN                          PG13  // X
#define X_DIR_PIN                           PG12
```

```
#define X_ENABLE_PIN                    PG14
#define X_CS_PIN                        PG10

#define Y_STEP_PIN                      PB3   // Y
#define Y_DIR_PIN                       PD3
#define Y_ENABLE_PIN                    PB4
#define Y_CS_PIN                        PD4
```

Xtrooder, on the other hand, provides a full-customizable 3D printer configuration in TOML format, as seen in the configuration section.

```
[steppers.x]
stepping_mode = "full"
distance_per_step = 0.15
steps_per_revolution = 200
bounds.min = -100
bounds.max = 100
positive_direction = "counterclockwise"
step.pin = "PC11"
dir.pin = "PC10"
```

Peripheral access Marlin[14] directly manipulates hardware registers to control peripherals such as stepper motors, heaters, and sensors. This provides high efficiency but requires detailed knowledge of the microcontroller's architecture. Heavily relies on interrupts for time-sensitive operations like stepper motor control. The code often includes device-specific code (e.g., AVR or STM32 HALs) and is usually tailored to a specific microcontroller. Resource sharing is managed manually, often with global variables or state flags. Since Marlin[14] is designed for performance and simplicity, there are fewer abstractions for safety. Mismanaging hardware peripherals can result in undefined behavior or hardware damage.

Embassy[3] uses hardware abstraction layers (HALs) that are typically provided by the Rust ecosystem (e.g., embedded-hal or chip-specific crates). and employs async/await syntax to manage peripherals in a non-blocking way. This allows tasks to yield control while waiting for I/O operations, improving efficiency. Peripherals are represented as ownership-based abstractions, preventing race conditions and unsafe sharing of resources. Embassy[3] code is portable and works across different hardware with minimal changes, thanks to its modular design.

4.2 Host testing

Every crate within the host workspace includes a built-in automated test suite, allowing developers to write and execute unit tests directly on the host machine using the cargo test command. This setup is ideal for verifying the correctness and functionality of individual components and ensures robust testing workflows. For asynchronous functions, the crates leverage the **tokio**[28] asynchronous runtime, which integrates seamlessly with Rust's test infrastructure. Developers can enable async tests by applying the `#[tokio::test]` attribute to test functions.

```
#[cfg(test)]
mod tests {
    use approx::assert_abs_diff_eq;
    use math::{
        common::RotationDirection,
        measurements::{Distance, Speed},
    };
    use tokio::time::sleep;
```

```rust
#[test]
fn test_stepper_step() {
    let step = StatefulOutputPinMock::new();
    let direction = StatefulOutputPinMock::new();
    let options = StepperOptions::default();
    let mut s = Stepper::new(step, direction, options);
    s.set_direction(RotationDirection::Clockwise);
    let res = s.step();
    assert!(res.is_ok());
    assert_abs_diff_eq!(s.get_steps(), 1.0, epsilon = 0.000001);
}

#[tokio::test]
async fn test_stepper_move_for_steps_success() {
    let step = StatefulOutputPinMock::new();
    let direction = StatefulOutputPinMock::new();
    let options = StepperOptions::default();
    let mut s = Stepper::new(step, direction, options);
    s.set_direction(RotationDirection::Clockwise);
    let angular_velocity = AngularVelocity::from_rpm(60.0);
    s.set_speed(angular_velocity);
    let steps = 20;
    let m = s.move_for_steps::<StepperTimer>(steps).await;
    assert!(m.is_ok());
    assert_abs_diff_eq!(s.get_steps(), 20.0, epsilon = 0.000001);
    assert_eq!(s.get_speed(), angular_velocity);
    assert_eq!(
        m.unwrap().as_micros(),
        Duration::from_millis(100).as_micros()
    );
}
```

5 Related work

5.1 Marlin

Marlin[14] firmware is a widely adopted open-source firmware that is primarily used in 3D printing applications but is also compatible with CNC machines and laser engravers. Originally designed for RepRap 3D printers, Marlin[14] has evolved into one of the most popular and feature-rich firmware solutions in the 3D printing ecosystem. It supports a broad range of microcontrollers, including AVR and ARM-based architectures, making it adaptable to a variety of hardware platforms (Marlin Firmware, 2024). This flexibility is one of the key reasons for its widespread use in the 3D printing community. The firmware offers extensive customization through its configuration files (Configuration.h and Configuration_adv.h), which allow users to enable or disable specific features depending on their hardware. Marlin[14] supports a wide range of kinematics, including Cartesian, Delta, and CoreXY, and can be fine-tuned to specific machine configurations. It also includes advanced motion control algorithms such as linear advance, jerk control, and acceleration management, which help improve print quality and reduce the risk of defects (Wang et al., 2021). Furthermore, Marlin[14] integrates advanced safety features like thermal runaway protection, which prevents overheating of the hotend and heated bed, and includes automatic shutdown mechanisms in the event of a fault (Marlin Firmware, 2024). One of the distinguishing features of Marlin[14] is its robust peripheral support. The firmware integrates seamlessly with various components such as LCD screens, touchscreens, SD card readers, and external control software like OctoPrint. Additionally, Marlin[14] supports multi-extruder setups, making it suitable for complex 3D printing applications that require dual or multi-material extrusion (Vasilyev, 2020). The firmware's auto-bed leveling

capabilities, which include support for popular sensors like BLTouch, are particularly important for ensuring consistent print quality, especially on machines with large or uneven print beds (Wang et al., 2021). Despite its broad capabilities, Marlin[14] is not without challenges. For beginners, configuring Marlin[14] can be difficult, especially when working with custom hardware or advanced features. Incorrect configuration settings can lead to issues ranging from poor print quality to hardware malfunctions (Prusa, 2021). Additionally, Marlin[14]'s performance is closely tied to the capabilities of the underlying hardware. While the firmware is optimized for a range of microcontrollers, older 8-bit systems may struggle to handle the more resource-intensive features like advanced motion control or high-speed printing (Vasilyev, 2020). Marlin[14]'s open-source nature allows for continuous development and improvement. The firmware's active community contributes by adding new features, fixing bugs, and expanding hardware support. New releases are frequently made available, often incorporating community feedback and new technological advancements (Marlin Firmware, 2024). The development process is managed through GitHub, where users can track progress, submit bug reports, and propose new features. This collaborative model ensures that Marlin[14] remains relevant and up-to-date with the latest trends in 3D printing. In conclusion, Marlin[14] Firmware is a powerful and highly customizable platform for 3D printing and other CNC-related applications. Its wide range of supported features, such as advanced motion control, multi-extruder setups, and robust safety mechanisms, make it a top choice for both hobbyists and professionals. However, it requires a certain level of expertise to configure and optimize, particularly for users working with custom hardware. Its open-source nature ensures ongoing development and expansion, which continues to drive innovation in the 3D printing space.

5.2 Klipper

Klipper[12] firmware is an open-source firmware designed for 3D printers that aims to significantly improve the performance and flexibility of printing systems. Unlike traditional 3D printer firmware such as Marlin[14], Klipper[12] takes a unique approach by offloading much of the computational workload from the printer's microcontroller to a more powerful host computer, usually a Raspberry Pi. This design choice allows Klipper[12] to handle complex tasks such as motion planning and G-code interpretation more efficiently, resulting in higher print speeds, better precision, and smoother overall operation. One of the key features of Klipper[12] is its motion planning system. While traditional firmware relies on the microcontroller to perform real-time calculations for movement, Klipper[12] uses the host computer to perform these calculations and then sends the results to the printer's microcontroller. This division of labor allows Klipper[12] to use more powerful algorithms for motion planning, enabling higher print speeds and improved print quality. The system supports advanced features like pressure advance, which compensates for lag in extruder response during high-speed moves, and input shaping, which reduces vibration and improves the accuracy of prints at higher speeds. By relying on the host computer's processing power, Klipper[12] can also handle complex kinematics more easily, including support for non-Cartesian motion systems like Delta and CoreXY. Another important aspect of Klipper[12] is its configuration system, which is entirely text-based and edited via simple configuration files. This is a departure from traditional firmware, where the configuration is typically done in C++ code. Klipper[12]'s configuration files are human-readable and can be easily modified to suit different hardware setups, making it accessible for both beginner and advanced users. The configuration files allow users to define everything from stepper motor settings to temperature sensors, pressure advance settings, and even custom macros. This flexibility makes Klipper[12] highly customizable for various 3D printer models and configurations, providing users with a great deal of control over their machine's behavior. In terms of hardware support, Klipper[12] is compatible with a wide range of 3D printers, including both Cartesian and non-Cartesian systems. It supports many different microcontrollers, from 8-bit AVR chips to more powerful ARM-based boards, with particular emphasis on those used in popular open-source 3D printers. The software can communicate with the microcontroller over USB or serial connections, and the host computer runs the Klipper[12] firmware, which sends the calculated commands to the microcontroller for execution. This system is highly scalable, allowing it to be used on everything from small desktop printers to large industrial machines. Klipper[12] also boasts a web interface known as Fluidd or Mainsail, which allows users to monitor and control their printer remotely through a browser. This interface is typically run on the host computer (such as a Raspberry Pi) and provides real-time data on the printer's status, as well as the ability to control the machine, upload G-code, and monitor the progress of prints. The web interface enhances the user experience by providing a simple and intuitive way to manage 3D printing operations remotely, eliminating the need for a dedicated screen or control panel on the printer itself. The performance

improvements offered by Klipper[12] are significant, particularly in terms of print quality and speed. Because the host computer handles the more computationally demanding tasks, the microcontroller can focus solely on executing the commands sent to it. This separation of tasks enables faster processing and better motion control, which can result in smoother prints with less artifacting. The firmware also enables higher stepper motor frequencies, reducing the likelihood of missed steps and enabling more precise movements at higher speeds. Additionally, because the motion planning is offloaded to the host computer, Klipper[12] can run more sophisticated algorithms, improving print quality even at higher speeds. While Klipper[12] offers several advantages, it also introduces some challenges. The most significant of these is the need for a host computer, such as a Raspberry Pi, to run the firmware. This adds an extra layer of complexity and cost compared to traditional 3D printer firmware, which runs directly on the printer's microcontroller. Setting up Klipper[12] requires additional hardware and software configuration, which may be intimidating for users new to 3D printing or those who prefer simpler plug-and-play solutions. However, for those willing to invest the time to set up the system, Klipper[12] provides significant benefits in terms of speed, quality, and flexibility.

6 Conclusion and future work

Conclusion Xtrooder introduces a modern, Rust-based firmware solution for 3D printers, focusing on safety, modularity, and performance. The current implementation achieves critical functionality, including linear and accelerated motion control for X, Y, Z, and E axes, SD card support for G-code storage and execution, filament recovery and retraction, and precise PID-based thermal management for the hotend and heated bed. Built with the Embassy framework, Xtrooder leverages its high-level abstractions and Hardware Abstraction Layers (HALs) to manage STM32 microcontroller peripherals efficiently, enabling predictable, real-time performance with minimal overhead. This approach demonstrates the effectiveness of combining modern, memory-safe programming languages like Rust with state-of-the-art frameworks for embedded systems.

By using Embassy's async programming model, Xtrooder maximizes concurrency while ensuring deterministic task scheduling, critical for time-sensitive operations in 3D printing. These features make the firmware highly efficient and robust, even on resource-constrained hardware. However, Xtrooder's current limitation to STM32 microcontrollers, while allowing for streamlined and optimized implementation, restricts its adaptability and limits its user base compared to competitors like Marlin[14], which supports a wide range of platforms.

Future Work To broaden its impact and appeal, future development of Xtrooder will focus on expanding hardware support, functionality, and usability. Extending compatibility beyond STM32 microcontrollers is a critical next step. Integrating additional HALs from the Embassy framework and developing custom abstractions for popular platforms like AVR and other ARM architectures will make XTrooder accessible to a broader audience. This would position the firmware as a versatile solution for various 3D printer configurations and hardware ecosystems.

Enhancing motion control capabilities is another key priority. Support for advanced kinematics such as CoreXY, Delta, and SCARA would significantly increase the firmware's applicability to a wider range of printer designs. Implementing features like input shaping to reduce vibration and pressure advance to optimize extrusion during high-speed motion would elevate print quality and enable XTrooder to compete with performance-driven firmware like Klipper[12].

Peripheral support is essential for modern 3D printers. Adding compatibility for automatic bed leveling probes, filament runout sensors, and environmental monitoring devices would greatly improve usability and reliability. Multi-extruder support, enabling multi-material and multi-color printing, is another feature that could significantly expand XTrooder's capabilities. These enhancements will make the firmware more attractive to both hobbyists and professionals seeking advanced functionality.

Improving the user experience is equally important. Developing a user-friendly interface, either through integration with existing tools like OctoPrint or by creating a dedicated web-based solution, would simplify configuration and operation. Such interfaces could provide real-time printer monitoring, remote control capabilities, and an intuitive way to manage settings, making the firmware more accessible to less technical users.

Lastly, extensive testing and benchmarking are essential to validate XTrooder's performance against established firmware solutions. Rigorous validation across multiple printer models and configurations will help identify areas for optimization and ensure compatibility and stability. Performance profiling will also be critical

for optimizing resource usage, ensuring that additional features do not compromise the firmware's efficiency or responsiveness.

Final Thoughts XTrooder represents a forward-looking approach to 3D printer firmware, blending modern programming practices with a focus on safety, modularity, and performance. Its use of the Embassy framework and Rust's inherent safety features highlights the potential for adopting innovative technologies in embedded system development. While the current implementation has established a solid foundation, the outlined future work will be instrumental in transforming XTrooder into a competitive alternative to traditional firmware like Marlin[14] and Klipper[12]. By expanding its hardware support, enhancing features, and improving user accessibility, XTrooder can evolve into a robust and versatile solution for the ever-growing 3D printing community.

7 Bibliography

References

[1] Cartridge heaters and how they work. URL https://www.omega.com/en-us/resources/cartridge-heaters.

[2] Crate confy. URL https://docs.rs/confy/latest/confy/.

[3] Embassy - the next-generation framework for embedded applications. . URL https://embassy.dev/.

[4] Crate embassy-executor. . URL https://docs.rs/embassy-executor/latest/embassy_executor/index.html.

[5] Crate embassy-futures. . URL https://docs.rs/embassy-futures/latest/embassy_futures/.

[6] Crate embassy-stm32. . URL https://docs.embassy.dev/embassy-stm32/git/stm32h753zi/index.html.

[7] Crate embassy-sync. . URL https://docs.embassy.dev/embassy-sync/git/default/index.html.

[8] Crate embassy-time. . URL https://docs.rs/embassy-time/latest/embassy_time/.

[9] Crate embedded-sdmmc. . URL https://docs.rs/embedded-sdmmc/latest/embedded_sdmmc/.

[10] Crate heapless. URL https://docs.rs/heapless/latest/heapless/.

[11] Hot end in 3d printers. URL https://www.3deasy.it/wp/2020/01/16/hotend/.

[12] Klipper3d firmware. . URL https://www.klipper3d.org/.

[13] Klipper3d - github page. . URL https://github.com/Klipper3d/klipper.

[14] Marlin firmware. . URL https://marlinfw.org/docs/basics/introduction.html.

[15] G-code from marlin. . URL https://marlinfw.org/meta/gcode/.

[16] Marlin - github page. . URL https://github.com/MarlinFirmware/Marlin.

[17] Crate measurements. URL https://docs.rs/measurements/latest/measurements/.

[18] Crate micromath. URL https://docs.rs/micromath/latest/micromath/.

[19] Platformio. URL https://platformio.org/.

[20] Crate quote. URL https://docs.rs/confy/latest/quote/.

[21] 3d printing applications: 12 industries and examples. URL https://www.raise3d.com/blog/3d-printing-applications/.

[22] G-code from reprap wiki. URL https://reprap.org/wiki/G-code.

[23] Rtic - a rust rtos. URL https://rtic.rs/2/book/en/rtic_and_embassy.html.

[24] Asynchronous programming in rust. URL https://rust-lang.github.io/async-book/.

[25] Crate serde. URL https://docs.rs/serde/latest/serde/.

[26] Crate syn. URL https://docs.rs/confy/latest/syn/.

[27] Thermistors and how they work. URL https://www.electricaltechnology.org/2021/11/thermistor.html.

[28] Tokio - an asynchronous rust runtime. URL https://tokio.rs/.

[29] Dejan. Drive a stepper motor using arduino. URL https://howtomechatronics.com/tutorials/arduino/stepper-motors-and-arduino-the-ultimate-guide/.

[30] Carmine Fiore. Stepper motors basics: Types, uses, and working principles. URL https://www.monolithicpower.com/learning/resources/stepper-motors-basics-types-uses.

[31] Omar Hiari. Sharing data amount rust embassy tasks. . URL https://dev.to/theembeddedrustacean/sharing-data-among-tasks-in-rust-embassy-synchronization-primitives-59hk/.

[32] Omar Hiari. Comparison between embassy and freertos - async rust vs. freertos. . URL https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown.

[33] Omar Hiari. Pwm generation using embassy. . URL https://dev.to/theembeddedrustacean/embedded-rust-embassy-pwm-generation-15cf.

[34] Jobit Joseph. Interfacing thermistor with arduino. URL https://circuitdigest.com/microcontroller-projects/interfacing-Thermistor-with-arduino.

[35] Murray Todd Williams. Read sensor data using embassy. URL https://murraytodd.medium.com/using-rust-embedded-to-capture-sensor-data-37db1f726d5c.