Summary

1. Introduction

This user manual describes StepperHAL, a C++ library designed as a Hardware Abstraction Layer (HAL) for controlling stepper motors on the STM32F4x1 family of microcontrollers. The library was developed to simplify motion management, offering an intuitive and configurable interface that integrates with the Arduino framework and the STMicroelectronics HAL.

1.1 Key Features

- **Flexible control**: StepperHAL offers the ability to manage motor movement at both constant speed and with acceleration and deceleration profiles.
- **Motion profiles**: The library supports trapezoidal and S-curve motion profiles for smooth and precise movements. These features can be enabled or disabled by changing the configuration settings.
- **Extensive configuration**: Motor configuration and pin mapping can be managed centrally, allowing you to define control pins and motor parameters such as steps per revolution, microsteps, and millimeters per revolution.
- **Debugging and diagnostics**: An integrated debug system is included, which provides detailed messages on the serial monitor for monitoring and troubleshooting.

1.2 Requirements

The StepperHAL library requires a specific development environment and hardware to function correctly.

Software Requirements

- **Arduino framework for STM32**: The library is based on this framework for peripheral management (Arduino_Core_STM32 (stm32duino)).
- **IDE (Integrated Development Environment)**: Arduino IDE or others that support the Arduino framework, such as Visual Studio Code with the PlatformIO extension, are recommended.

Hardware Requirements

- **STM32F4x1 Microcontroller**: The library's architecture is optimized for this specific family of MCUs, although it may work with minor modifications on other STM32 microcontrollers.
- **Stepper motor driver**: A driver (e.g., A4988, DRV8825) is required to drive the motors, connected to the microcontroller pins defined in StepperHAL_Config.h.
- **Stepper motors**: Supports NEMA17, NEMA23, or similar motors.
- **Development board**: A board based on STM32F4x1, such as the "Black Pill" or a Nucleo.

2. Configuration

The StepperHAL configuration chapter guides you through customizing the library to suit your project. All configuration options are located in the file StepperHAL_Config.h.

2.1 Feature flags

"Feature flags" allow you to enable or disable specific library features during compilation. This helps to optimize code size and performance by including only what you need.
You can enable or disable features by modifying the following macros in the StepperHAL_Config.h file:

- **USE_S_CURVE**: Enables S-curve motion profiles, which offer smoother acceleration and deceleration. To activate it, set to 1.
- USE_TRAPEZOIDAL: Enables trapezoidal motion profiles, which are simpler and suitable for fast movements. To activate it, set to 1.

  Note: If both motion profiles are disabled (0), the library will automatically enable the trapezoidal profile as a fallback option.

2.2 Pin mapping

This section of the StepperHAL_Config.h file is essential for defining the connections between the microcontroller and the motor drivers.
Define the pins for each motor using the dedicated macros. For example:

```
#define PIN_DIR_MOTOR1  PA_9
#define PIN_EN_MOTOR1   PA_8
```

- **PIN_DIR_MOTORx**: The pin connected to the DIR (direction) input of the driver.
- PIN_EN_MOTORx: The pin connected to the EN (enable) input of the driver. If an enable pin is not used (the driver is always on), set it to -1. It can be shared with other EN pins. It's important to note that the STEP pin (_stepPin) is not defined in this section, but is automatically determined by the library based on the timer and channel chosen for each motor. This is because the generation of the square wave for step control is managed via hardware by the timer, which is mapped to a specific microcontroller pin.

Timer	**Channel**	**STEP PIN**
TIM2	CH1	PA_15
TIM3	CH2	PB_5
TIM4	CH1	PB_6

TIM5	CH3	PA_2

Default timer, channel, and STEP pin; they can be changed but it is not recommended.

2.3 Constructor parameters

The constructor parameters, also defined in StepperHAL_Config.h, allow you to configure the physical and mechanical properties of each motor.
The format for each motor is as follows:
#define MOTORx_PARAMS (timer, channel, dirPin, enPin, stepsPerRev, microstep, mmPerRev)

- **timer**: The STM32 hardware timer to use (e.g., TIM2, TIM3).
- **channel**: The timer channel.
- **dirPin**: The direction pin previously defined.
- **enPin**: The enable pin previously defined.
- **stepsPerRev**: The number of physical steps of the motor for each full revolution (e.g., 200).
- **microstep**: The microstepping value of the driver (e.g., 1 for full step, 256 for 1/256 step).
- **mmPerRev**: The millimeters per revolution; screw pitch, drum or pulley circumference. If a linear system is not used, set this value to 1.0f.

Alternatively, you can use the following format:
#define MOTORx_PARAMS (timer, channel, dirPin, enPin, effectiveStepsMotorRev, mmPerRev)

- **effectiveStepsMotorRev**: The total steps per revolution value assigned by the driver (e.g., 1000).

2.4 Global instances

To simplify usage and ensure a single source of truth, the library defines global instances of the StepperHAL objects for each motor. You do not need to manually create new StepperHAL objects in your main code (.ino).
These instances are declared in the StepperHAL_Instances.h file and defined in the StepperHAL_Config.cpp file, using the parameters you set in the previous chapter (do not modify).
Example declaration (StepperHAL_Instances.h):
extern StepperHAL motor1;
extern StepperHAL motor2;
extern StepperHAL motor3;
extern StepperHAL motor4;

Example definition (**StepperHAL_Config.cpp**):

```
#include "StepperHAL_Config.h"
#include "StepperHAL_Instances.h"
#include "StepperHAL_STM32F4x1.h"

StepperHAL motor1(MOTOR1_PARAMS);
StepperHAL motor2(MOTOR2_PARAMS);
StepperHAL motor3(MOTOR3_PARAMS);
StepperHAL motor4(MOTOR4_PARAMS);
```

How to use them:
In your Arduino sketch, you can simply include the StepperHAL_Instances.h file to access these instances and call the library methods. No additional object is required.
For example, to initialize motor 1, your code will be:

```
#include "StepperHAL_Instances.h"

void setup() {
  motor1.begin();
  // ...
}
```

This centralized approach makes the code cleaner and more manageable, avoiding potential conflicts or resource duplication.

3. Initialization

Before you can use the motors, you need to initialize the library and configure the hardware peripherals. This is done with the begin() function.

What the begin() function does:

- **Pin configuration**: Sets the direction and enable pins as outputs.
- **Timer configuration**: Configures the STM32 hardware timer chosen in **StepperHAL_Config.h** for generating the PWM signal needed to drive the motor.
- **Driver enablement**: Enables the motor driver via the enable pin (if specified).
- **Initial direction setting**: Sets the motor's direction to a default value.
- Interrupt enablement: Enables the timer interrupt. This operation is crucial, as step advancement is handled within the interrupt service routine (ISR) to ensure precision and timing.
  If the EN (enable) pin is used, the motorX.setEnable() function must be called in setup() and set to true.

How to call begin():
The begin() function must be called in the setup() alongside setEnable() of your Arduino sketch for each motor you want to use (if the motor driver uses a pin for EN).

```
#include "StepperHAL_Instances.h"

void setup() {
  // Initialize serial communication for debugging
  Serial.begin(115200);

  // Initialize all motors you use in your project
  motor1.begin();
  motor1.setEnable(true);
  // ...
}

void loop() {
  // Your code to control the motors goes here
}
```

4. Movement API

The StepperHAL library provides a complete set of functions to control the motors, accepting different units of measurement and managing both constant velocity and acceleration profile movements.

4.1 Data types for parameters

The movement methods use dedicated data types to specify the unit of measurement:

- **step**: The number of steps or microsteps.
- **mm**: The distance in millimeters.
- **Speed**: The speed in revolutions per minute (RPM).
- **Feed**: The speed in mm/min.
- mms2: The acceleration or deceleration in mm/s^2. These types are defined as struct for explicit unit of measurement, making the code more readable. For example, to create a value of type mm, you use mm(10.0f).

4,2 Constant velocity movement

These methods move the motor at a fixed speed, without using acceleration profiles. They are useful for simple movements or calibration, where controlled acceleration is not required.

Absolute movement (go to a position)

- void moveToPosition(mm distance, Speed rpm): Moves the motor to an absolute position in mm at a speed in RPM.
- void moveToPosition(mm distance, Feed feedRate): Moves the motor to an absolute position in mm at a feed rate in mm/min.
- void moveToPosition(step steps, Speed rpm): Moves the motor to an absolute position in steps at a speed in RPM.
- void moveToPosition(step steps, Feed feedRate): Moves the motor to an absolute position in steps at a feed rate in mm/min.

Relative movement (move from here)

- void moveRelative(mm delta, Speed rpm): Moves the motor by a relative distance in mm at a speed in RPM.
- void moveRelative(mm delta, Feed mmPerMin): Moves the motor by a relative distance in mm at a feed rate in mm/min.
- void moveRelative(step delta, Speed rpm): Moves the motor by a relative number of steps at a speed in RPM.
- void moveRelative(step delta, Feed mmPerMin): Moves the motor by a relative number of steps at a feed rate in mm/min.

4.3 Movement with acceleration profiles

These methods, available if USE_TRAPEZOIDAL or USE_S_CURVE is enabled, use acceleration and deceleration profiles to ensure smooth movements, reducing vibrations and step loss.

Absolute movement (go to a position)

- void moveToPositionWithAccel(mm distance, Speed rpm, mms2 accel): Moves the motor to an absolute position in mm with a constant acceleration up to a cruise speed in RPM.
- void moveToPositionWithAccel(mm distance, Feed feedRate, mms2 accel): Similar to the previous one, but uses the feed rate in mm/min.
- void moveToPositionWithAccel(step steps, Speed rpm, mms2 accel): Moves the motor to an absolute position in steps with acceleration and speed in RPM.
- void moveToPositionWithAccel(step steps, Feed feedRate, mms2 accel): Moves the motor to an absolute position in steps with acceleration and feed rate.

Variants with separate acceleration and deceleration
These methods offer more precise control, allowing you to set different values for acceleration and deceleration. They are ideal for applications that require asymmetric motion profiles, such as when motor torque changes based on load or direction.

- void moveToPositionWithAccel(mm distance, Speed rpm, mms2 accel, mms2 decel): Moves the motor to an absolute position in mm with a motion profile that uses separate acceleration (accel) and deceleration (decel), up to a cruise speed in RPM.
- void moveToPositionWithAccel(mm distance, Feed feedRate, mms2 accel, mms2 decel): Similar to the previous one, but uses a feed rate in mm/min for the cruise phase.
- void moveToPositionWithAccel(step steps, Speed rpm, mms2 accel, mms2 decel): Moves the motor to an absolute position in steps with separate acceleration and deceleration profiles, up to a speed in RPM.
- void moveToPositionWithAccel(step steps, Feed feedRate, mms2 accel, mms2 decel): Moves the motor to an absolute position in steps with separate acceleration and deceleration profiles, up to a feed rate.

Relative movement (move from here) with acceleration
These methods move the motor by a specified number of steps or distance relative to the current position, using acceleration and deceleration profiles. They are ideal for making precise and smooth movements from one point to another.
Variants with single acceleration
These methods use the same acceleration value for the ramp-up and deceleration phases.

- void moveRelativeWithAccel(mm delta, Speed rpm, mms2 accel): Moves the motor by a relative distance in mm at a speed in RPM, using the specified acceleration profile.
- void moveRelativeWithAccel(mm delta, Feed feedRate, mms2 accel): Moves the motor by a relative distance in mm at a feed rate in mm/min.
- void moveRelativeWithAccel(step delta, Speed rpm, mms2 accel): Moves the motor by a relative number of steps at a speed in RPM.
- void moveRelativeWithAccel(step delta, Feed feedRate, mms2 accel): Moves the motor by a relative number of steps at a feed rate in mm/min.

Variants with separate acceleration and deceleration
These methods allow you to set different values for acceleration and deceleration.

- void moveRelativeWithAccel(mm delta, Speed rpm, mms2 accel, mms2 decel): Moves the motor by a relative distance in mm with separate acceleration and deceleration profiles, up to a speed in RPM.
- void moveRelativeWithAccel(mm delta, Feed feedRate, mms2 accel, mms2 decel): Moves the motor by a relative distance in mm with separate acceleration and deceleration profiles, up to a feed rate in mm/min.
- void moveRelativeWithAccel(step delta, Speed rpm, mms2 accel, mms2 decel): Moves the motor by a relative number of steps with separate acceleration and deceleration profiles.
- void moveRelativeWithAccel(step delta, Feed feedRate, mms2 accel, mms2 decel): Moves the motor by a relative number of steps with separate acceleration and deceleration profiles, up to a feed rate.

4.4 Interrupting a movement

- void stop(): This command **immediately stops** the motor, interrupting any ongoing movement. Unlike an operation that waits for a position to be reached, stop() forces an instant interruption of the movement, which can be useful for managing emergency situations or for immediate manual control.

4.5 Managing the S-Curve profile

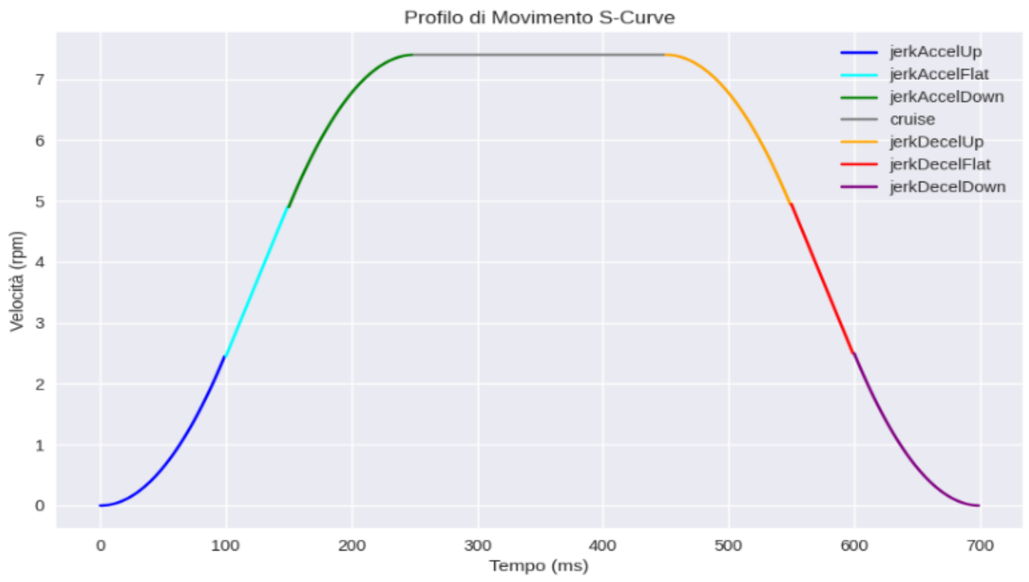
These methods are only available if the USE_S_CURVE profile is enabled. They allow you to customize the "S" acceleration curve that controls the variation of acceleration and deceleration to achieve smoother movements, reducing vibrations and mechanical shock.

What is Jerk?
The Jerk is the rate of change of acceleration over time. In a standard motion profile (trapezoidal), the acceleration changes instantaneously from zero to a constant value, creating a "jolt" or "jerk" at the beginning and end of the movement. The S-curve profile eliminates this abrupt change by managing acceleration gradually. A high jerk corresponds to a more sudden change in acceleration, while a low jerk ensures a softer and more progressive movement.

- void useSCurve(bool enable): Enables or disables the use of the S-curve profile.
- void useDefaultJerk(bool enable): Uses the predefined calculation models to automate jerk management, creating a balanced S-curve profile, enabled by default.
- void setJerkRatio(float ratio): Sets a ratio that modifies the balance between acceleration and deceleration when using the predefined calculation models. It only takes effect when using the useDefaultJerk(true) method.
- void setManualJerk(...): This method allows for advanced jerk customization. It offers precise control over each of the six phases of the S-curve profile, allowing you to adjust the acceleration and deceleration trend. The six parameters correspond to the following

phases of movement:

- **jerkAccelUp**: The beginning of the acceleration ramp, when the motor starts to ramp up speed.
- **jerkAccelFlat**: The middle part of the acceleration phase, where acceleration is constant.
- **jerkAccelDown**: The end of the acceleration ramp, when the motor approaches the cruise speed.
- **jerkDecelUp**: The beginning of the deceleration ramp, when the speed starts to decrease.
- **jerkDecelFlat**: The middle part of the deceleration phase.
- **jerkDecelDown:** The end of the deceleration ramp, when the motor comes to a complete stop softly.
  It is recommended to start with an automatic configuration (useDefaultJerk(true)) to understand the motor's behavior. Subsequently, you can enable debugging (StepperHAL_DEBUG_ENABLE to true) to view the jerk values in the log, and then gradually vary the individual parameters via setManualJerk and observe how the changes influence the movement.



4.6 Visualization of the S-Curve

The S-Curve motion profile is distinguished by its ability to make movement smoother through the gradual management of acceleration and deceleration. This profile is divided into seven phases, six of which are the jerk phases.

As can be seen from the graph:

Acceleration Jerk (phase 1-3): The speed increases in a controlled manner. The "jerk" (the rate of change of acceleration) is gradual:

- *jerkAccelUp:* The acceleration increases from zero to its maximum value.
- *jerkAccelFlat:* The acceleration remains constant.
- jerkAccelDown: The acceleration decreases to zero.

Cruise Speed (phase 4): The motor moves at a constant speed with no acceleration or deceleration.

Deceleration Jerk (phase 5-7): The speed decreases in a controlled manner. Here too, the jerk is gradual:

- *jerkDecelUp:* The deceleration increases from zero to its maximum value.
- *jerkDecelFlat:* The deceleration remains constant.
- *jerkDecelDown:* The deceleration decreases to zero, bringing the motor to a gentle stop.

5. Configuration and status API

These methods allow you to configure the motor and check its status in real time.

5.1 Managing minimum speed

These methods allow you to define and retrieve the minimum speed at which the motor can move. Setting a minimum speed is crucial to prevent the motor from stalling at too low frequencies.

- void setMinFreqHz(float freqHz): Sets the minimum step frequency directly in Hertz. It is used by setMinRPM() to convert the value into RPM and ensure that the frequency never drops below a minimum value, preventing stalls at speeds close to zero. Default values are in **StepperHAL_Config.h**.
- void setMinRPM(float rpm): This method sets the motor's minimum speed in revolutions per minute (RPM). The library internally calculates the equivalent frequency in Hertz (Hz) and stores it. During any movement, the motor speed will never drop below this value, even if the set cruise speed is lower.
- float getMinFreqHz() const: Returns the minimum step frequency in Hertz (steps per second) that was calculated and set internally via setMinRPM(). It is a useful method for checking the minimum frequency value.

5.2 Movement support methods

These methods do not initiate a movement, but allow you to manage its behavior, compensation, or interruption.

- void invertDIR(bool state): Inverts the motor's direction.
- void setTiming(uint32_t stepPulseUs, uint32_t timerBaseFreq): Initializes the internal _timing structure with the duration of the STEP signal and the timer clock frequency, then calculates the real timer parameters (prescaler, auto-reload, counters) necessary to generate precise step pulses.
  - **stepPulseUs (uint32_t)** • The minimum duration of each STEP pulse in microseconds (µs).
  - **timerBaseFreq (uint32_t)** • The frequency in Hertz (Hz) of the peripheral timer clock used to generate STEP.
  - Call immediately after begin(). It must be executed before any movement command (moveTo...) to ensure the timer timing values are correct.
- void backlashCompensation(step value): Sets a backlash value in steps.
- void backlashCompensation(mm value): Sets a backlash value in millimeters. Backlash is the mechanical play that occurs during the motor's direction reversal. This value is used to automatically compensate for movement to ensure precise positioning. To manage backlash, the StepperHAL library maintains two internal positions: a logical position and a real position.
- **The logical position** is the ideal position the motor should have.

- The real position is the physical position of the motor, which may differ from the logical position.
  If no compensation is set (backlashCompensation(0)), the two positions coincide. When you set a backlash value (in mm or steps), the library automatically accounts for this play. The real position will differ from the logical position by the steps necessary for compensation. Each time the motor reverses its direction, StepperHAL will apply the compensation. This happens automatically, without you having to intervene further in the code. The discrepancy between the real and logical position occurs always and only after a movement in the opposite direction to the first direction used (e.g., if the first movement is CW, the discrepancies will be after a CCW movement and vice versa).

5.3 Position management

These methods allow you to define or reset the motor's position, a fundamental operation for calibration and correct functioning. Their main difference lies in how they affect the internal backlash compensation state.

- void resetPosition(): Resets the motor's current position to zero. It is useful for establishing a new reference point (home) without having to physically move the motor. This method also resets the internal backlash state. This means that, after this function call, the motor's logical and real positions will coincide again, and the backlash compensation mechanism will restart from zero.
- void setPositionSteps(int32_t steps): Sets the motor's current position to a specific value in steps.
- void setPositionMM(float mm): Sets the motor's current position to a specific value in millimeters.
  These methods are useful for calibrations, for example, to set the initial position to a value other than zero. Unlike resetPosition(), these methods do not alter the backlash state. If compensation has already been applied, it will remain active, ensuring that the real position continues to account for mechanical play and the last direction of movement.

5.4 Status and query methods

This section describes the methods that allow you to query the motor's current status and retrieve information for motion control.

- bool isActive(): Returns true if the motor is moving, false otherwise. This method is useful for checking the motor's status, for example, before starting a new movement or to wait for the current movement to be completed.
- bool hasBacklash(): Returns true if a backlash value has been set using the backlashCompensation() methods. It's a simple verification method to know if play compensation is active or not.
- int32_t getPositionSteps(): Returns the current position in steps. This value corresponds to the logical position, which is the motor's ideal position, not yet corrected for mechanical

play.

- float getPositionMM(): Returns the current position in millimeters. This value also corresponds to the logical position, converted to millimeters. It is a way to read the motor's position in a more intuitive unit of measurement.
- uint32_t getStepsPerRev(): Returns the number of physical steps for each revolution of the motor. This value is a basic parameter that is set in the constructor and does not change during operation.
- bool targetReached() const: This method checks if the motor has reached the end point of the desired movement. It returns true if the current position matches the effective destination position and false otherwise. It is a fundamental method to use in a waiting loop to know when a movement has ended. The effective destination position already accounts for internal compensations, such as backlash, ensuring a precise check.

6. Debugging and diagnostics

The StepperHAL library includes an integrated debug system that provides detailed real-time information about the motor's behavior. It is an essential tool for calibration, optimization, and troubleshooting problems such as stalling or inaccurate movements.

6.1 Enabling debug

The debug system is controlled by the directive #define StepperHAL_DEBUG_ENABLE that is in the StepperHAL_Debug.h file.
To enable debugging, ensure that the line is set to true:
#define StepperHAL_DEBUG_ENABLE true
To disable debugging and optimize performance, set the value to false. When debugging is disabled, the code does not generate any output and there is no unnecessary overhead.

6.2 Interpreting the logs

After enabling debugging, information is printed to the serial terminal. Depending on the motion profile used (S-curve or trapezoidal), the log will provide specific details.

Example log (S-Curve profile)
This log shows the details of a movement with S-acceleration.

```
[StepperHAL] (motor2) Jerk mode: default
[StepperHAL] (motor2) Movement started from logical position=0 : real=0 / target=6000 /
steps needed=6000 / Dir=CW
[StepperHAL] (motor2) Jerk default — aUp=23.2 aFlat=0.0 aDown=23.2 dUp=23.2 dFlat=0.0
dDown=23.2 JerkRatio=1.00
[StepperHAL] (motor2)   S-curve phases acceleration (Up/Flat/Down)= 592/594/592
[StepperHAL] (motor2)   S-curve phases deceleration (Up/Flat/Down)= 592/594/592
[StepperHAL] (motor2) Separate DMA profile -> 6000 steps, accel = 100.00 mm/s², decel =
100.00 mm/s²
[StepperHAL] (motor2)   -> Acceleration: 1778
[StepperHAL] (motor2)   -> Cruise:        2444
[StepperHAL] (motor2)   -> Deceleration: 1778
[StepperHAL] (motor2)   -> Profile: S-curve
[StepperHAL] (motor2) Separate DMA -> Requested target=0 Vmax 800.00 RPM@ 13333.33
Hz
[StepperHAL] (motor2) Movement completed: stopped at logical position=6000: real=6000 /
target=6000 / Dir=CW
```

- Jerk mode: default: The library is using the default jerk values.
- Jerk default — ...: The calculated jerk values for the six phases of the S-profile are displayed.

- S-curve phases acceleration/deceleration: Indicate the number of steps in each of the three sub-phases (Up, Flat, Down) of the acceleration and deceleration ramp.
- Acceleration / Cruise / Deceleration: Indicate the total number of steps dedicated to each phase of the movement (acceleration ramp, constant cruise speed, deceleration ramp). The sum of these values gives the total number of steps.
- Vmax 800.00 RPM@ 13333.33 Hz: Summarizes the maximum speed reached during the movement, in both RPM and step frequency.

Example log (trapezoidal profile)
This log shows a movement with a trapezoidal profile, which is simpler and does not include the jerk phases.

```
[StepperHAL] (motor2) Movement started from logical position=10000 : real=10000 /
target=0 / steps needed=10000 / Dir=CCW
[StepperHAL] (motor2) Separate DMA profile -> 10000 steps, accel = 100.00 mm/s², decel =
100.00 mm/s²
[StepperHAL] (motor2)   -> Acceleration: 1000 steps
[StepperHAL] (motor2)   -> Cruise: 8000 steps
[StepperHAL] (motor2)   -> Deceleration: 1000 steps
[StepperHAL] (motor2)   -> Profile: trapezoidal
[StepperHAL] (motor2) Separate DMA -> Requested target=0 Vmax 600.00 RPM@
10000.00 Hz
[StepperHAL] (motor2) Movement completed: stopped at logical position=0: real=0 /
target=0 / Dir=CCW
```

- Acceleration / Cruise / Deceleration: Here too, the number of steps for each phase of the movement is indicated. The sum of these values allows you to verify the correctness of the profile calculation.
- Vmax 600.00 RPM@ 10000.00 Hz: Summarizes the maximum speed reached. The debug system is a powerful tool for calibration, especially when optimizing motion profiles to minimize vibrations and step loss.

7. Language selection

The StepperHAL library offers the option to display debug messages in different languages, a useful option for better readability based on user preferences. This functionality is managed by a simple directive in the StepperHAL_Lang.h file.

7.1 How to change the language

To change the language of debug messages, simply modify the following line in the StepperHAL_Lang.h file:

```
#define LANGUAGE_IT // Comment for English
```

To enable the Italian language, make sure the #define LANGUAGE_IT line is not commented out.

To switch to the English language, comment out the line by adding // at the beginning:

```
//#define LANGUAGE_IT // Comment for English
```

This compile-time approach ensures that the code is lightweight and that the performance impact is minimal, as messages are included only in the selected language.

7.2 The msg() function

The localization logic is managed internally by the msg() function, which takes two strings as parameters: one for the Italian version and one for the English version.

```
inline String msg(const String& it, const String& en) {
#ifdef LANGUAGE_IT
  (void)en;  // avoids unused parameter warning when LANGUAGE_IT is defined
  return it;
#else
  (void)it;  // avoids unused parameter warning when LANGUAGE_IT is not defined
  return en;
#endif
}
```

This function is used everywhere in the library to print debug messages. For example, in the StepperHAL_STM32F4x1.cpp file you will find calls similar to this:

```
SerialDB.print(msg("Movimento completato: fermo...", "Movement completed: stopped..."));
```

Based on the value of LANGUAGE_IT, the msg() function will return the string in Italian or English. This ensures that all output messages are consistent with the chosen language, without you having to modify the library code.

8. Warning (hal_conf_extra.h)

The hal_conf_extra.h file is required to explicitly enable or disable HAL modules without modifying the internal files of the STM32 Arduino core. In the case of StepperHAL, it is essential to disable the core's HAL_TIM module, otherwise a conflict occurs with the ISR functions (TIMx_IRQHandler) already defined by StepperHAL.

In fact, the STM32 Arduino core includes a file called HardwareTimer.cpp that defines the same ISR functions for hardware timers (TIM2, TIM3, TIM4, etc.). If both definitions are present, the linker throws a "multiple definition" error.

To avoid this conflict, inside the directory where the library is installed, there is a hal_conf_extra.h file containing:

cpp
```
#define HAL_TIM_MODULE_ONLY
```

this macro disables the core's HAL_TIM module, leaving only StepperHAL's definitions active.