AN6179

Application note

How to integrate the STL firmware into a time critical user application

Introduction

This document highlights some problematic points and provides tips on how to integrate ST safety-oriented libraries into an application handling a time-critical control. Although the integration examples in this document are based on the X-CUBE-CLASSB firmware for home appliances, the basic principles are also valid for X-CUBE-STL dedicated to industrial applications.

1 General information

Table 1. Referenced document

N°	Description
[1]	STM32G4 series UL/CSA/IEC 60730-1/60335-1 self-test library user manual (UM3167)

2 Overview

The latest versions of STMicroelectronics safety firmware packages use software self-test modules focused on generic parts of the microcontroller. These modules must be supplemented by additional application-specific tests provided by the end user, dedicated to the microcontroller peripherals involved in the safety tasks and depending on the specific application. The most efficient application-specific testing is achieved when supported by proper measurements implemented in the application hardware design. Examples include reference levels, comparison of redundant I/Os, and verification loopbacks. The STM32 family safety manuals and STL-associated documentation provide numerous methods and knowledge to manage this user-specific development.

Safety standards require the repetition of the entire STL testing cycle at regular intervals during the application runtime. When the application runtime is not extremely long, large memories are often tested in their entirety once at every application startup and step-by-step in parts during runtime, or focused on the most critical data to make the testing cycle shorter. The frequency of the self-test cycle repetition depends on the available application process safety time. The goal is to ensure the required safety level and system reliability by applying an efficient diagnostic system capable of discovering faults and errors that could potentially lead to dangerous system behavior before they propagate into a serious system failure. The portion of these errors covered by the STL diagnostic determines the system safety integrity level.

Users must understand that the coverage of these STL tests executed by software is limited due to the impossibility of detecting a significant portion of so-called transient errors with short lifetimes. Therefore, it is practically impossible to achieve higher safety integrity levels by relying solely on such relatively slow software testing. To achieve higher safety integrity levels, users or electronic producers must apply specific hardware structures in the application design or in the design of its components based on redundancy principles, such as ECC on memories, dual channels with comparators, and voters.

3 Interference of STL running at application

The software-based testing cycle flow is compiled from a sequence of tests programmed and configured via STL scheduler services. This STL flow must coexist with other software handling the application flow, and any unexpected interference between STL and application software must be suppressed. Most of these constraints are easily manageable if the user respects the instructions provided in the STL user guides. This document focuses on demanding cases, where the interrupt masking window applied during execution of specific test modules such as RAM and TM7 ones at X-CUBE-CLASSB STL providing destructive operations upon the memory content (see more details at chapter Interrupt management at STL user guides). This masking would interfere with the application's interrupts sensitive to low latency or jitter. Adopting proper protection mechanisms is necessary unless the user fully takes responsibility and care of these collisions, which is not trivial. This can raise significant constraints and obstacles in systems with limited performance, especially when handling time-critical processes.

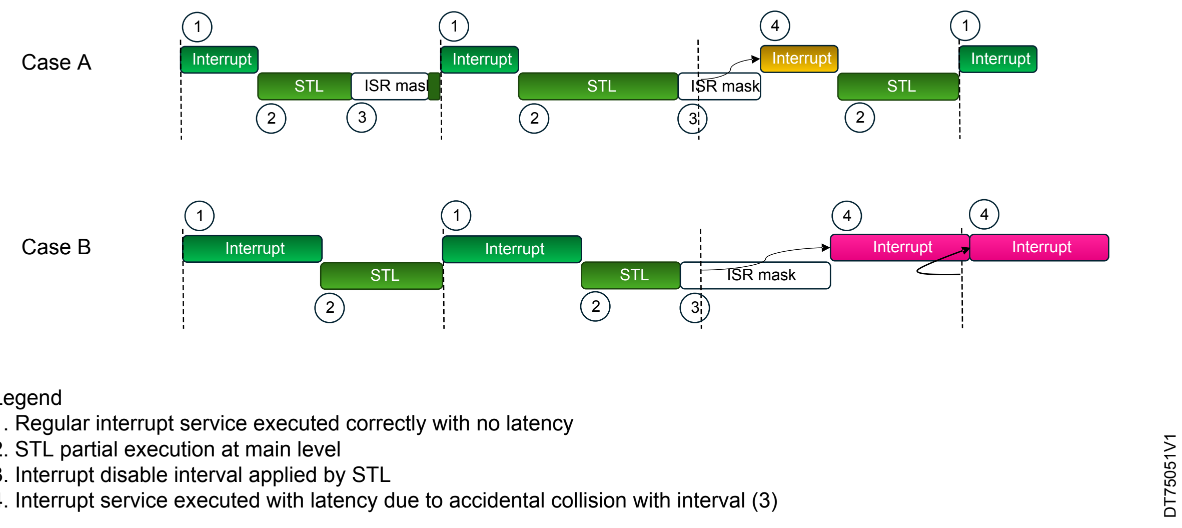
Commonly, the STL flow can be executed as the lowest priority level task, allowing it to be interrupted or suspended by a higher priority task, including the RTOS task switcher. If the maximum possible interrupt service latency is still acceptable, the application can still run correctly. This is the case in **case A** timing scheme shown in the upper part of Figure 1, which explains the effect of interrupt masking performed by the STL when the application executes regular interrupt services without applying any compensation method. Some services, like the yellow one in Figure 1, can be executed with latency potentially equal to the maximum STL interrupt masking time in the worst case. This can be acceptable if such latency does not impose significant constraints and does not lead to system failure or malfunction in the end. Note that Figure 1 explains the effect of interrupt masking performed by the STL when the application executes regular interrupt services without applying any latency-compensating method.

Conversely, when system interrupt services are critical and must be executed regularly with minimal or no latency, executing the STL without additional control can result in some critical services not being executed at the expected time slots, or even potentially causing their chaining. This can lead to serious application failures, as shown by the red services in the **case B** timing scheme in Figure 1.

A typical example is a motor control application where the motor control regulation (PWM) cycle must be executed very precisely within given equidistant intervals. The parameters of each cycle need to be precomputed and programmed in advance based on measurements and calculations done in the preceding cycle. The most critical case occurs when such an application operates near the system performance limits, and/or the maximum STL interrupt masking time consumes a significant part of the required control cycle duration.

The remaining sections of this document provide options for coping with these timing constraints in STL integration. A detailed time analysis and understanding of the sequence of all critical processes is a crucial step in applying an efficient solution.

Figure 1. Effect of interrupt masking performed by the STL



Legend
1. Regular interrupt service executed correctly with no latency
2. STL partial execution at main level
3. Interrupt disable interval applied by STL
4. Interrupt service executed with latency due to accidental collision with interval (3)

4 How to compensate effects of the STL interrupt masking

Few possible methods how to deal with the STL interrupt masking are provided in this section.

4.1 Assuming responsibility for all possible interferences and conflicts

This is the most challenging option when STL is compiled with the defined compilation parameter STL_ENABLE_IT, which suppresses all interrupt masking during STL execution. In such case, user needs to be very careful when executing modules that would normally apply the interrupt masking time. A possible solution can be to implement, for example, a RAM test on a part of the RAM that is currently not in use, while the application is temporarily redirected to use another part of the RAM that does not overlap with the tested area.

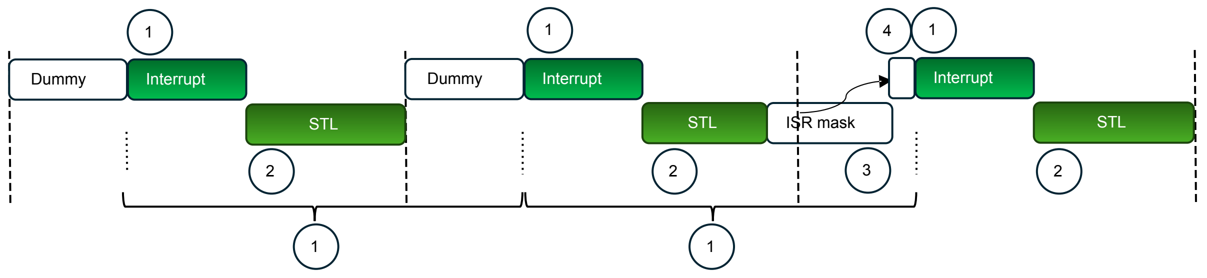
4.2 No concerns about interrupt latency

This option is applicable when the latencies introduced by STL are not critical to the system flow. This is illustrated as case A in Section 3. Users must perform a timing analysis while being fully aware of the possible impact of these latencies, including secondary aspects like chaining the service of the same source when a new pending request arises during the previous service execution, as shown in the case B timing scheme in Section 3. An acceptable solution can be the integration of the STL library into an RTOS, considering the possible latency of task switching as not critical. Usually, this option can be adopted for applications with significant performance overhead, where the latencies introduced by STL are commonly marginal compared to the execution periods and time frames dedicated to the implemented tasks, and no immediate reaction of the system is required.

4.3 Adaptable dummy Loop at start of critical interrupt

This option is suitable when a user needs a simple solution for an application process sensitive to latencies that must be executed at equidistant intervals with no (low) jitter. The implementation of this method is trivial, but the cost is performance loss at each entry to the service protected this way. The system waits in an adaptable synchronization dummy loop inserted at the beginning of the interrupt service, while preventing all the lower priority processes from running. The software loop simply reads the value of a timer raising the interrupt (or being triggered by a time-critical event, which is the source of the service) until it reaches a level corresponding to the dummy loop duration time. Then, the execution of the service is released. The level must be tuned to measure a longer interval than the maximum interrupt masking time produced by STL. The timing scheme is provided in Figure 2. Snapshot of the associated code modification is visible at bottom of the figure.

Figure 2. Compensating interrupt latency with adaptable dummy Loop



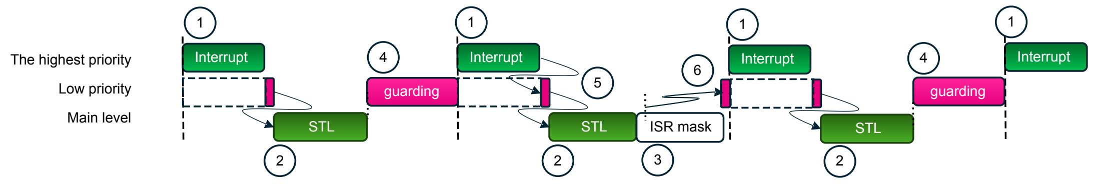
```
Void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim) {
If (htim->Instance == TIM1) {
   while (htim->Instance->CNT < GUARDING_LOOP_LENGTH)
    { }    // wait
    //  Here, do anything after jitter has been taken out
  }
}
```

Legend
1. Start of the regular interrupt service execution is postponed by purpose due to adaptable dummy loop inserted at begin of the service. Due to that compensation, the service is always executed at regular equidistant intervals.
2. STL partial execution at main level
3. Interrupt disable interval applied by STL
4. Dummy loop is adjusted to compensate interrupt service latency raised due to accidental collision of the interrupt with interval (3)

4.4 Guarding loop service protecting higher priority latency-critical interrupt entry

This option is more sophisticated and demanding, as the user needs to apply an additional specific lower priority interrupt service routine dedicated to the guarding loop. This interrupt must be periodically triggered in precise time slots to prevent the interrupt-blocking STL code from executing during critical time windows. The goal is to suspend the STL execution in advance to protect critical service entry, even if the STL applies the longest interrupt masking interval. The duration of the guarding loop must be tuned to be longer than the maximum STL interrupt masking interval. The timing-critical service must unblock the guarding loop to enable the next execution of the lowest main level collecting flow of the STL tests. The guarding loop service can execute application operations that do not apply any interrupt masking and involve detection of the guarding service end. Once detected, the guarding service terminates itself while normal runtime is resumed. The timing scheme is provided in Figure 3. This method is also implemented in the STL integration example associated with this application note.

Figure 3. Interrupt latency compensated by low-priority guarding loop



Legend
1. Regular critical interrupt service executed correctly with no latency
2. STL partial execution at main level
3. Interrupt disable interval applied by STL
4. Low level guarding interrupt service suspends STL execution with sufficient margin to protect critical process entry (1) and so prevents and compensates occasional interval (3) occurrence
5. End of the critical service releases the guarding loop interrupt and STL can continue at execution
6. Pending request raised by guarding service interrupt suspends the STL execution still in time just after interval (3) is completed

DT75053V1

5 Implementing latency-compensating guarding loop in motor control applications

This section demonstrates the integration of STL into a concrete motor control application, with a guarding loop compensating the latency of critical motor control interrupts. The example uses the P-NUCLEO-IHM03 pack, consisting of a NUCLEO-G431RB microcontroller board, an X-NUCLEO-IHM16M1 motor control expansion board, and a GBM2804H-100T motor. It showcases sensorless control of a PMDC motor using a three-shunt topology driver. Figure 4 and Figure 5 illustrate the basic topology of the applied switches and current measurement on the motor winding shunts, while Figure 6 and Figure 7 provide time schemes of the motor control process.

An efficient STL guarding loop implementation (see Figure 6) must be based on detailed timing analysis of the critical services within the motor control cycle. Three synchronized PWMs control the motor phase currents via associated output compare outputs, driven by a single timer configured in symmetrical center-aligned mode. See items 3 and 4 in Figure 6. The most time-critical service within the PWM period is the PWM computation procedure (6 in Figure 6), which calculates, plans, and preprograms all parameters and measurements for the next PWM cycle. Some parameters are propagated into the capture compare registers from their preprogrammed shadow registers at the timer update event, raised at the start of a new PWM cycle. This event also triggers a short service with the highest interrupt priority (7 in Figure 6) to apply new configuration of the ADC registers valid for newly starting cycle.

The PWM computation process uses the latest available ADC measurement results (5 in Figure 6) and must be completed before the start of the next PWM cycle. This ensures that the new cycle planning and configuration are done in time, benefiting from the applied hardware control associated with the timer update event. ADC conversion performs selected shunt voltage measurements, running in the background and usually starting in the middle of the PWM cycle. The specific shunts measured and the exact moment of their measurement are re-determined every period. In specific cases, the PWM computation procedure (6 in Figure 6) can plan different ADC conversion timings or specific shunt combinations for the next cycle to ensure the measurement of settled current values. The goal is to avoid measurements near expected switch changes, which can cause unstable voltage effects and false current measurements. In the 3-shunt topology, two motor phase currents are measured, and the third is calculated from these values, based on the principle that the sum of all stabilized phase currents must be zero (See Figure 4 and Figure 5).

Figure 4. **Current flows through the shunt when the corresponding low-side switch is closed**

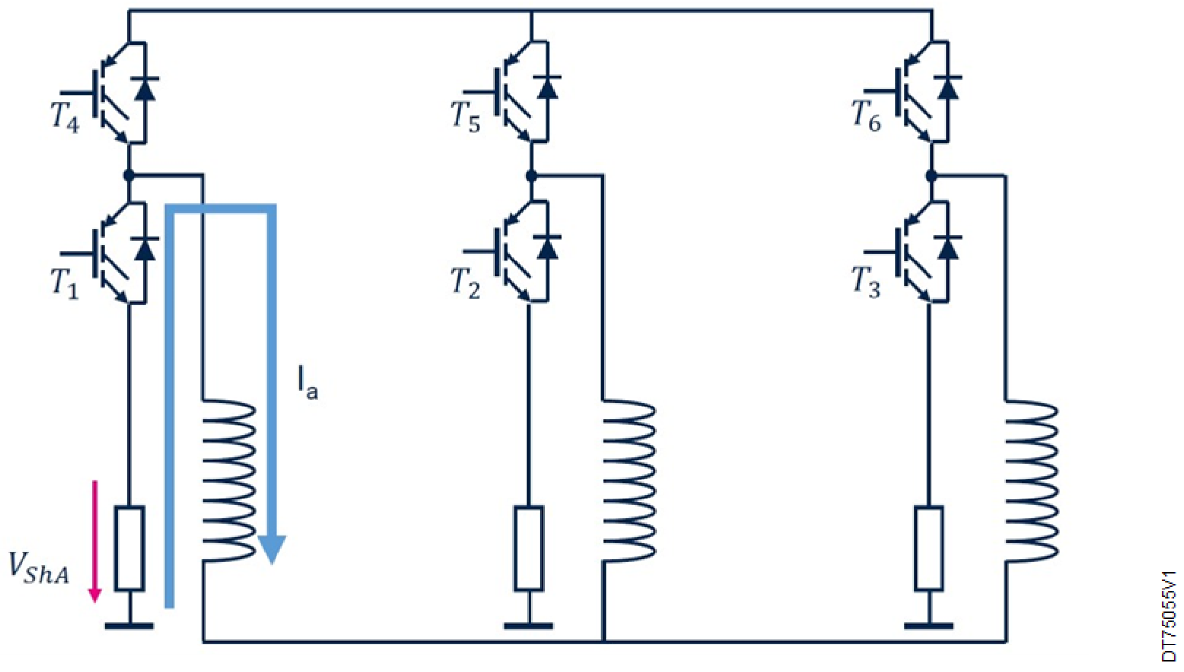
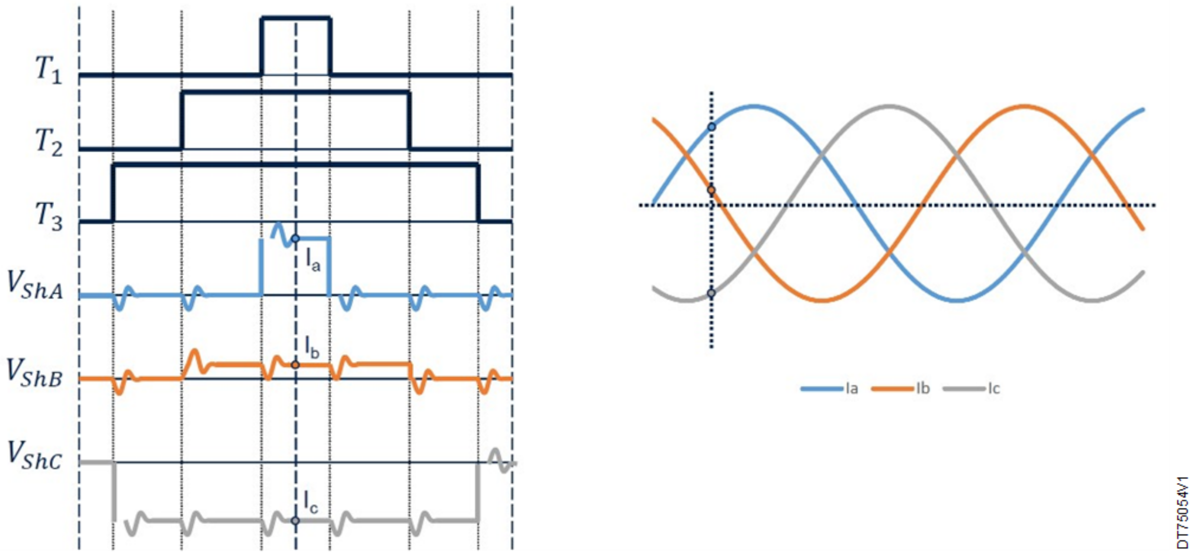


Figure 5. **Phase current measurement using shunt resistors and PWM duty cycles**



Note: *Depending on the PWM duty cycles, the voltages on the shunt resistors are proportional to the phase currents after sure certain stabilization time (to avoid inverter switching that introduces a noise on shunts). Sum of all the currents then should be zero (Ia + Ib + Ic = 0), therefore two phase currents are measured at a time and remaining one is calculated.*

The most critical moment of the PWM cycle is defined by the end of the ADC conversion (5 in Figure 6), which raises a high-priority interrupt to execute the PWM cycle computation process (6 in Figure 6) for the next period (FOC_HighFrequencyTask). Figure 7 extends some details of the Figure 6 with focus on second half of the PWM cycle. If the ADC conversion starts at the middle of the PWM cycle, the critical frequency for managing the next PWM cycle is defined by the sum of the ADC conversion and its interrupt service durations, which must be managed within the second half of the PWM cycle. If the PWM cycle is relatively slow and the STL maximum interrupt masking time can furthermore fully fit within the PWM half-cycle, no guarding loop is necessary (case II in Figure 7). However, if the STL maximum masking time could delay the end of the PWM computation service beyond the PWM cycle like case I in Figure 7, a guarding loop must be implemented as shown in Figure 6 (see item 11). This loop ensures that the STL execution is temporarily suspended to allow sufficient time for the next PWM computation procedure (6 in Figure 6) within the ongoing cycle. The STL can resume once the HighFrequencyTask is completed and so computation procedure of the next PWM cycle is done. Physically the guarding loop ends when interrupt service handling the HighFrequencyTask terminates (6 in Figure 6).

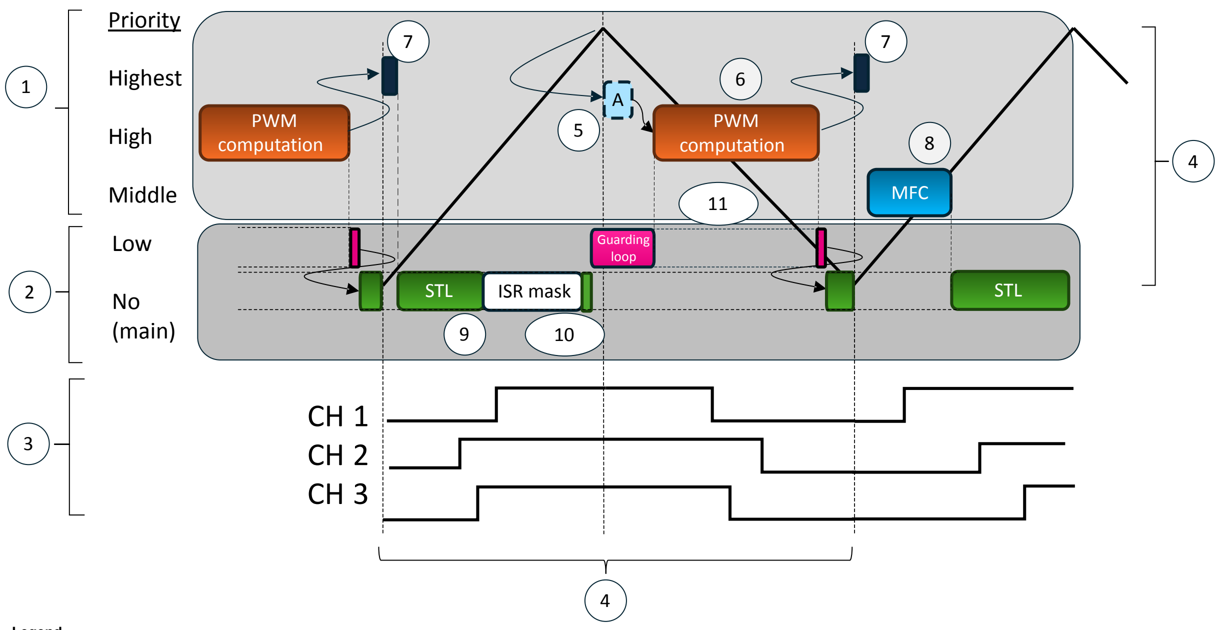
The guarding loop procedure prevents any occasional disruption of the new PWM cycle planning by an interrupt disabled during STL execution (9 in Figure 6). To be effective, the guarding loop interrupt service entry must be tuned to the end of the PWM period, accounting for the maximum STL interrupt disable interval and the time needed to complete the PWM computation service (6 in Figure 6) before the end of the ongoing cycle. This ensures that the STL applies the longest interrupt disable interval within its allowance window. The guarding loop service polls a specific release flag, inserted by the user at the end of the interrupt service handling the HighFrequency task. Once detected, the guarding loop terminates, unblocking the STL execution process, which can then resume at the main level.

There is no need to guard the execution of the highest priority TIM1 interrupt service, as the service is very short and always propagates itself within the first half of the PWM cycle. Suppose any interrupt disable interval can never exceed half of the cycle duration.

The medium frequency motor control task (8 in Figure 6) handles less critical timing processes (speed regulator, state machine, processing system faults). It occurs regularly but asynchronously to the PWM cycle. It is called from the SysTick interrupt service, which has a higher priority than the guarding service, therefore interrupting it. Its contribution must be included in the calculation of the system overall performance budget, ensuring sufficient time for STL execution at the lowest level, including the completion of the STL test cycle within the application process safety time.

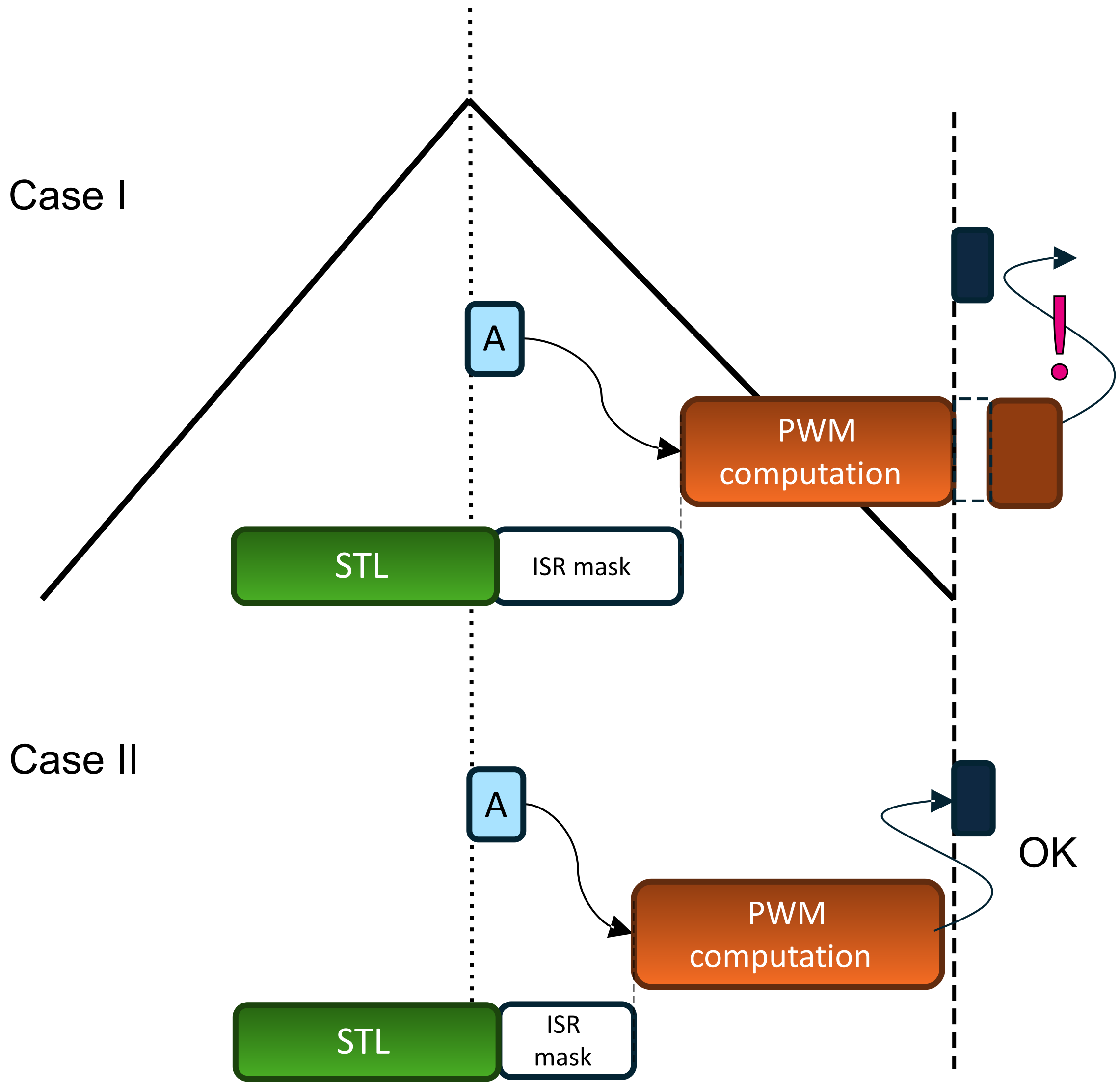
To relax main level performance, users can consider executing light application tasks, such as some fast STL tests, within the guarding loop time, provided they do not limit polling of the guarding terminating condition significantly. Users must avoid any interrupt masking at this level. All procedures and STL modules requiring masking must be executed exclusively at the lowest priority (main) level, guarded by the guarding loop implementation.

Figure 6. Guarding loop for critical interrupt latency in motor control



Legend
1. Motor control background process
2. STL partial execution at main level protected by guarding loop interrupt service routine
3. Timer 1 channel 1-3 outputs (control of motor driver low-side switches)
4. Timer 1 counter value (center-aligned PWM mode)
5. ADC conversion of the selected shunt's voltage measurement (triggered by hardware)
6. PWM planning, computation, and preprogramming procedure
7. Timer counter update physically propagating the precomputed PWMs and ADC measurement strategy for the newly starting cycle
8. Medium-frequency motor control task (not synchronized with PWM cycle)
9. STL partial execution cycle
10. Maximum interrupt masking interval caused by STL
11. Guarding loop interrupt service routine

Figure 7. **Time analysis of the single PWM cycle**



Case I: Guarding loop must be applied as there is a potential danger that the PWM computation procedure should not be completed before the next PWM period starts at the most critical case

Case II: Guarding loop implementation is not necessary as the PWM computation procedure is always manageable still within the ongoing PWM period even if the process faces an STL maximum interrupt masking interval

6 Timing analysis of motor control vs. available safety task time

An initial timing analysis must be done to consider the following criteria:

- The necessity of any guarding loop implementation to prevent any timing conflicts as visible in Figure 7. The goal is to verify how the critical motor control processes fit into a single PWM period summarized at **table 1** in Figure 8.
- The remaining performance budget available for STL overall execution management from a safety point of view. The estimation of time necessary to execute all the remaining motor control and application processes gives some picture about the time remaining to handle the STL overall flow cycle. See **table 2** in Figure 8.

The example is built on the STM32G4 MCU running at a frequency of 170 MHz while executing the motor control critical code from flash memory. Optionally, this can be forced into CCM-SRAM by the MC library compilation flag. As detailed in the next chapter, users can measure and calculate that the three-shunt FOC motor can be executed reliably up to approximately 38 kHz for such a configuration. The limiting factor is the sum of ADC conversion time (~0.5 µs) and the execution of the ADC1_2_IRQHandler (~12.7 µs), as both must be executed within the second half of the PWM cycle.

If we consider the maximum interrupt masking time applied by the STL library (see [1]), the achievable PWM frequency would be reduced to approximately 29 kHz to accommodate the possible increased interrupt service latency if no guarding loop is applied. By adding a guarding loop that implements STL suspension 4 µs before the latest possible entry to the ADC1_2_IRQHandler procedure, the safely achievable PWM frequency can return to the 38 kHz limit.

The trade-off for this adjustment is a reduction in the time available for all remaining application processes running at middle and lower levels (including the STL running at the lowest main level) to 34.5% of the overall MCU performance budget, which is mostly available during the first half of the PWM period. See **table 1** in Figure 8.

These values and frequencies are precalculated with a conservative approach since the duration of the PWM computation procedure (FOC_HighFrequencyTask called from the ADC1_2_IRQHandler) can slightly vary depending on the motor status. The worst-case scenario, measured at motor startup, was considered. Moreover, the critical part of the PWM computation procedure ends slightly before the ADC1_2_IRQHandler terminates. Therefore, even better results (PWM frequency up to 50 kHz) can be achieved in motor control, while a small overlap of this handler into the next PWM cycle could still be acceptable without corrupting the control process. However, the rest of the available MCU performance is significantly reduced (~16%) if the PWM cycle is set to 20 µs in this case.

Based on these numbers, users can estimate the time required to complete a single run of the test cycle when all the modules are executed once over the tested areas. For example, if a 170-KB code area should be tested in flash via the hardware CRC unit and a 160 KB code area in RAM, all the tests should be executed within an interval of approximately 46 ms, based on data adapted from table 34 in the user manual [1]. Users can then easily calculate that such a cycle, collecting only the MCU generic test APIs (supported by STL), can hardly be finished below ~140 ms (38 kHz) or ~320 ms (50 kHz) if considering the performance budget available for STL, assuming it is executed exclusively at the main level. It is evident that this is a purely theoretical case, not achievable in any real application. The contribution of all application processes, including application-specific tests, must be considered in a real application, further reducing the available budget for the STL. The MCU bus matrix load can also cut into the available space if the CPU competes with other controllers of the matrix, such as DMA transfers, which is the case with the STM32G4.

Moreover, in real application cases, memories are not tested over their entire ranges within a single run but in parts, in atomic steps during runtime, to split the MCU performance equally among other low-level tasks. Repetition of these atomic steps consumes a bit more MCU performance. Usually, these partial STL memory test modules executed in steps are completed by repeating all the other MCU testing modules. All these factors result in a longer interval to complete the STL test overall, depending on the scheduled flow, size of the tested memory areas, atomic steps applied, and the periodicity of their execution. Nevertheless, the calculated or measured verified time of the overall STL cycle execution should not exceed the process safety time defined by the application safety task.

Figure 8. Time analysis comparison

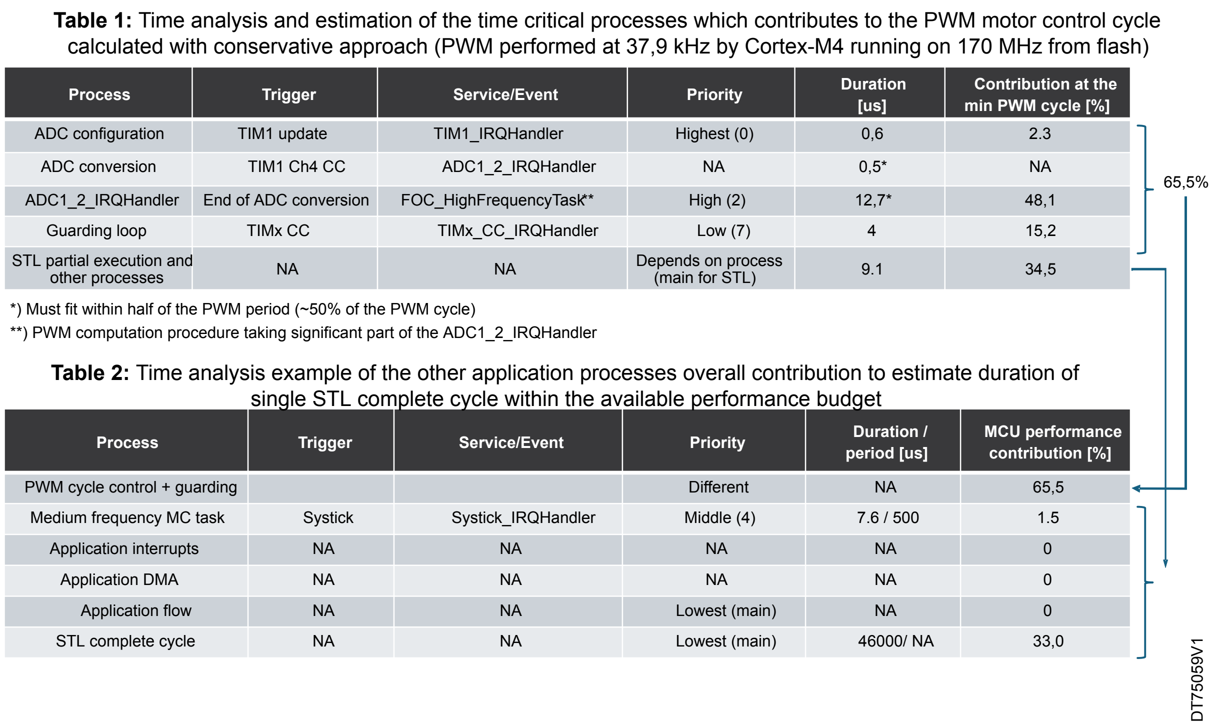

65,5%
DT75059V1

Table 1: Time analysis and estimation of the time critical processes which contributes to the PWM motor control cycle calculated with conservative approach (PWM performed at 37,9 kHz by Cortex-M4 running on 170 MHz from flash)

Process	Trigger	Service/Event	Priority	Duration [us]	Contribution at the min PWM cycle [%]
ADC configuration	TIM1 update	TIM1_IRQHandler	Highest (0)	0,6	2.3
ADC conversion	TIM1 Ch4 CC	ADC1_2_IRQHandler	NA	0,5*	NA
ADC1_2_IRQHandler	End of ADC conversion	FOC_HighFrequencyTask**	High (2)	12,7*	48,1
Guarding loop	TIMx CC	TIMx_CC_IRQHandler	Low (7)	4	15,2
STL partial execution and other processes	NA	NA	Depends on process (main for STL)	9.1	34,5

*) Must fit within half of the PWM period (~50% of the PWM cycle)
**) PWM computation procedure taking significant part of the ADC1_2_IRQHandler

Table 2: Time analysis example of the other application processes overall contribution to estimate duration of single STL complete cycle within the available performance budget

Process	Trigger	Service/Event	Priority	Duration / period [us]	MCU performance contribution [%]
PWM cycle control + guarding			Different	NA	65,5
Medium frequency MC task	Systick	Systick_IRQHandler	Middle (4)	7.6 / 500	1.5
Application interrupts	NA	NA	NA	NA	0
Application DMA	NA	NA	NA	NA	0
Application flow	NA	NA	Lowest (main)	NA	0
STL complete cycle	NA	NA	Lowest (main)	46000/ NA	33,0

7 Steps to integrate STL into a simple MCSDK project

The MCSDK project used in the STL integration example associated with this document was made by the following steps:

The instructions in this section describe how to integrate the STL library into any MCSDK project using the same motor driver topology. Instructions to create a simple minimal example project in MC workbench are provided as a reference and describe the steps used to create the included example project. For usage instructions of the MCSDK, refer to the MCSDK documentation.

Create a simple MCSDK project

Motor control workbench (version 6.3.0 used here)

1. Create a new project:
   - **Number of motors:** 1 motor
   - **Driving algorithm:** FOC (Field-Oriented control)
   - **Hardware mode:** Pack
   - **Select hardware:** P-NUCLEO-IHM03
   - **PWM frequency:** Set to a conservative or higher value (for example, 50 kHz to test the system behavior near its limits)
     - Navigate to "PWM generation" block -> Config -> PWM frequency (in Hz) -> OK
2. Save the project: Save the project to a suitable location.
3. Generate the project:
   - Project generation settings:
     - **STM32CubeMX version:** 6.12.0 used
     - **Preferred target toolchain:** Select your preferred toolchain
     - **Firmware package version:** FW V1.6.0 (as recommended)
     - **Drive type:** HAL (default)
   - Click on **Generate**.
   - Note: Generating the MC workbench project also runs CubeMX project generation. There is no need to open or change the CubeMX project at this point.

IDE project

1. Open the generated project: Open the generated project in your IDE of choice.
2. Modify the `main.c` file:
   - **Define the variable:** Define the following variable (make it global and volatile so that it can be modified by the debugger):

     ```
     volatile float programmed_speed_f = 150.0;
     ```

   - **Add code in `while(1)` Loop:** Add the following code into the main application `while(1)` loop in `main()` (within the `/* USER CODE BEGIN WHILE */` block):

     ```
     MCI_State_t MciState = MC_GetSTMStateMotor1();
     if (MciState == FAULT_OVER) {
         MC_AcknowledgeFaultMotor1();
     }
     if (MciState == IDLE) {
         MC_ProgramSpeedRampMotor1_F(programmed_speed_f, 5000);
         MC_StartMotor1();
     }
     ```

3. Build and run the project: Now, build and run the project. The motor should start running at 150 RPM or any speed set by the debugger before starting or restarting the motor.

Note: *When changing parameters in MCSDK up to version 6.3.0, the PWM frequency and regenerating the project, as this causes the `drive_parameters.h` file to be generated with incorrect parameters. This issue has been fixed in the version 6.3.1. A workaround for earlier versions of MCSDK is to generate the project from the `.stwb6` file in a new directory and replace the incorrectly overwritten `drive_parameters.h` file with one from a freshly generated separate project.*

In principle, there is no need to regenerate the motor control workbench project. The user needs to apply the following changes to the already generated STM32CubeMX project and associated IDE configuration only.

Perform the following steps to add the STL library to the MCSDK project:

1. Copy the **Middlewares** directory from the STL into the project directory structure on the file system
2. Perform the following modifications at IDE configuration:
   STM32CubeIDE
   - Create a folder `Middlewares/ST/STM32_Safety_STL` in the project structure and link the `stl_user_param_template.c` and `stl_util.c` files into this folder.
   - In the **Project** → **Properties** → **C/C++ General** → **Paths and Symbols** window of the project:
     - In the **Library Paths** tab, add `../Middlewares/ST/STM32_Safety_STL/Lib`.
     - In the **Libraries** tab, add `:STL_Lib.a`.
     - In the **Includes** tab, add `../Middlewares/ST/STM32_Safety_STL/Inc`.
   - In the linker file (`STM32G431RBTX_FLASH.ld`):
     - Add the following section between `.fini_array` and `.data` (necessary for RAM STL tests):
       `backup_buffer_section (NOLOAD): { *(backup_buffer_section) } >RAM`

   EWARM
   - Create a folder `Middlewares/ST/STM32_Safety_STL` in the project structure and link the `stl_user_param_template.c` and `stl_util.c` files into this folder.
   - In **Project** → **Options**:
     - In **C/C++ Compiler** → **Preprocessor** → **Additional include directories**, add the new directory:
       `$PROJ_DIR$/../Middlewares/ST/STM32_Safety_STL/Inc`
     - In **Linker** → **Library** → **Additional libraries**, add the new library:
       `$PROJ_DIR$/../Middlewares/ST/STM32_Safety_STL/Lib/STL_Lib.a`
   - In the linker file (`stm32g431xx_flash.icf`), replace the following line:
     ```
     place in RAM_region { readwrite, block CSTACK, block HEAP };
     ```
     with the following two lines:
     ```
     define block FIXED_ORDER_RAM with fixed order
               { section backup_buffer_section, readwrite, block CSTACK, block
     HEAP };
     place in RAM_region { block FIXED_ORDER_RAM };
     ```

   Keil® MDK
   - Create a folder `Middlewares/ST/STM32_Safety_STL` in the project structure and link the `stl_user_param_template.c`, `stl_util.c`, and `STL_Lib.a` files into this folder.
     - Right-click on `STL_Lib.a` file, choose **Options for File**, and change **File Type** to **Object file**.
   - Change the following project options:
     - In the **Target** tab, select **Use default compiler version 6** in the **Code Generation -> ARM Compiler** subwindow.
     - In the **C/C++ (AC6)** tab, uncheck the **Short enums/wchar** tick box and add `../Middlewares/ST/STM32_Safety_STL/Inc` to the **Include Path**.
     - In the **Linker** tab, uncheck the **Use Memory Layout from Target Dialog** tick box.
     - In the **Linker** tab, open the **Scatter File** via the **Edit** button and insert the line `*(backup_buffer_section)` at the start of the RAM section.
     - Change the **.sct** file location to the root of the Keil® project (recommended).

3. Perform the following modifications in STM32CubeMX:
   - In **Project Manager** → **Code Generator** → **Generated files** tab:
     - Uncheck **Delete previously generated files when not regenerated**.
     - Check **Backup previously generated files when regenerating** (recommended).
   - In the **Pinout & Configuration** tab:
     - In the **Timers** tab, set up the following parameters for **TIM4**:
       - **Clock Source**: internal clock
       - **Prescaler**: `((TIM_CLOCK_DIVIDER) - 1)` (change "Decimal" to "No Check")
       - **Counter Mode**: Up
       - **Counter Period**: `(guarding_delay)` (change "Decimal" to "No Check")
     - In **System Core** → **NVIC** tab:
       - In the **NVIC** subtab, enable **TIM4 global interrupt** and set **Preemption Priority** to 7.
       - In the **Code generation** subtab, uncheck the **Call HAL handler** column for **TIM4 global interrupt**.
   - Generate the code
4. Perform the next modifications at application **main.c** file here.
   - At the beginning of the file

```
#include "stl_user_api.h" // include header file with STL API definitions
volatile int guarding_delay = 1000; // define debugger tunable variables, values to
be tweaked later
volatile int STL_activate = 0;
```

   - Before the main loop body entry

```
LL_TIM_SetOnePulseMode(TIM4, LL_TIM_ONEPULSEMODE_SINGLE);// enable TIM4 one-pulse
mode
__HAL_TIM_CLEAR_IT(&htim4, TIM_IT_UPDATE);
__HAL_TIM_ENABLE_IT(&htim4, TIM_IT_UPDATE);

STL_SCH_Init(); // call STL scheduler initialization
```

   - Within the main loop body

```
if (STL_activate != 0) { // conditional insertion of the STL flow (here TM7 is
called exclusively)
    STL_TmStatus_t StlTmStatus = STL_NOT_TESTED;
    STL_Status_t StlStatus = STL_SCH_RunCpuTM7(&StlTmStatus);
    if (StlTmStatus != STL_PASSED || StlStatus != STL_OK) {
    // fail, add error handler entry here
  }
}
```

5. Perform next modifications at application **stm32g4xx_mc_it.c** here:
   – At the beginning of the file

```
extern volatile int MC_hi_f_task_done;  // global variable declaration - to
signalize guarding loop end, variable is defined in stm32g4xx_it.c
volatile int wait_loop_hi_f_enabled = 1;  // to control enable and disable the
guarding loop while debugging
extern volatile int guarding_delay;  // to control the guarding loop timing while
debugging
extern TIM_HandleTypeDef htim4;  // to control TIM4
```

   – At the end of ADC1_2_IRQHandler ("USER CODE BEGIN HighFreq" block)

```

MC_hi_f_task_done = 1;  // guarding loop can terminate (placed at the end of
ADC1_2_IRQHandler "USER CODE BEGIN HighFreq" block)
```

   – At the start of TIMx_UP_M1_IRQHandler ("USER CODE BEGIN TIMx_UP_M1_IRQn 0" block)

```

if (wait_loop_hi_f_enabled) {  // guarding loop control at the start of
TIMx_UP_M1_IRQHandler ("USER CODE BEGIN TIMx_UP_M1_IRQn 0" block)
    int x = guarding_delay - TIM1->CNT;  // subtract TIM1 value from guarding delay
to compensate ISR entry latency
    if (x > 0) {
        TIM4->ARR = x;
    } else {
        TIM4->ARR = 0;
    }
    TIM4->CNT = 0;
    NVIC_ClearPendingIRQ(TIM4_IRQn);
    __HAL_TIM_CLEAR_IT(&htim4, TIM_IT_UPDATE);
    TIM4->CR1  |= TIM_CR1_CEN;
}
```

6. Perform next modifications at application **stm32g4xx_it.c** file here:
   – At the beginning of the file

```
volatile int MC_hi_f_task_done = 0;  // global variable declaration
extern volatile int wait_loop_hi_f_enabled;
```

   – At the beginning of TIM4_IRQHandler()

```

    if (TIM4->SR & TIM_SR_UIF) {
        __HAL_TIM_CLEAR_IT(&htim4, TIM_IT_UPDATE);
    }
    MC_hi_f_task_done = 0;
    while (wait_loop_hi_f_enabled && !MC_hi_f_task_done) {
        if (!(TIM1->DIER & TIM_DIER_UIE)) {
            // Motor control timer IRQ cycle stopped, break out:
            break;
        }
    }
```

7. Build and debug the project

As implied by the code adapted above, the guarding loop is inserted by default. STL can be started by changing the `STL_activate` variable using the debugger, as well as tweaking the guarding loop timing by changing the `guarding_delay` variable. Note that the STL flow is reduced in the shown code snippets. The TM7 module is called only to perform significant stress when measuring the guarding loop efficiency due to the frequent interrupt disable intervals applied by this module. The included example does not demonstrate the usage of flash, RAM, or the remaining CPU test modules, nor does it show the correct handling of error outputs from these modules. The flow must be adapted by the end user. Refer to the project examples delivered with the STL library.

8 Setup to observe and measure the PWM cycle parameters

Users can implement a specific setup to measure the critical processes within the PWM loop. This could be especially helpful if the application runs close to the system performance limit and fine-tuning all the parameters is a must to prevent any unexpected jitter or motor control cycle interference. The best method is observing some auxiliary GPIOs toggling within the intervals of executed critical routines. To do so, users can measure and set up the following GPIOs:

- **PC5**: TIM1 CH4N signal (used for ADC measurement) to trigger a stable signal for the oscilloscope.
- **PC10**: Set and reset when the TIM1 update interrupt is entered and exited to measure the highest priority task duration (7 in Figure 6).
- **PC11**: Set and reset when the high-frequency task or its critical part is entered and exited (6 in Figure 6).
- **PC12**: Signals that the guarding loop is active or interrupted.

CubeMX setup:

- At **Pinout & Configuration** → **Timers** → **TIM1** → **Mode**, change **Channel 4 Mode** from **PWM Generation no output** to **PWM Generation CH4N**.
- At **Pinout & Configuration** → **Timers** → **TIM1** → **Parameter Settings**.
  - Change **Counter Period** from `((PWM_PERIOD_CYCLES) / 2)` to `((PWM_PERIOD_CYCLES) / 2 + 1)` to ensure a nonzero width pulse.
  - At **PWM Generation Channel 4** → **Mode**, change the PWM back to **mode 2** (note that reconfiguration of CH4 reverts it to mode 1).
- At **Pinout & Configuration** → **Timers** → **TIM1** → **GPIO Settings**, change **PC5 GPIO speed** to **Medium**.
- At **Pinout view**, change **PC12**, **PC11**, and **PC10** modes to **GPIO Output**.
- Regenerate the CubeMX project.

Application code modification:

1. Insert the following line in the `main.c` file before the main body loop entry (block **USER CODE BEGIN 2**):
```
TIM1->CCER |= TIM_CCER_CC4NE;  // enable TIM1 CH4N output signal
```
2. Place the following lines in `stm32g4xx_it.c` at the beginning and end of `TIM4_IRQHandler`:
```
GPIOC->BSRR = 1<<12;  // at the beginning
GPIOC->BRR = 1<<12;   // at the end
```
3. Place the following lines in `stm32g4xx_mc_it.c`:
   - At the beginning and end of `ADC1_2_IRQHandler` (blocks **USER CODE BEGIN ADC1_2_IRQn 0** and **1**):
```
GPIOC->BSRR = 1<<11;  // at the beginning
GPIOC->BRR = 1<<11;   // at the end
```
   - At the beginning and end of `TIMx_UP_M1_IRQHandler` (blocks **USER CODE BEGIN TIMx_UP_M1_IRQn 0** and **1**):
```
GPIOC->BSRR = 1<<10;  // at the beginning
GPIOC->BRR = 1<<10;   // at the end
```

Oscilloscope setup for timing requirement measurement:

- Use an oscilloscope with infinite persistence enabled.
- Trigger off **PC10** (TIM1 update interrupt).
- Place **cursor_1** exactly on the **PC5** (TIM1 CH4N) pulse (when stable, that is, no deviation is happening).
- Clear the persistence memory.
- Leave running for a long enough time.
- Place **cursor_2** on the latest recorded falling edge of **PC11** (HighFrequencyTask).

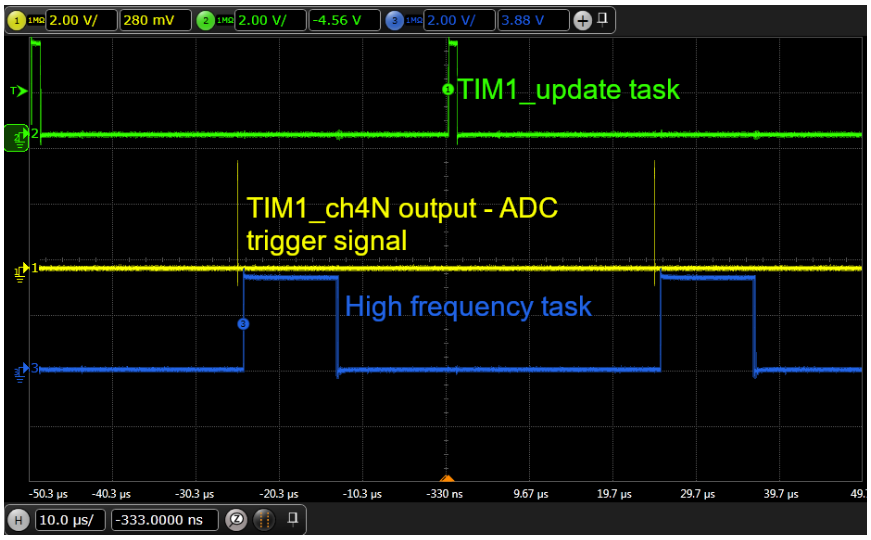
To measure the duration of the entire part of the high-frequency task accurately, it is necessary to perform the measurement on a project with a low enough PWM frequency so that there is no danger of ISRs interfering with the measurements (for example, 20 kHz) and with STL not running (clean project).

The high-frequency task takes the longest time interval when the motor is starting. Make sure that during the measurement, the motor is restarted regularly (for example, by setting its requested speed at a stable value below 1 RPM, which causes the motor to stall and restart repeatedly). Alternatively, users can measure the critical part of the high-frequency task only. In this case, the **PC11** pin toggle must be placed in the `mc_tasks_foc.c` file into the block **USER CODE BEGIN HighFrequencyTask SINGLEDRIVE_2**.

At the rest of this chapter user can find a few snapshots taken during measurement of the P-NUCLEO-IHM03 pack applied for control of the GBM2804H-100T motor.

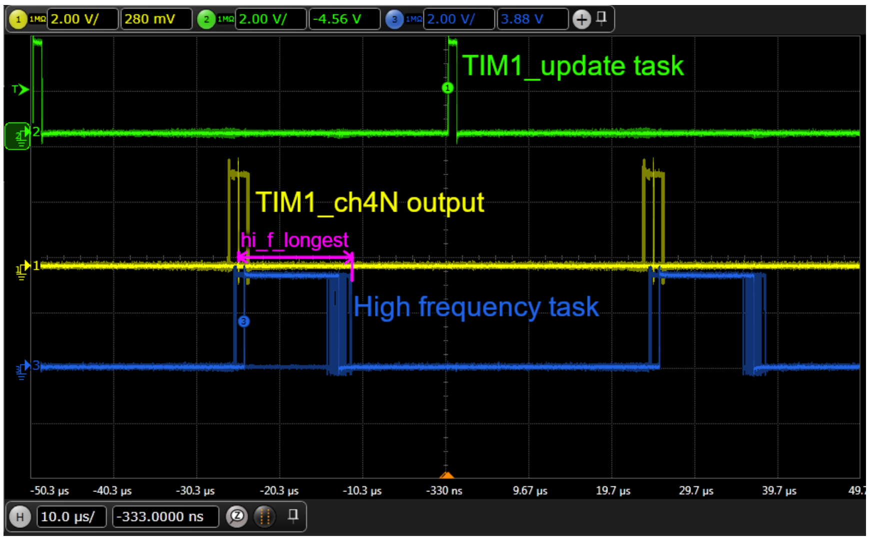
1. Motor stable state at PWM frequency 20 kHz [1]
   – The motor already started up.
   – STL activity is disabled. No guarding loop is applied.

Figure 9. Timing of motor control tasks during normal operation



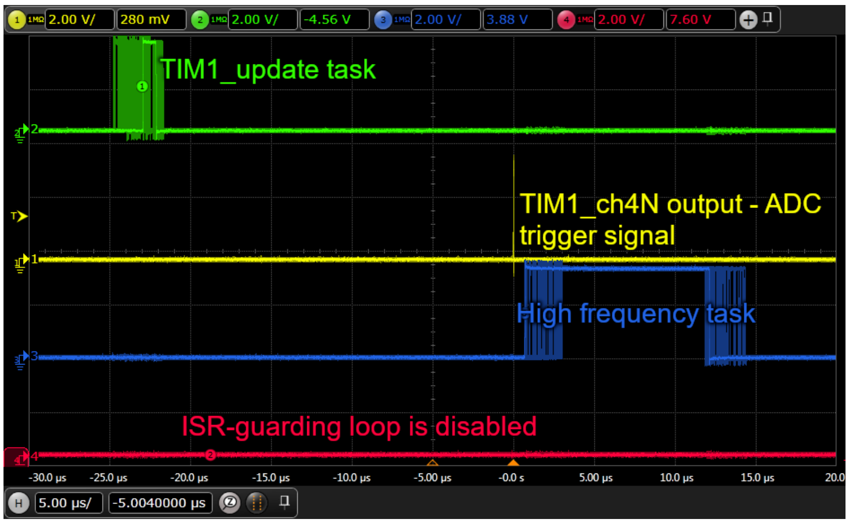
2. Motor startup with PWM frequency of 20 kHz [1]
   – High-frequency task takes the longest. Its start can be deviated in time too.
   – The moment of its worst-case can be measured from the center of the PWM pulse.
   – STL activity is disabled. No guarding loop is applied.

Figure 10. Timing of motor control tasks during motor startup



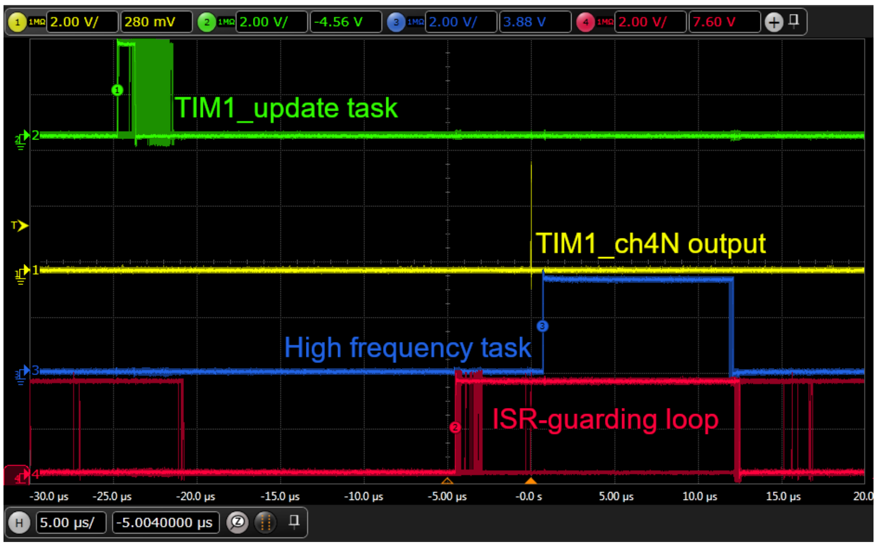
3. Motor stable state with PWM frequency of 20 kHz [2]
   - STL is running, with interrupt masking enabled, and the guarding loop is disabled.
   - Interrupt masking causes jitter on the high-frequency task.

Figure 11. **Timing of motor control tasks during normal operation with STL enabled**



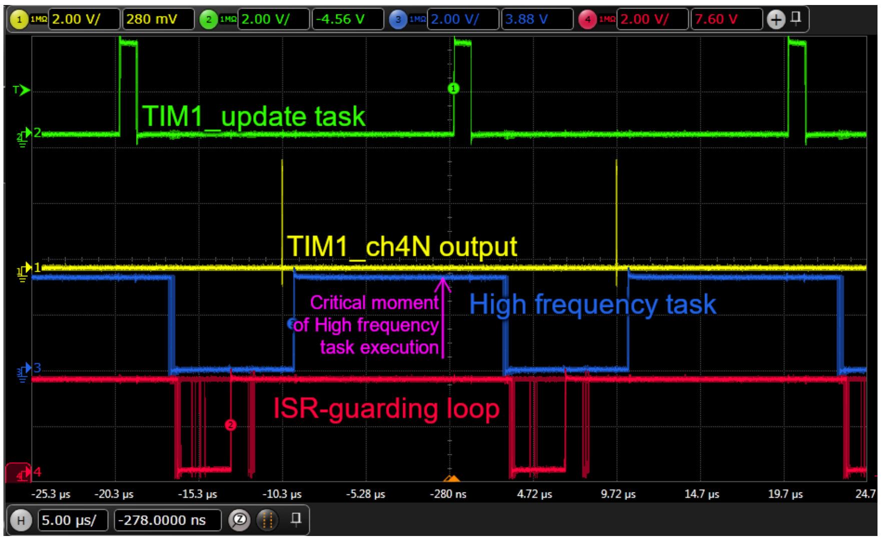
4. Motor stable state at PWM frequency 20 kHz [2]
   - STL is running, guarding loop is enabled.
   - Jitter on the high-frequency task disappears. Its execution slot is protected by a low-level guarding interrupt service preventing any interrupt masking there.
   - Jitter observed on the TIM1_update task execution signals that the task is not protected by the guarding loop. The loop is released by the high-frequency task. In this case, its protection is not necessary as it does not take up a significant portion of the half-cycle.
   - Jitter can be observed on guarding loop entry, as well as occasional longer pulses when the guarding loop gets interrupted by the SysTick handler.

Figure 12. **Timing of motor control tasks during normal operation with STL enabled but guarded**



5. Motor stable state at critical PWM frequency 50 kHz [2]
   – This figure shows a case where the critical part of the high-frequency task still gets executed before the TIM1 update event. It still fits into the second half PWM cycle. Even though its entire execution time stretches into the next PWM cycle begin, the motor control still functions well if the guarding loop is used. As a result of this extreme case, little execution time is left for the main task. Signs of task starvation by the guarding loop or higher priority interrupts can be observed.

Figure 13. Timing of motor control tasks with STL active but guarded at nearly maximum possible PWM frequency



Note:

[1]	*Triggered off PC10 (TIM1_update task)*
[2]	*Triggered off PC5 (TIM1_ch4N)*

Revision history

Table 2. Document revision history

Date	Revision	Changes
27-Nov-2024	1	Initial release.

Contents

1 General information 2

2 Overview 3

3 Interference of STL running at application. 4

4 How to compensate effects of the STL interrupt masking. 5

4.1 Assuming responsibility for all possible interferences and conflicts 5

4.2 No concerns about interrupt latency 5

4.3 Adaptable dummy Loop at start of critical interrupt. 5

4.4 Guarding loop service protecting higher priority latency-critical interrupt entry 6

5 Implementing latency-compensating guarding loop in motor control applications . . 7

6 Timing analysis of motor control vs. available safety task time 11

7 Steps to integrate STL into a simple MCSDK project. 13

8 Setup to observe and measure the PWM cycle parameters 17

Revision history 21

List of tables 23

List of figures. 24

List of tables

Table 1. Referenced document 2
Table 2. Document revision history 21

List of figures

Figure 1. Effect of interrupt masking performed by the STL 4
Figure 2. Compensating interrupt latency with adaptable dummy Loop 5
Figure 3. Interrupt latency compensated by low-priority guarding loop 6
Figure 4. Current flows through the shunt when the corresponding low-side switch is closed 7
Figure 5. Phase current measurement using shunt resistors and PWM duty cycles 8
Figure 6. Guarding loop for critical interrupt latency in motor control 9
Figure 7. Time analysis of the single PWM cycle 10
Figure 8. Time analysis comparison 12
Figure 9. Timing of motor control tasks during normal operation 18
Figure 10. Timing of motor control tasks during motor startup. 18
Figure 11. Timing of motor control tasks during normal operation with STL enabled 19
Figure 12. Timing of motor control tasks during normal operation with STL enabled but guarded 19
Figure 13. Timing of motor control tasks with STL active but guarded at nearly maximum possible PWM frequency 20