# Introduction to I3C for STM32 MCUs

## Introduction

The number of sensors connected to a main device acting as a controller has steadily increased in the last couple of years. Managing multiple sensors is becoming more complex with heterogeneous serial interfaces like the still widespread, low-cost $I^2C$, SPI, capable of handling more bandwidth, or the traditional UART, and extra signals (GPIOs) for interrupt and/or power modes.

A high level of hardware and software integration, power efficiency, high speed, cost effectiveness, and fast time-to-market are required. For this reason, in 2017, the MIPI Alliance introduced a new serial interface called improved inter-integrated circuit (I3C). The MIPI I3C standard is defined by the MIPI specification, version 1.1.1. I3C supports the new features and maintains some backward compatibility with $I^2C$.

The new series of STM32 MCUs integrate the I3C peripheral, supporting the set of required features in SDR mode, as defined by the MIPI specification v1.1.1.

The purpose of this application note is to provide some I3C examples based on STM32CubeMX, to cover most of the I3C communication modes available on STM32 microcontrollers and provide recommendations for the correct use of the I3C peripheral.

This application note starts with an I3C bus overview, followed by a description of the I3C features, and a use case based on STM32CubeMX for communicating only with I3C targets via an I3C mixed bus through dynamic address assignment, CCC commands, data exchange with sensors (LSM6DSO and LIS2DW12) in private or direct mode, and the management of in-band interrupts from targets.

**Table 1. Applicable products**

| Type | Products |
|---|---|
| Microcontroller | STM32H5 series, STM32H7R3/7S3 line, STM32H7R7/7S7 line, STM32U3 series, STM32N6 series |

# 1 General information

This document applies to the Arm® Cortex® core-based STM32 microcontrollers.

*Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

# 2 I3C bus overview

This section provides a general summary description of the new I3C protocol and gives an overview of typical operations and fundamental modes using I3C in SDR mode.

## 2.1 Operations

The I3C bus supports different modes and operations of various message types:

- Broadcast and direct common command code (CCC) messages to communicate with all targets or a specific one.
- Dynamic addressing: I3C assigns a dynamic address, unlike $I^2C$, which has a static address.
- Private read/write transfers.
- Legacy $I^2C$ messages: the I3C controller can communicate with $I^2C$ devices on the bus.
- In-band interrupt (IBI): the target device, which is connected to the bus, can send an interrupt to the controller over the two-wire (SCL/SDA).
- Hot-join request: the target can join the I3C bus after initialization.
- Controller role request.

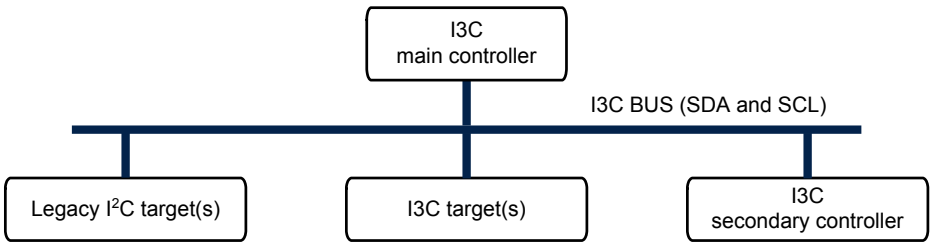
## 2.2 I3C bus

### Bus configuration

I3C is a two-wire interface, similar to the $I^2C$ bus. The I3C bus consists of a serial data line (SDA) and a serial clock line (SCL), for a pure I3C bus (or a mixed bus if an $I^2C$ target device is connected without clock stretching). There are four main types of devices on the bus:

- I3C main controller.
- I3C secondary controller.
- I3C targets.
- Legacy $I^2C$ targets.

**Figure 1. I3C communication interface**




### Bus communication

SCL can run at up to 12.5 MHz in push-pull mode and up to 4 MHz in open-drain mode.

- Pure bus: (only I3C devices on the bus).

SCL can be set to 12.5 MHz, and this is the ideal case in terms of performance.

- Mixed fast bus: ($I^2C$ and I3C devices on the bus).

### Case: $I^2C$ devices have a 50-ns spike filter

- SCL supports a limited range of speeds.
- Mixed slow bus.

**Case: I²C device does not support a 50-ns spike filter**

- SCL is limited to the slowest I²C device connected to the bus.

## 2.3 Bus format

All transfers on the I3C bus begin with a START (S), followed by a header byte (7h'7E), target address, broadcast command code, or more, and end with a STOP (P) from the controller.

The active controller on the bus is responsible for generating the I3C timing in SDR mode and to send I3C commands (CCC), addressing all targets or a specific target.

| | | | | | | |
|---|---|---|---|---|---|---|
| S/Sr | I3C reserved byte 7'h7E | Rn/W | ACK/NACK | DATA | T | P |
| | I3C target address | | | | | |
| | I2C target address | Rn/W | ACK/NACK | DATA | ACK/NACK | |

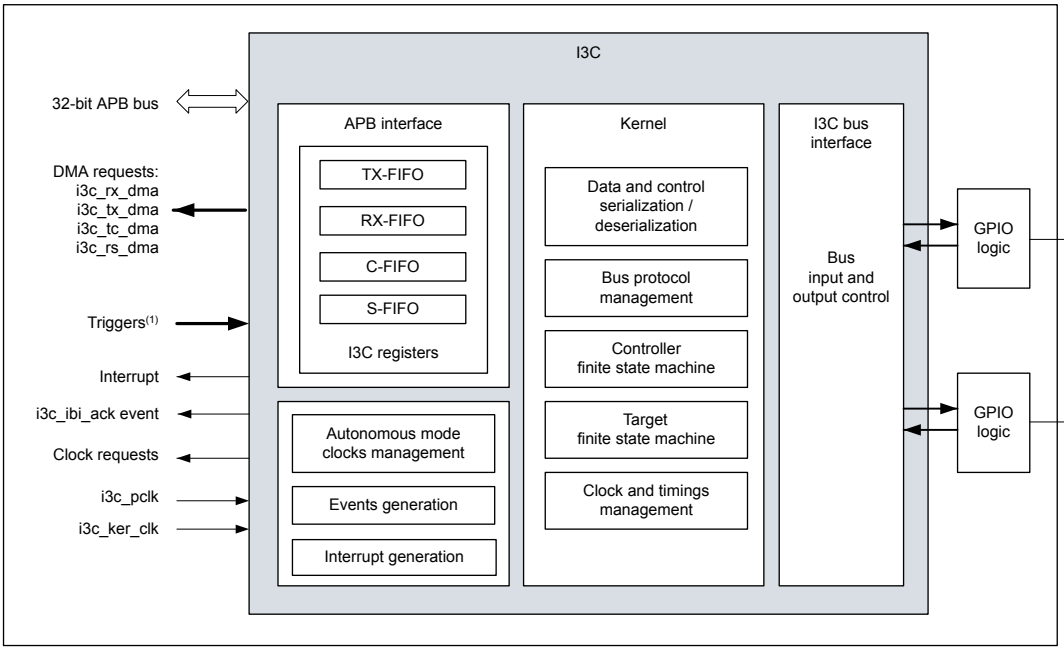
**T-bit:**

- SDR controller written data as parity: as an odd parity.
  The value of this parity bit shall be the XOR of the eight data bits with 1, that is: XOR (data [7:0], 1).
  Example:
- 0x0F: T-bit value is 1.
- 0xF2: T-bit value is 0.
- SDR (read) data: as end-of-data:
  - 1: Continue the message.
  - 0: End the message.

## 2.4 Block diagram

The following figure shows the I3C block diagram.

**Figure 2. I3C block diagram**




1. *This feature is implementation-dependent, and can be unavailable.*

The transmit/receive FIFO threshold can be configured as threshold 1 byte or 4 bytes and may be used during any transfer.

- C-FIFO/S-FIFO size is two words.
- TX-FIFO/RX-FIFO size is eight bytes.

# 3 I3C versus I²C

The I3C protocol is based on the same serial two-wire interface as I²C, allowing for communication with legacy I²C devices with some restrictions.

I3C has improved low-power performances over I²C, mainly because SCL and, most of the time, SDA are driven in push-pull mode, and because SDA can be in open-drain mode with an integrated controller pull-up.

I3C also reduces power consumption compared to I²C modes and provides a higher data rate at 12.5 MHz. All these improvements are achieved with I3C while maintaining some compatibility with legacy I²C devices.

**Table 2. I3C versus I²C capabilities**

| Parameters | I3C | I²C |
|---|---|---|
| Protocol | Serial, half-duplex | Serial, half-duplex |
| Frequency | Up to 12.5 MHz (SDR) | Up to 1 MHz |
| Signaling | Open-drain<br>Push-pull | Only open-drain |
| Pull-up resistors at bus level | Not required | Required |
| I3C reserved address 7'h7E | To begin I3C SDR transfer | Not allowed |
| Address | 7-bit (Static address for I2C devices and dynamic address for I3C devices) | 7-bit static address<br>10-bit static address |
| Dynamic addressing | Supported | Not supported |
| In-band interrupt | Supported | Not supported |
| Hot-join | Supported | Not supported |
| Multicontrollers | Supported | Supported |

# 4 STM32H5 implementation

The I3C SDR-only peripheral supports all features of the MIPI I3C specification v1.1 for the I3C primary controller, secondary controller, and target. It can control all I3C-bus-specific sequencings, protocols, arbitrations, and timings, and can act as controller or target.

## 4.1 I3C MIPI support

The I3C peripheral supports the MIPI specification v1.1, as shown in the table below.

**Table 3. I3C peripheral versus MIPI v1.1**

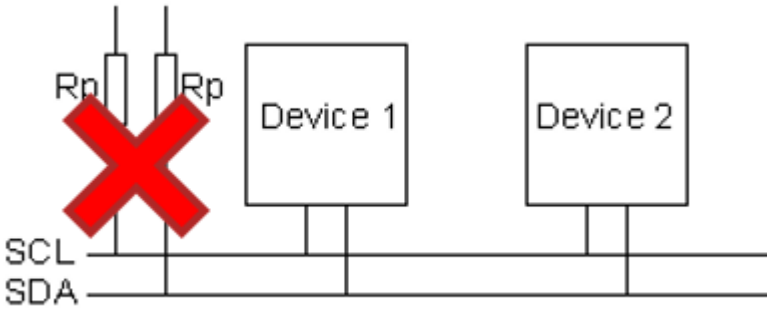
| Feature | MIPI I3C v1.1 | I3C peripheral | | Comments |
|---|---|---|---|---|
| | | Controller | Target | |
| I3C SDR message | X | X | X | - |
| Legacy $I^2C$ message | X | X | - | Mandatory as controller when there is an I2C target.<br>Optional in MIPI v1.1 when target. |
| HDR message | X | - | - | Optional in MIPI v1.1. |
| Dynamic address | X | X | X | - |
| Static address | X | X | - | I3C peripheral cannot be a target on an $I^2C$ bus. |
| Grouped addressing | X | X | - | Optional in MIPI v1.1. |
| CCCs | X | X | X | All mandatory CCCs + some optional CCCs are supported. |
| Error detection & recovery | X | X | X | - |
| IBI | X | X | X | - |
| Secondary controller | X | X | X | - |
| Hot join | X | X | X | - |
| Target reset | X | X | X | - |
| Asynchronous timing control 0 | X | X | - | Mandatory in MIPI v1.1 when controller.<br>Optional in MIPI v1.1 when target. |
| Synchronous & asynchronous timing control 1, 2, 3 | X | - | - | Optional in MIPI v1.1. |
| Device to device tunneling | X | X | - | Optional in MIPI v1.1. |
| Multi-lane data transfer | X | - | - | Optional in MIPI v1.1. |
| Monitoring device early termination | X | - | - | Optional in MIPI v1.1. |

## 4.2 I3C peripheral integration

### Integration with bus

Pull-up is on during open-drain phase, and is off during push-pull phase. It is handled automatically by the hardware.

**Figure 3. No pull-up needed in push-pull mode**

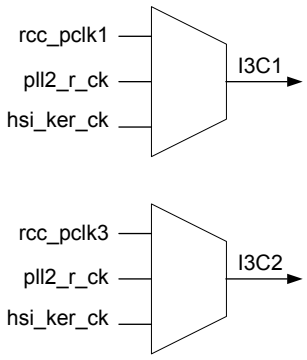



### Integration with RCC

From the I3Cx mux, we can select the clock source:

- rcc_pclk1/3
- pll2_r_ck
- hsi_ker_ck

**Figure 4. I3Cx multiplexer**

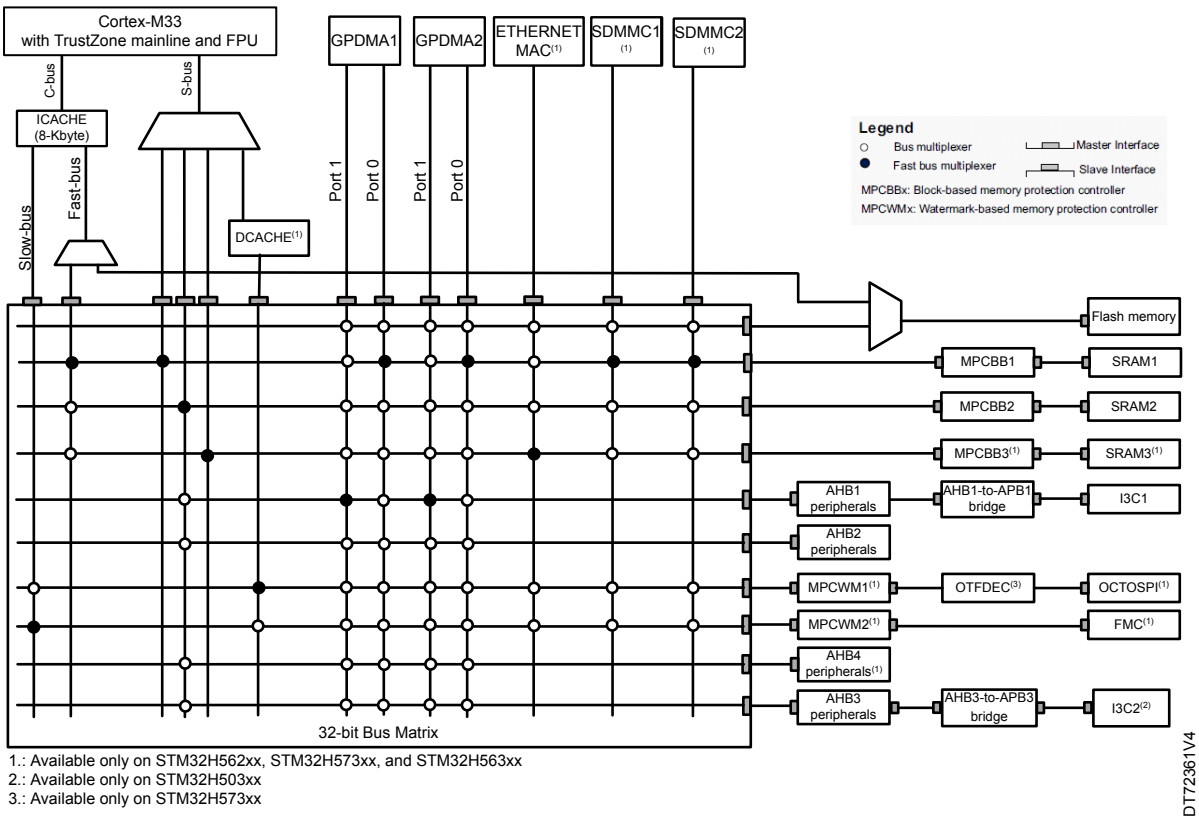



DT71380V1

### GPDMA

- The I3C peripheral can be used with GPDMA in order to off-load the CPU.

**Figure 5. Example of I3C mode GPDMA architecture**




1.: Available only on STM32H562xx, STM32H573xx, and STM32H563xx
2.: Available only on STM32H503xx
3.: Available only on STM32H573xx

# 5 I3C hardware requirements

The I3C interface provides major efficiencies in bus power management while providing significant speed improvements compared to $I^2C$.

## 5.1 Electrical characteristics

This section describes the electrical parameters of the I3C interface in two modes (for more details, refer to the MIPI specification):

- Push-pull mode.
- Open-drain mode.

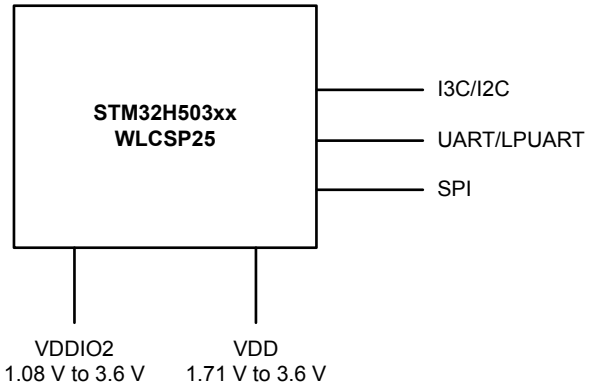
**Table 4. Electrical characteristics**

| Parameters | Min | Typical | Max | Unit |
|---|---|---|---|---|
| Operating voltage | 1.10 | 1.20 | 1.30 | V |
| | 1.65 | 1.80 | 1.95 | V |
| | 2.97 | 3.30 | 3.63 | V |
| SCL clock frequency | 0.01 | 12.5 | 12.9 | MHz |
| Total bus capacitance | - | 50 | - | pF |

## 5.2 I3C-dedicated VDDIO2 supply pin

I3C and other communication interfaces such as $I^2C$, USART/LPUART, and SPI are mapped on GPIOs that have a dedicated VDDIO2 supply pin down to 1.08 V. Therefore, all these communication interfaces, including I3C, can operate at 1.2 V.

**Figure 6. STM32H503xx communication interfaces operating with VDDIO2 at 1.2 V (USART/LPUART, SPI, and $I^2C$/I3C)**



# 6 Bus timing

The active controller provides an open-drain and a push-pull mode on the bus. It can also issue a START or STOP by driving the SDA line low while keeping the SCL line high.

- The device acts as a controller: the user can adjust timings by using I3C timing register 0, 1, and 2.
- The device acts as a target: the bus available condition and the bus idle condition can be modified.

SCL can run at up to 12.5 MHz in push-pull mode → 1.4 Mbyte/s.

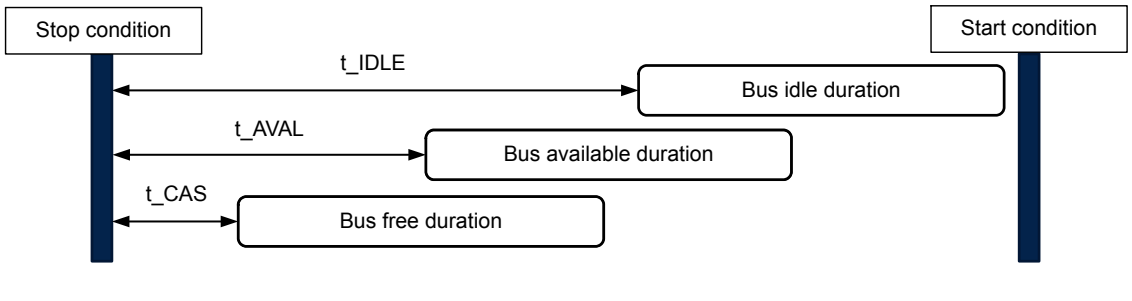
Frequency of the I3CCLK kernel clock > 2x frequency of the SCL clock

*Note: Sustaining FSCL max = 12.5 MHz means a frequency of the I3CCLK kernel clock > 25 MHz*

Before a START, there are three conditions to be fixed for a controller/target, as outlined below:

1. Bus free duration:
   - For pure bus: only I3C devices are connected on the bus.
   - For mixed bus: at least one legacy $I^2C$ device is present on the I3C bus.
2. Bus available duration: a target may only issue a START request after a t_AVAL.
3. Bus idle duration: the SDA and SCL lines sustain a high level for a duration of at least t_IDLE.

**Figure 7. Bus timing**




### Maximum achievable I3C frequencies with STM32 products

SCLH and SCLL are programmed numbers of the internal clock. Therefore, the period of SCL is a multiple of internal clock periods (see I3C timing register 0 (I3C_TIMINGR0) section in the corresponding reference manual).

STM32U3 products example:

For a 96 MHz frequency:

- 7 periods of the internal clock give an SCL period of 72.91 ns → Frequency = 13.7 MHz.
- 8 periods of the internal clock give an SCL period of 83.3333 ns → Frequency = 12 MHz.
- 9 periods of the internal clock give an SCL period of 93.744 ns → Frequency = 10.66 MHz.

The maximum frequency that can be reached with a 96 MHz internal clock is 12 MHz because 13.7 MHz is above the maximum frequency allowed by the standard.

For a 50 MHz frequency (using HSE):

- 3 periods of the internal clock give an SCL period of 60 ns → Frequency = 16.6 MHz.
- 4 periods of the internal clock give an SCL period of 80 ns → Frequency = 12.5 MHz.
- 5 periods of the internal clock give an SCL period of 100 ns → Frequency = 10 MHz.

The maximum frequency that can be reached with a 50 MHz internal clock is 12.5 MHz because 16.6 MHz is above the maximum frequency allowed by the standard.

# 7 $t_{SCO}$ timing

$t_{SCO}$ (clock-to-data turnaround delay) is the internal delay in the I3C target device from the SCL input to the start of the SDA output (for more details, refer to the MIPI specification). This parameter drives the maximum effective frequency of the reads that a controller should issue to accommodate for a given target capability.

The default maximum clock-to-data turnaround delay is 12 ns accordoing to the MIPI I3C timing specification.

An I3C target may have higher $t_{SCO}$ than 12 ns, in this case it must inform the controller that it has a maximum data speed limitation:

- During the address assignment procedure, when it acknowledges the ENTDAA CCC and provides its capability register; more specifically the target set to 1 the BCR[0] to inform the controller.
- On a direct GETBCR CCC from the controller, when it returns the same bit BCR[0].

The STM32 I3C peripheral is compliant with the MIPI specification when acting as target and is able to set BCR[0] = 1 when $t_{SCO}$ > 12 ns (refer to I3C_BCR register of the reference manual (RM0492)).

If BCR[0] = 1, the controller should then query more specifically about the nature of the maximum data speed limitation of the target by issuing a direct GETMXDS CCC.

The STM32 I3C peripheral is also compliant with the MIPI specification on a direct GETMXDS CCC: if $t_{SCO}$ > 12 ns, it then returns a maxRd byte with maxRd[5:3] = 111. Then the user should read the exact value reported in the datasheet and adjust consequently the maximum speed on the I3C bus for read transfers.

# 8 I3C features description

## 8.1 Dynamic address assignment mode

This section describes the I3C broadcast ENTDAA CCC, as it is communicated on the I3C bus and as it is programmed.

## 8.2 Bus initialization

In I3C, each device connected to the I3C bus needs to have a unique address. The dynamic address is assigned by using the CCCode ENTDAA: 0x07 (enter dynamic address assignment) during the initialization of the bus, or when a new device is connected to the I3C bus. In a system that mixes $I^2C$ devices and I3C devices, the controller should send the SETAASA: 0x29 (set all addresses to static address) before sending the ENTDAA CCC, to assign only dynamic addresses to the I3C targets. This process is called dynamic address assignment.

Once a device connected to the bus receives a dynamic address, the dynamic address shall be used in communication on the I3C bus, until and unless the controller changes, updates, or clears the dynamic address by CCC codes (SETNEWDA, RSTDAA).

## 8.3 Bus format

In the DAA process, the active controller may drive the SDA line and can start/stop the communication with a START, repeated START, or STOP at any time the I3C bus is in bus free condition.

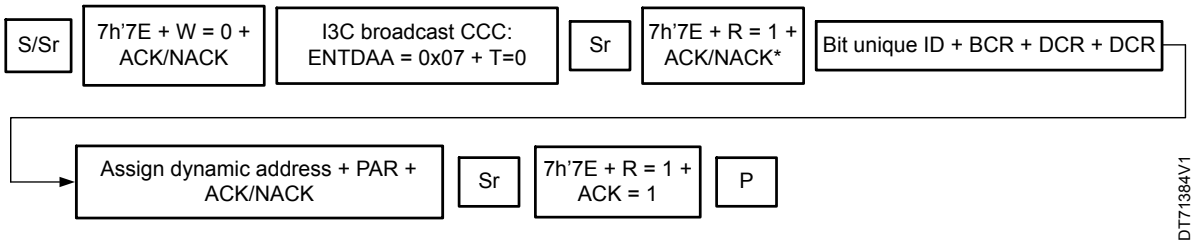
The command code for the ENTDAA broadcast CCC is 0x07. Support for this CCC is required, unless this I3C device has an I2C static address.

After sending the ENTDAA broadcast CCC, the target that supports the broadcast command code, can send its own characteristics to be identified by the controller for assignment of a dynamic address after the arbitration.

The dynamic address process shall be ended only by the active controller.

Figure 8 shows a dynamic address assignment transaction using the ENTDAA CCC.

**Figure 8. I3C broadcast ENTDAA CCC**




Where:

- *ACK/NACK: acknowledge from the addressed target without handoff. (See MIPI v1.1.5.1.2.3.1: "Transition from address ACK to SDR Controller Write Data")
- Sr: repeated start (only by the active controller)
- S: start (only by the active controller)
- P: stop (only by the active controller)
- 7h'7E: broadcast address
- T: transition bit
- PAR: parity bit

Every I3C device, which has no dynamic address already assigned on the bus, is ready to participate and respond to the I3C broadcast address shall send its own 48-bit provisioned ID, BCR, and DCR until the arbitration.

The device, whose concatenated provisioned ID, BCR, and DCR have the lowest value wins the arbitration round, due to the nature of arbitration.

Table 5. **PID, DCR, and BCR**

| 48-bit provisioned ID | BCR | DCR | - |
|---|---|---|---|
| 48 bits | 8 bits | 8bits | Lowest value concatenated provisioned ID, BCR, and DCR wins the arbitration round |

After the arbitration, the main controller assigns a 7-bit as a dynamic address to the target followed by a parity bit.

The parity bit (PAR) is calculated as an odd parity.

The *Odd parity is the inverse of the 7-bit XOR.

Therefore, ~XOR (dynamic address [7:1]) is placed in position 0.

- If the parity is valid, then the target shall acknowledge receipt of the dynamic address on the next SCL clock.
- If the parity is invalid, then the target shall passively NACK on the next SCL.

## 8.4 I3C target address restrictions

An I3C controller device assigns 7-bit dynamic addresses in the range of 7'h03 to 7'h7D with exceptions (these restrictions are illustrated in MIPI I3C).

The active controller may choose the dynamic addresses from a set of values, as follows:

Table 6. **Target address available for use**

| Target dynamic address | | Restriction | Description |
|---|---|---|---|
| Binary | Hex | | |
| 0x0000000 | 7'h00 | Shall not use | I3C reserved |
| 0x0001000 - 0x0111101 | 7'h08 – 7'h3D | Available for use | 54 addresses |
| 0x0111111 -0x1011101 | 7'h3F – 7'h5D | Available for use | 31 addresses |
| 0x101 1111-0x110 1101 | 7'h5F – 7'h6D | Available for use | 15 addresses |
| 0x110 1111-0x111 0101 | 7'h6F – 7'h75 | Available for use | 7 addresses |
| 0x111 0111 | 7'h77 | Available for use | 1 address |
| 0x111 1111 | 7'h7F | Shall not use | I3C reserved: Broadcast address single bit error detect |

Table 7. **Conditional target address**

| Target dynamic address | | Restriction | Description |
|---|---|---|---|
| Binary | Hex | | |
| 0x0000011-0x0000011 | 7'h03-7'h04 | Conditional | Available for use only if no legacy $I^2C$ devices supporting $I^2C$ "High-speed mode" are present on the bus |
| 0x1111000-0x1111001 | 7'h78-7'h79 | Conditional | Available for use only if no legacy $I^2C$ devices are present on the bus that both<br>1. Support $I^2C$ "extended address mode"<br>2. Either have an extended address, or would be impacted by the extended address mechanism |
| 0x111 1011 | 7'h7B | Conditional | Available for use only if no legacy $I^2C$ devices are present on the bus that both:<br>1. Support I2C "extended address mode"<br>2. Either have an extended address, or would be impacted by the extended address mechanism |
| 0x111 1101 | 7'h7D | Conditional | Available for use only if no legacy $I^2C$ devices supporting $I^2C$ "device ID mode" are on the bus. |

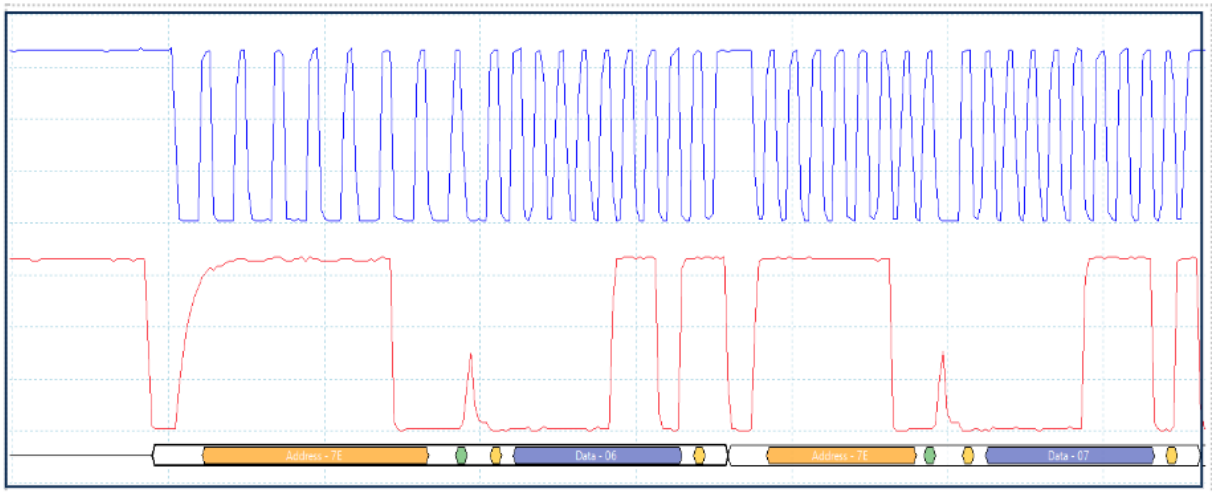
## 8.5 Changing dynamic address

There are two modes for assignment of dynamic address to only I3C devices:

1. Directly by using ENTDAA CCC 0x07.
   If a device has a dynamic address then it shall not respond to this command.
2. RSTDAA reset dynamic address assignment: 0x06 (reset the current dynamic address and wait for new assignment.)

This CCC is used to tell I3C devices to clear or reset their dynamic address. After that, the devices on the I3C bus are ready to have their dynamic address by using ENTDAA: 0x07 as shown in the below figure:

**Figure 9. RSTDAA then ENTDAA**




*Note: The device acts as a target, so check these flags:*

1. *I3C_EVR.DAUPDF: dynamic address update flag.*
2. *I3C_DEVR0.DAVAL: is cleared on the acknowledgement of the RSTDAA CCC.*

## 8.6 I3C SDR direct/broadcast CCC

### 8.6.1 I3C CCCs

These common command codes (CCC) can be sent to a specific target or all targets on the bus in the I3C bus.

So, there are two main kinds of CCC messages that allow the master to communicate with target(s) on the bus:

- Broadcast CCC messages: is seen by all targets - such as RSTDAA, ENTDAA, or DISEC).
- Direct read/write: is seen by a specific target by selecting its dynamic address (such as GETPID or GETBCR).

## 8.7 Frame format

All command codes share the same frame format for the broadcast/direct CCCs, as shown below:

- START or repeat START
- I3C reserved byte 7'h7E + W = 0 + ACK
- I3C command code [7:0] + T + optional data + T + Sr
- I3C target address + Rn = 1 + ACK
- Data [7:0] + T*
- STOP or repeated START

Figure 10. **I3C broadcast CCC transfer**

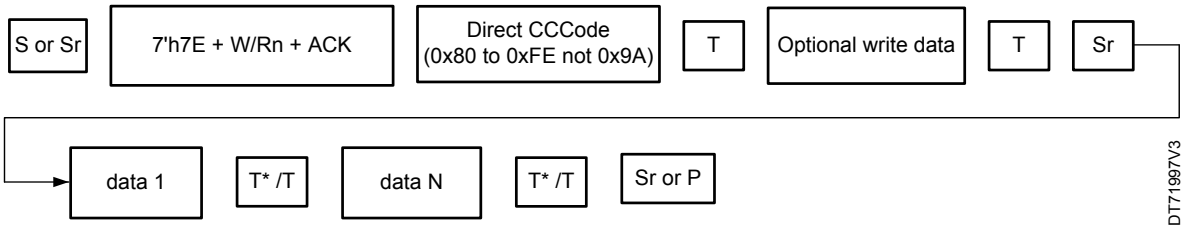
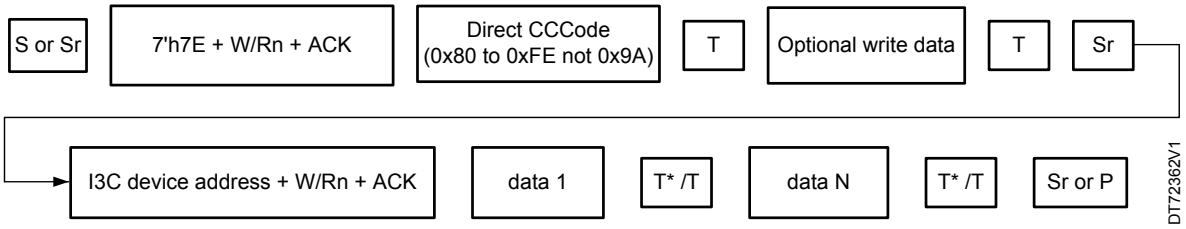



Figure 11. **I3C direct CCC transfer**




*Note:*
- *I3C broadcast CCC (from 0x00 to 0x7F, except 0x07 and 0x2A)*
- *I3C direct CCC (from 0x80 to 0xFE, except 0x9A)*

Where:
- T: Transition bit (parity bit)
- T*: Transition bit (end of data for read data)
- Optional write data: this field is used with some broadcast CCC messages, and for some direct read/write CCC messages. It is followed by a T-bit.

For some direct CCCs, a defining byte is optional; for others, it is always required.

## 8.8 Supported common command codes

Table 8. **List of supported CCCs supported by LSM6DSO**

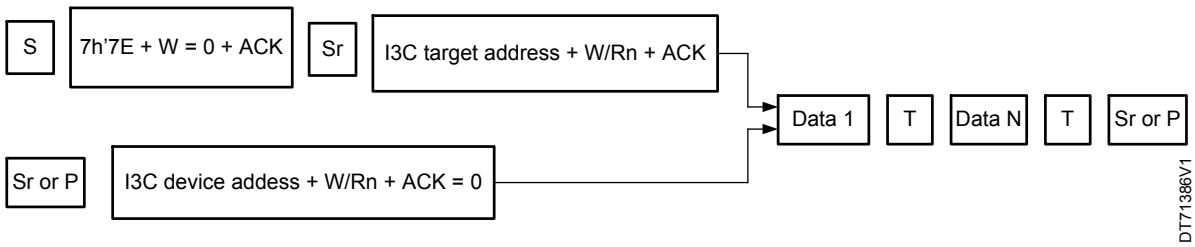
| Command name | Command code | CCC type |
|---|---|---|
| ENTDAA | 0x07 | Broadcast |
| SETDASA | 0x87 | Direct |
| ENEC | 0x00 | Broadcast |
| | 0x80 | Direct |
| DISEC | 0x01 | Broadcast |
| | 0x81 | Direct |
| ENTAS 0...3 | 0x82 to 0x85 | Direct |
| | 0x02 to0x05 | Broadcast |
| SETXTIME | 0x28 | Broadcast |
| | 0x98 | Direct |
| GETXTIME | 0x99 | Direct |
| RSTDAA | 0x06 | Broadcast |
| | 0x86 | Direct |
| SETMWL | 0x08 | Broadcast |
| | 0x89 | Direct |

| Command name | Command code | CCC type |
|---|---|---|
| SETMRL | 0x09 | Broadcast |
| | 0x8A | Direct |
| SETNEWDA | 0x88 | Direct Set |
| GETMWL | 0x8B | Direct Get |
| GETMRL | 0x8C | Direct Get |
| GETPID | 0x8D | Direct Get |
| GETBCR | 0x8E | Direct Get |
| GETDCR | 0x8F | Direct Get |
| GETSTATUS | 0x90 | Direct Get |
| GETMXDS | 0x94 | Direct Get |

## 8.9 SDR private transfer

This section illustrates I3C private read/write messages as shown in the figure below:

**Figure 12. SDR private transfer**




Where:

- ACK = Acknowledge (SDA low)
- S = START condition
- Sr = Repeated START condition
- P = STOP condition
- T = Transition bit

The private transfer begins with a start from the controller or a repeated start after a previous transfer and then the controller issues the broadcast header address followed by the target address or directly by sending the device address.

- SDR private read: T = 1 informs the controller that there is additional data, whereas T = 0 signals the end.
- SDR private write: T is considered as a parity bit: odd parity.

## 8.10 In-band interrupt (IBI)

To generate an interrupt by the target on the I3C bus for the controller without an external interrupt line, the I3C protocol allows the target to initiate the bus after the bus available duration.

### 8.10.1 Address arbitration

The address arbitration is an address following a START, used for the arbitration in open-drain mode, when both two devices (may be two concurrent targets and/or a concurrent target with the controller) are requesting a communication message on the I3C bus at the same time.

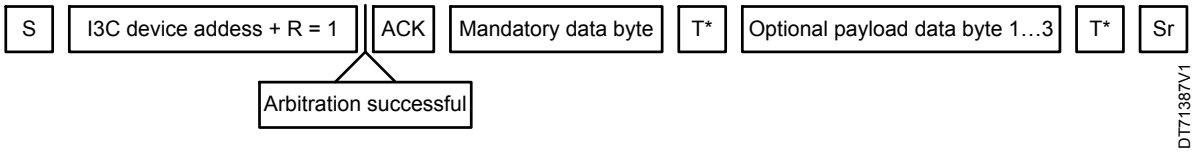
If two devices are sending their own address (it may be the reserved address 7'h7E + RnW = 0 from the controller), the first device, which is not actively pulling SDA low and instead is issuing passively a '1' bit (high-Z) starting from the most significant bit until the low significant bit of the address, is the device, which loses the arbitration phase and then should keep listening to the I3C bus.

Alternatively, a target may initiate a start request while the I3C bus is idle by pulling the SDA low while SCL is high. Then the controller must activate the SCL clock after tCAS for enabling the target to issue its IBI request and provide its own target address.

### 8.10.2 IBI process and arbitration

During the IBI process, the active controller shall provide a START and after the target sends its address with a read. After the arbitration, if the controller provides an ACK, and the target sends the mandatory data payload if any, or may send the optional payload data byte as presented in the figure below:

**Figure 13. Successful IBI sequence with mandatory data byte**




*Note: The arbitration is successful if more than targets on the bus simultaneously request an interrupt.*

Where:

- ACK: controller ACK.
- T*: transition bit (end of data for read data) from controller and/or from target.

While requesting for IBI, the target may lose the arbitration or may be not acknowledged by the controller, and then it may continue to attempt the IBI request until the procedure is successfully completed, or it can choose optionally do not try again.

### 8.10.3 Mandatory data byte

The mandatory data byte is the data that follows the target address during the IBI procedure after the controlled ACKs the request. It gives more information about the interrupt and this byte is divided in two parts (refer to MIPI V1.1.1 tab13):

- **Specific interrupt identifier**: MDB [4:0].
- **Interrupt group identifier**: MDB [7:5].

This byte depends on the value of the bit BCR 2 (IBI payload):

- 0: no data byte followed the IBI.
- 1: at least one mandatory data byte follows the accepted IBI (and at most four data bytes).

## 8.11 Legacy $I^2C$ on the I3C bus

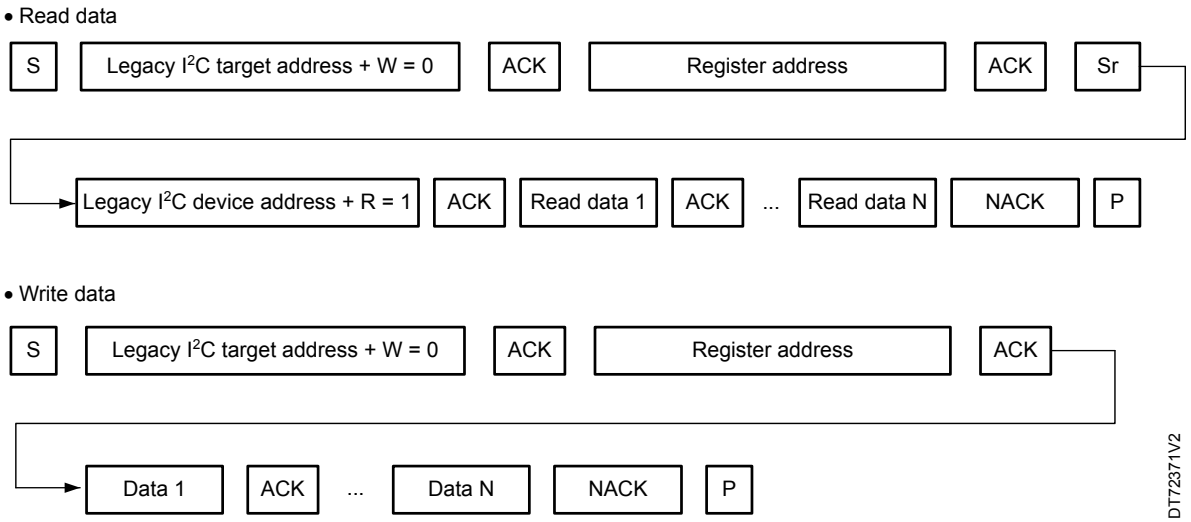
The I3C protocol supports existing $I^2C$ target devices and messages on the I3C bus with some limitations:

- If a spike filter is implemented on the $I^2C$ devices connected to the bus, it means that the I3C can operate at the maximum frequency.
- If the spike filter is not implemented in any $I^2C$ target device, the I3C bus allows the slowest frequency of the legacy $I^2C$ and can eliminate certain I3C bus features.
- The extended address with 10 bits is not used on the I3C bus.
- The legacy $I^2C$ target does not perform the clock stretching.

### 8.11.1 Frame format

The figure below presents both a legacy I²C and typical read register-based device transfer (write register address followed by data reads), and a legacy I²C typical write register-based device transfer (write register address followed by data writes).

**Figure 14. Legacy I²C transfer on I3C bus**




Where:

- S = START
- Sr = repeated START
- P = STOP
- ACK = acknowledge
- NACK = not acknowledged

# 9 Examples of I3C communications with an STM32H503RB Nucleo board

This section demonstrates how to use STM32CubeMX to create an I3C serial communication application using NUCLEO-H503R8 and X-NUCLEO-IKS01A3 boards, and provides an easy guide for example implementation.

## 9.1 STM32Cube firmware examples

The STM32CubeH5 and STM32xx firmware packages offer a large set of examples implemented and tested on the corresponding boards.

**Table 9. Firmware available examples**

| Firmware package | Example name | Board |
|---|---|---|
| STM32CubeH5 | I3C_controller_ENTDA_IT<br>I3C_Target_ENTDAA_IT<br>I3C_Controller_Private_Command_IT<br>I3C_Target_Private_Command_IT<br>I3C_Controller_InBandInterrupt_IT<br>I3C_Target_InBandInterrupt_IT<br>I3C_Target_WakeUpFromStop<br>I3C_Controller_I2C_Com<br>I3C_Target_I2C_Com<br>I3C_Controller_Switch_To_Target<br>I3C_Target_Switch_To_Controller<br>I3C_Controller_HotJoin_IT<br>I3C_Target_HotJoin_IT | NUCLEO-H503RB |

## 9.2 I3C examples based on STM32CubeMx

This section illustrates six typical examples using the I3C device as controller and target:

- ENTDAA.
- Direct read.
- Private read.
- In-band interrupt.
- Mixed communication.

## 9.3 Hardware and software settings

The table below lists the development environment details.

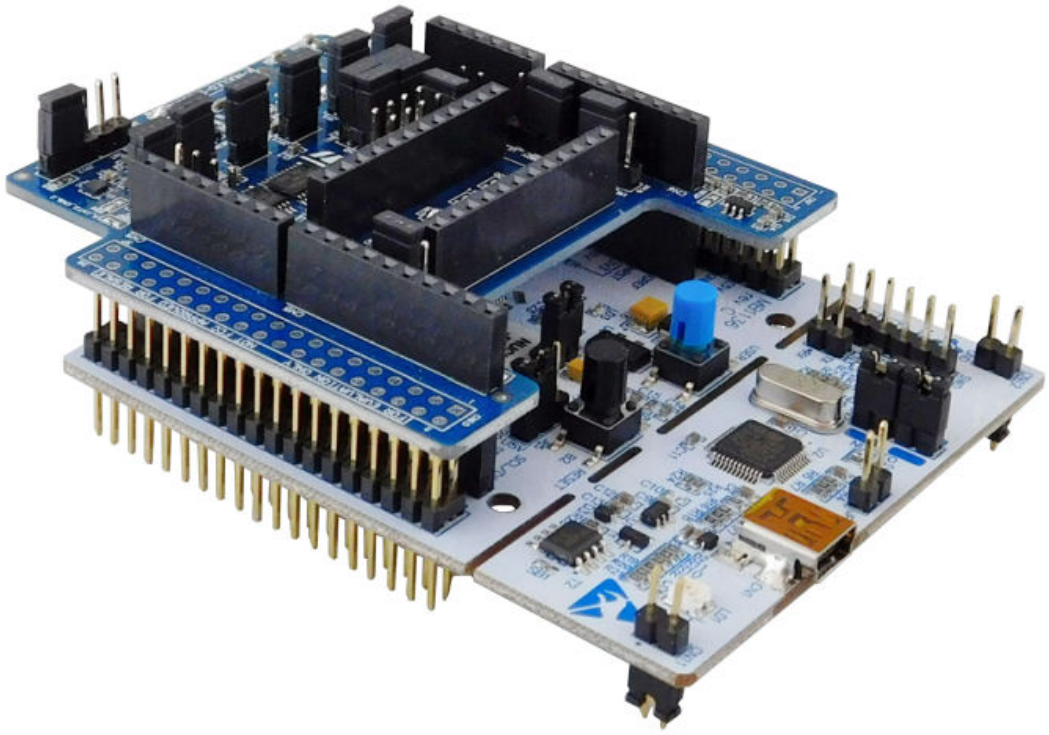
**Table 10. Software and hardware environment**

| Use case | Hardware | Software |
|---|---|---|
| Dynamic addressing | STM32H503 board (controller I3C)<br>Accelerometer and gyroscope LSM6DSO (target I3C)<br>• Keep the JP2 | STM32CubeMX<br>STM32Cube_FW_H5_V1.0.0 or higher<br>IAR Systems |
| Direct transfer | | |
| Private transfer | | |
| IBI | STM32H503 board (controller I3C)<br>STM32H503 board (target I3C) | |

| Use case | Hardware | Software |
|---|---|---|
| Mixed Bus | STM32H503 board (controller I3C)<br>STM32H503 board (target I3C)<br>Accelerometer LIS2DW12 (legacy $I^2C$)<br>• Keep the JP1 | STM32CubeMX<br>STM32Cube_FW_H5_V1.0.0 or higher<br>IAR Systems |

The X-NUCLEO-IKS01A3 board is compatible with STM32 Nucleo boards and interfaces with the STM32 microcontroller via the I2C/I3C pins.

**Figure 15. X-NUCLEO-IKS01A3 plugged on an STM32 Nucleo board**



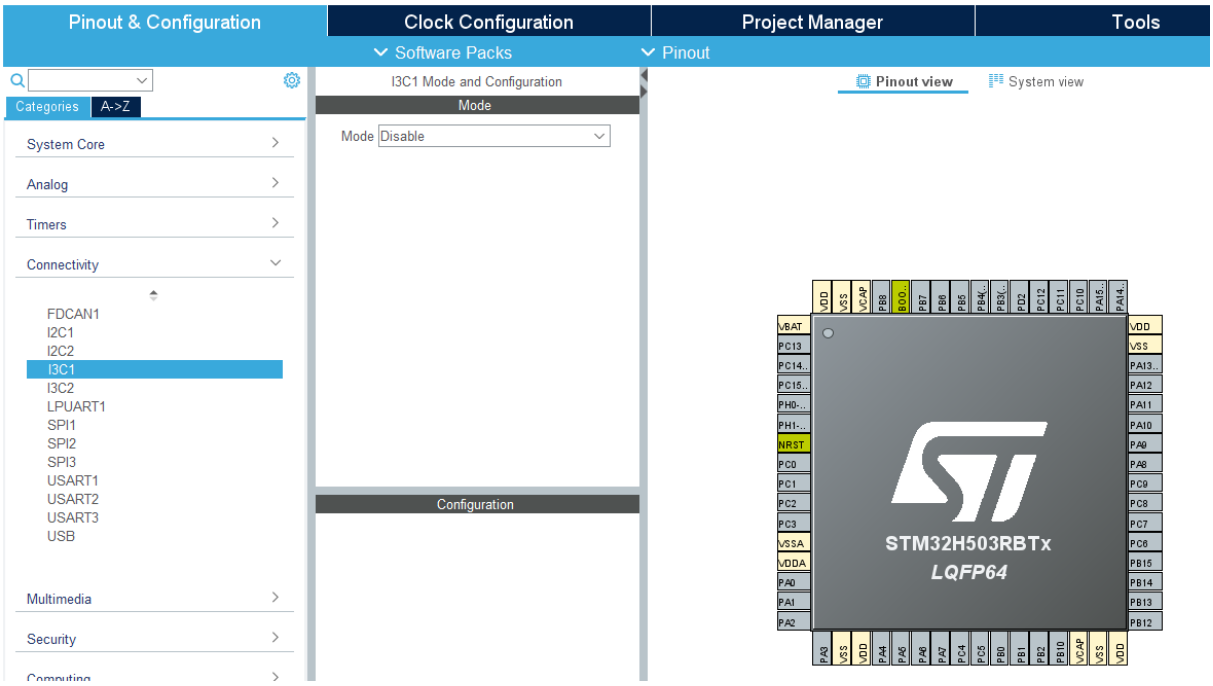
## 9.4 Common configuration

### 9.4.1 STM32CubeMX - I3C GPIO configuration

Open STM32CubeMX, select “Access to MCU selector", and execute the following steps:

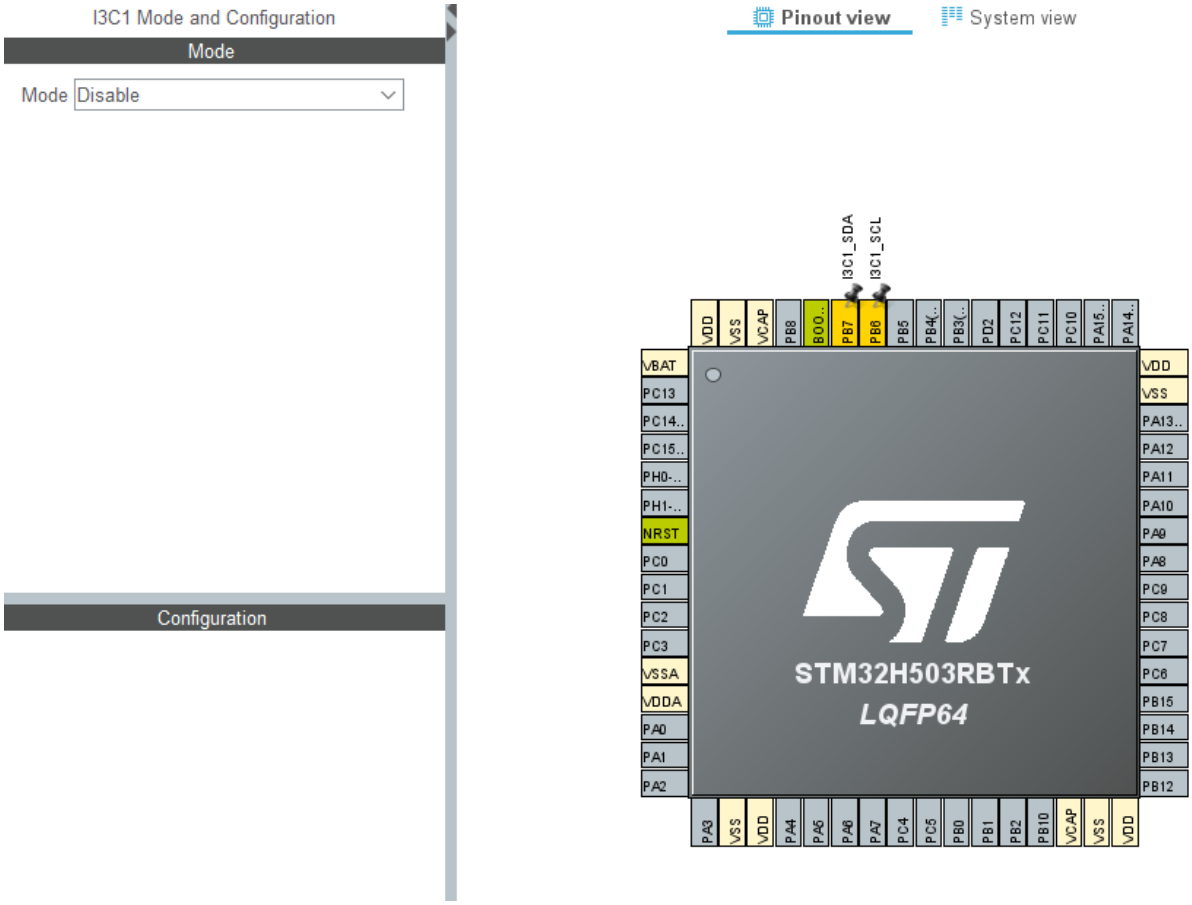
- Creating a new STM32CubeMX project and selecting the corresponding MCU.
- Click on “Connectivity > I3C1”.

**Figure 16. I3C1 selection**




- Selecting the right pins:
  - PB6: I3Cx_SCL
  - PB7: I3Cx_SDA

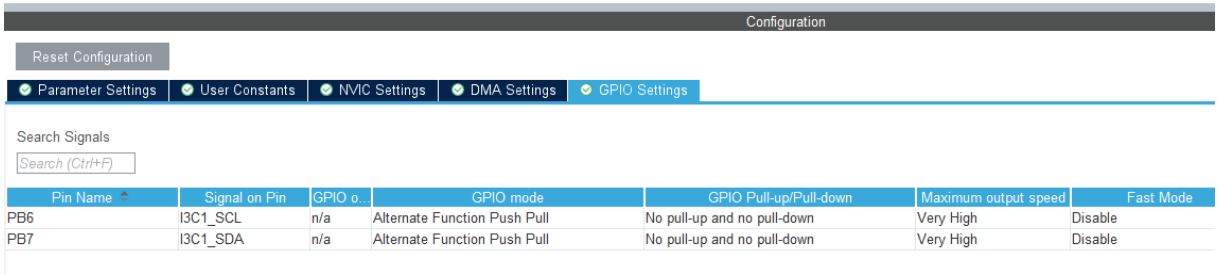
Figure 17. **I3C1 pins**




- GPIOs settings:

**For PB6 and PB7 no pull needed. See the figure below.**

Figure 18. **I3C1 GPIOs configuration**



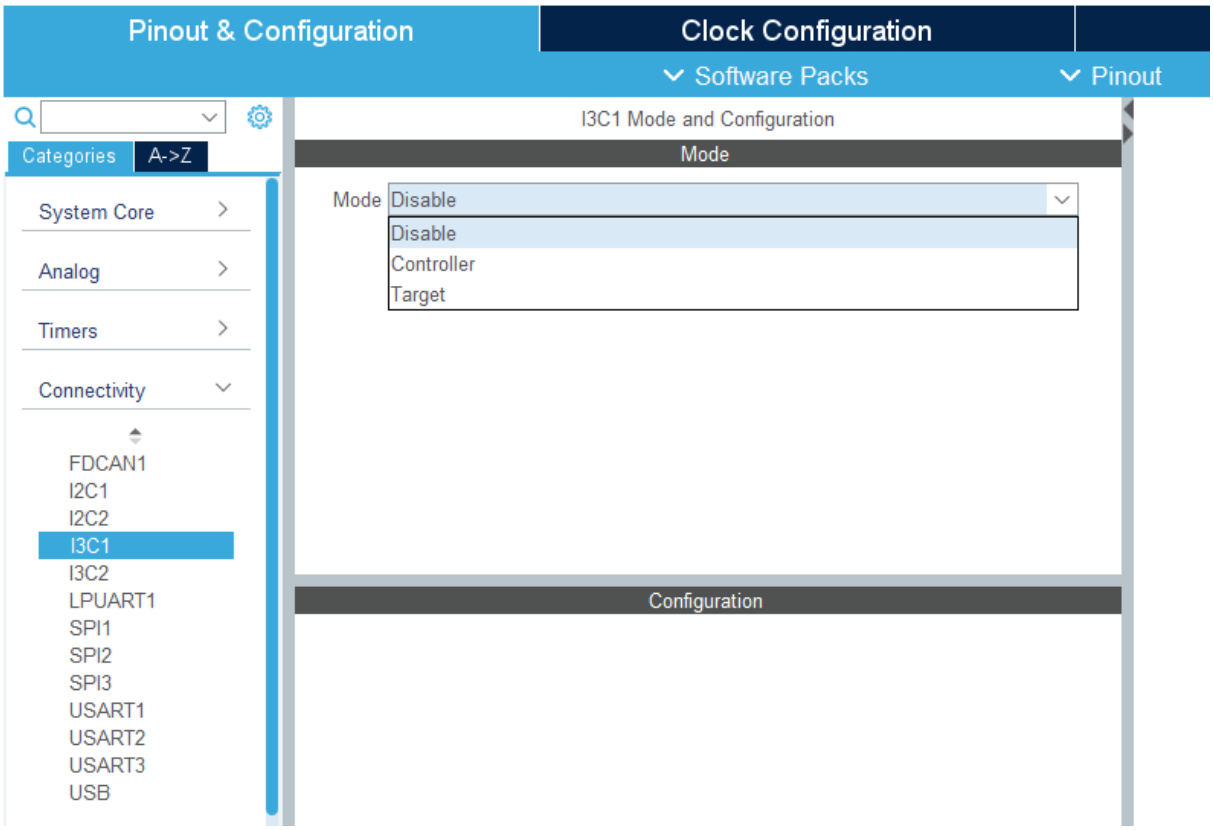

| Pin Name | Signal on Pin | GPIO o... | GPIO mode | GPIO Pull-up/Pull-down | Maximum output speed | Fast Mode |
|---|---|---|---|---|---|---|
| PB6 | I3C1_SCL | n/a | Alternate Function Push Pull | No pull-up and no pull-down | Very High | Disable |
| PB7 | I3C1_SDA | n/a | Alternate Function Push Pull | No pull-up and no pull-down | Very High | Disable |

*Note:*

- *As a workaround, the user can enable the pull-up at the controller startup phase, then no pull after a delay.*
- *No pull-up needed for the target.*

**Select controller or target mode**

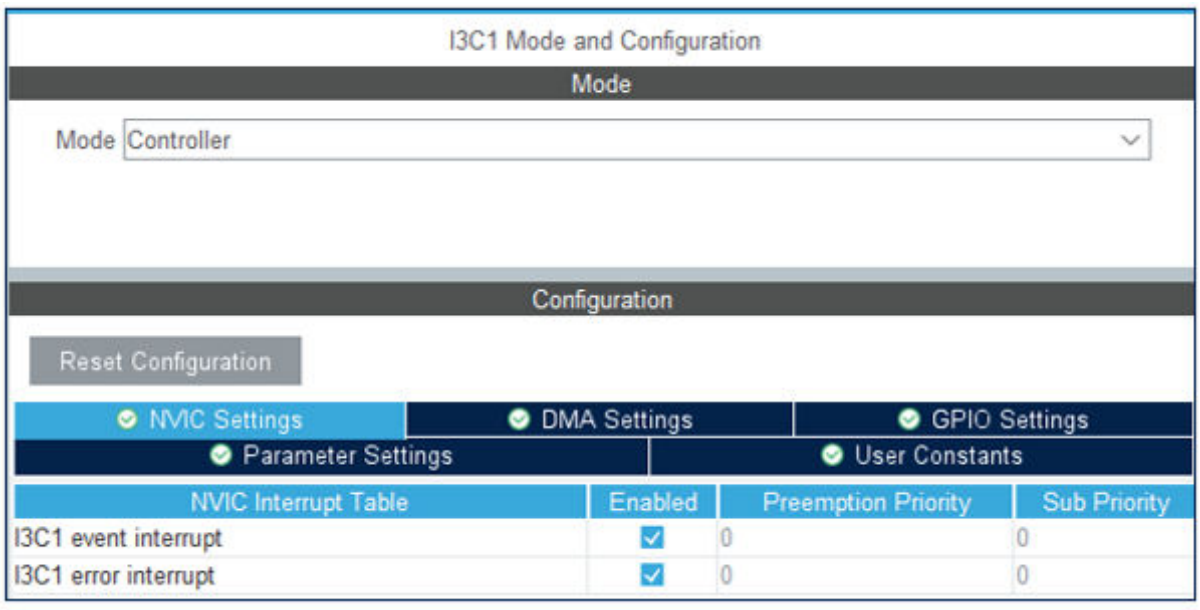
**Figure 19. I3C mode selection**




### 9.4.2 STM32CubeMX - I3C interrupts

Select the NVIC settings tab configuration window and check the I3C1 event interrupt, as shown in the following figure:

**Figure 20. Enable interrupt**



## 9.5 STM32CubeMX – I3C controller mode

This part describes how to set parameters for I3C in controller mode. The user can configure frequency, duty cycle, and bus usage.

### 9.5.1 Bus characteristics

For I3C bus max frequency communication, it depends on the hardware constraints. In the examples, there are two uses cases:

- The examples using the sensor shield can reach only 3 MHz.
- The examples using Nucleo boards can reach maximum I3C frequency: 12.5 MHz.

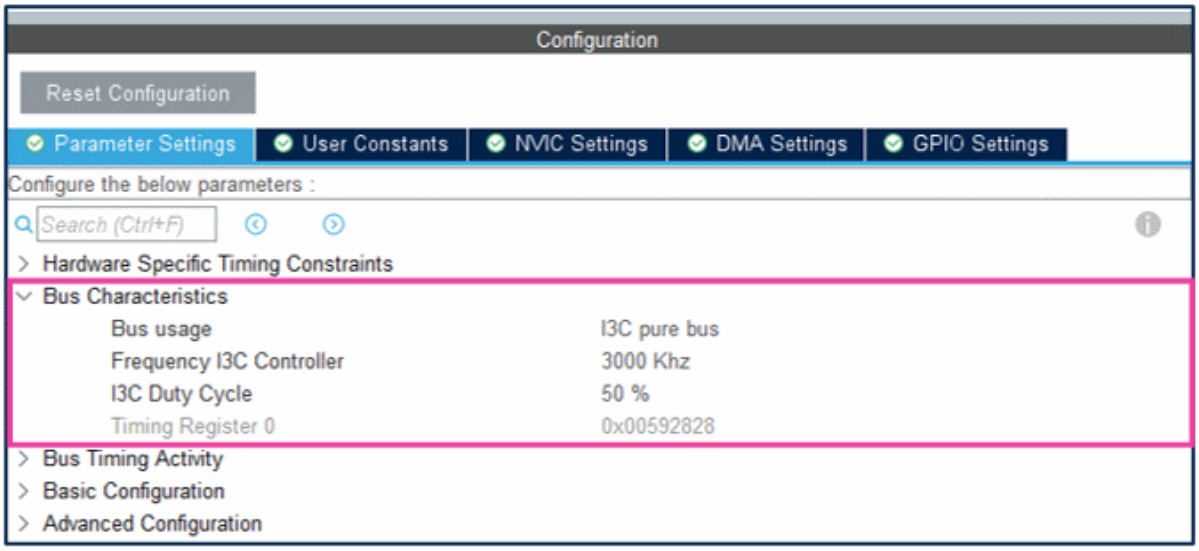
### 9.5.2 Communication at 3 MHz push pull

- Bus usage: I3C pure bus
- The communication runs at 3 MHz
- Timing register 0: 0x00552828

These values are calculated automatically by STM32CubeMX:

- Bits 31:24 = 0x00: SCL high duration for legacy $I^2C$ messages: no $I^2C$ on bus
- Bits 23:16 = 0x55: SCL low duration in open drain phases
  $t_{SCLL\_OD} = (SCLL\_OD + 1)xt_{I3CCLK} = (85 + 1)*(1/250) = 344$ ns
- Bits 15:8 = 0x28: SCL high duration for I3C messages.
  $t_{SCLH\_I3C} = (SCLH\_I3C + 1)xt_{I3CCLK} = (40 + 1)*(1/250) = 164$ ns
- Bits 7:0 = 0x28: SCL low duration in I3C push-pull phases.
  $t_{SCLL\_PP} = (SCLL\_PP + 1)xt_{I3CCLK} = (40 + 1)*(1/250) = 164$ ns

**Figure 21. Bus characteristics for 3 MHz frequency**




*Note: The definition of the duty cycle depends on user hardware constraints; it impacts open-drain phases and ensures good SDA signal quality. The higher the frequency, the more reduced the duty cycle.*

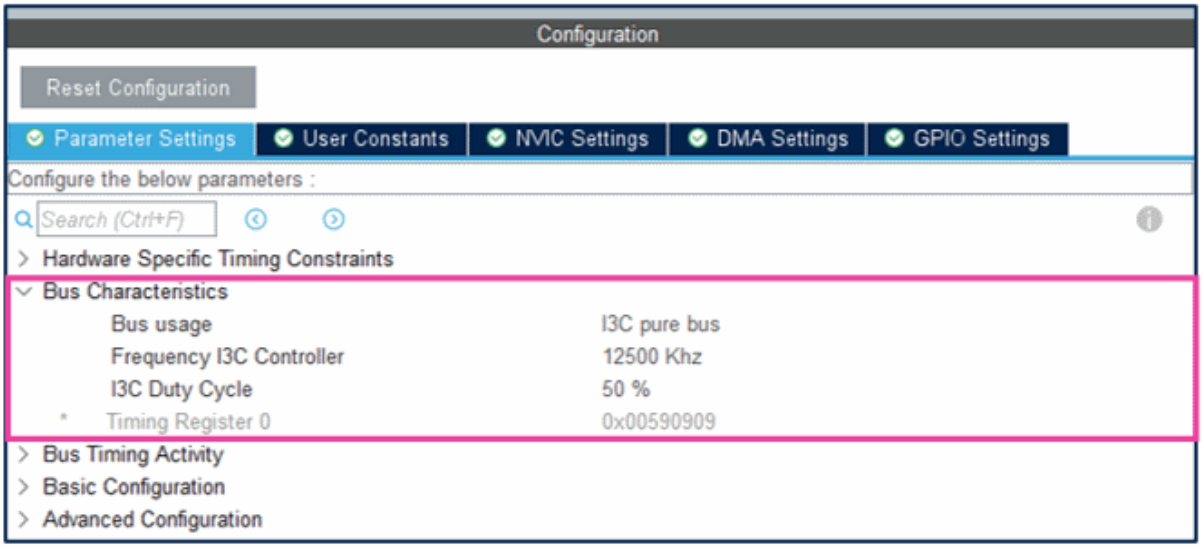
### 9.5.3 Communication at 12.5 MHz push pull

These values are calculated automatically by CubeMX:

- Bus usage: I3C pure bus
- The communication runs at 12.5 MHz

- Timing register 0: 0x00550909
  - Bits 31:24 = 0x00: SCL high duration for legacy $I^2C$ messages: no $I^2C$ on bus.
  - Bits 23:16 = 0x55: SCL low duration in open drain phases.
    $t_{SCLL\_OD}$ = (SCLL_OD + 1)x$t_{I3CCLK}$ = (85 + 1)*(1/250) = 344 ns
  - Bits 15:8 = 0x09: SCL high duration for I3C messages (both in push-pull and open-drain phases).
    $t_{SCLH\_I3C}$ = (SCLH_I3C + 1)x$t_{I3CCLK}$ = (09 + 1)*(1/250) = 40 ns
  - Bits 7:0 = 0x08: SCL low duration in I3C push-pull phases.
    $t_{SCLL\_PP}$ = (SCLL_PP + 1)x$t_{I3CCLK}$ = (09 + 1)*(1/250) = 40 ns

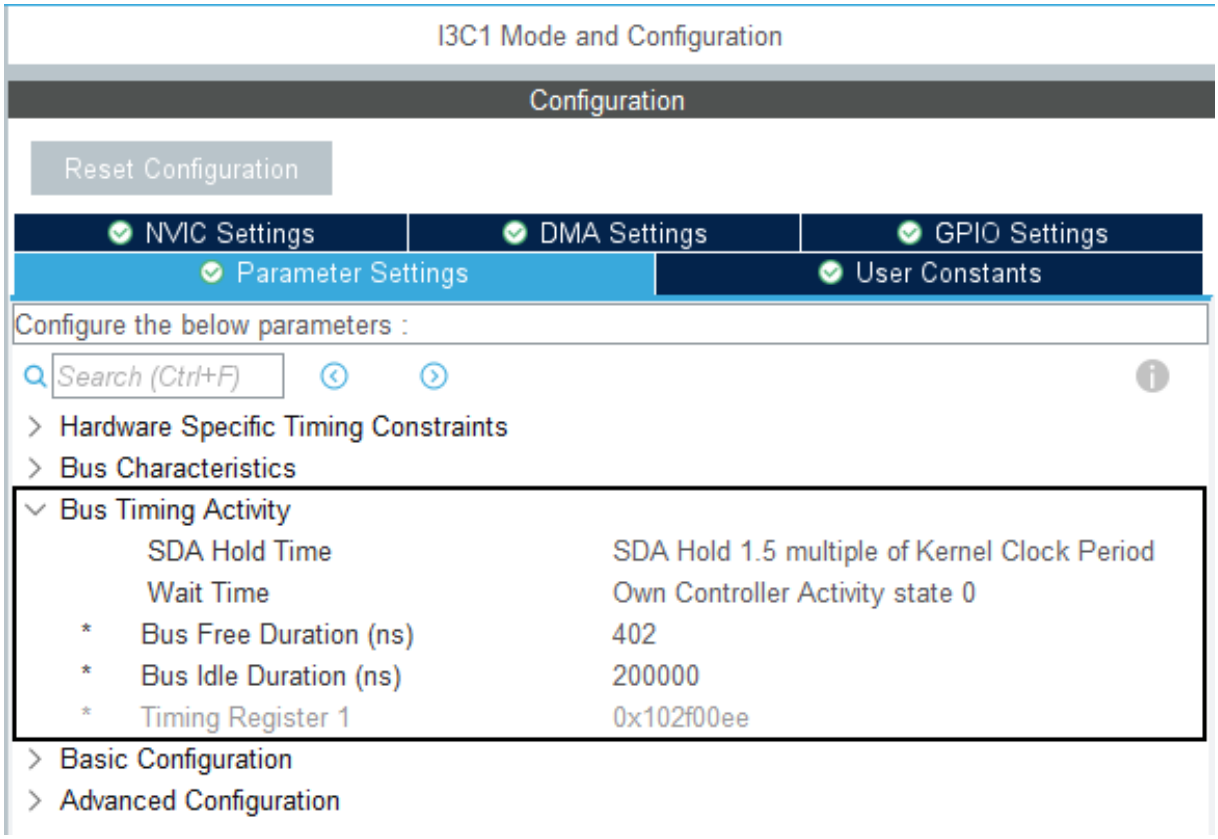
**Figure 22. Bus characteristics for 12.5 MHz frequency**




### 9.5.4 Bus timing

The figure below shows the bus timing configuration in the next examples.

Figure 23. **Bus timing activity**




**Timing register 1: 0x103200f8**

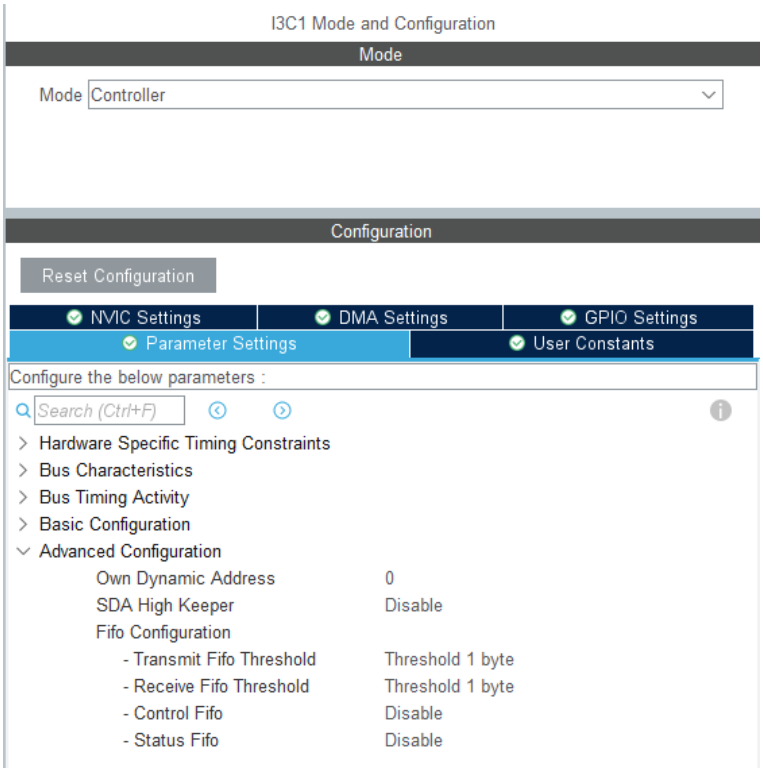
- Bit 28 SDA_HD (timing register 1) = “1”: SDA hold time = 1.5 * $t_{I3CCLK}$
- Bits 27:23: reset value “0”
- Bits 22:16 = FREE [6:0] = 0x02F
- $t_{CAS}$ = [(FREE [6:0] +1) * 2 - (0.5 + SDA_HD)] * $t_{I3CCLK}$
- Bus free duration =
  $t_{CAS}$ = [([6:0] +1) * 2 - (0.5 + SDA_HD)] * $t_{I3CCLK}$
  $t_{CAS}$ = [(47 + 1) * 2 - 1.5] * (1 / 250 MHz)
  $t_{CAS}$ = 378 ns
- Bits 15:0: reset value = 0x0000
- Bits 9:8: 0x00, no new controller
- Bits 7: 0 = 0xEE: t_AVAL = (AVAL [7 :0] + 2) x $t_{I3CCLK}$ and t_IDLE = t_AVAL * 200:
  - Bus available duration = 1µs
  - Bus idle duration = 200µs
  - The value of the bus free duration set on this example is a link to the Nucleo hardware environment.
- Wait time: own controller activity state 0
  - 00: activity state 0 is the initial activity state of this I3C before and when becoming controller.

### 9.5.5 FIFO

This figure shows the configuration of the FIFO for data that transmitted and received byte by byte.

- Transmit FIFO threshold: threshold 1 byte.
- Receive FIFO threshold: threshold 1 byte.

**Figure 24. FIFO configuration**




*Note: In the case of applications that need the best performance, the user can choose 4 bytes as the threshold to lower the CPU and/or the DMA load, and lower the system bus load.*

### 9.5.6 System clock configuration

To configure the system clock, select the system clock at 250 MHz and follow the below instructions:

- Select HSI from PLL1 source mux.
- Select PLLCLK from system clock mux.
- Set HCLK at 250 MHz.
- Select PCLK from the I3C1 clock mux.

PLLM = 4, PLLN = 31, PLLP = 2, PLLQ = 2, PLLR = 2, APB1 prescaler =1.

**Figure 25. Clock tree**

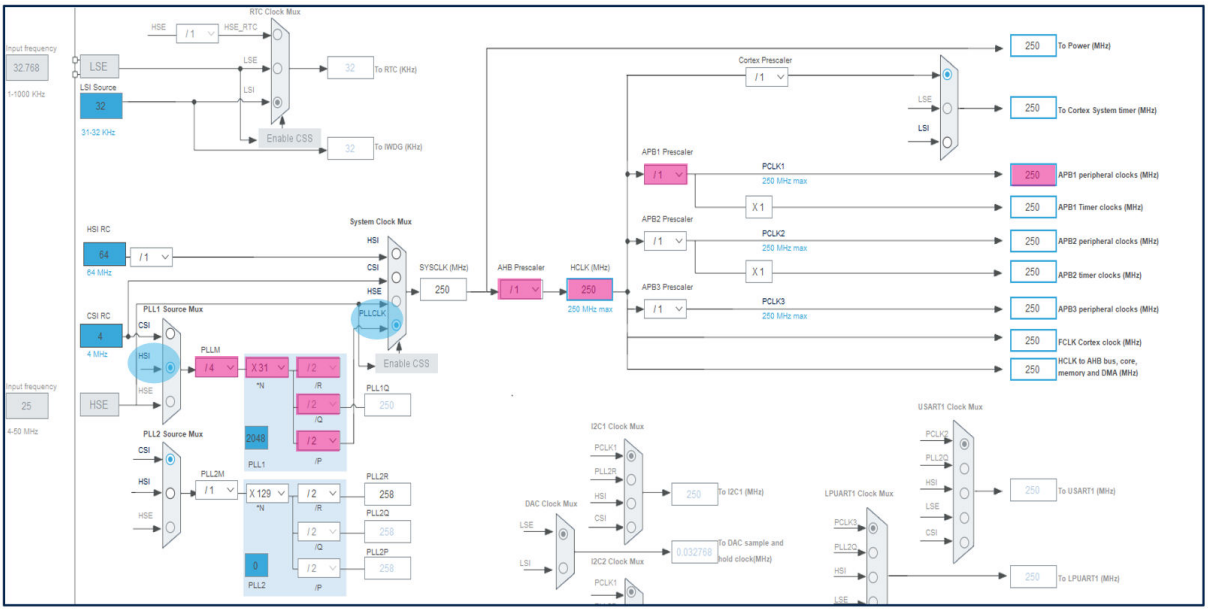
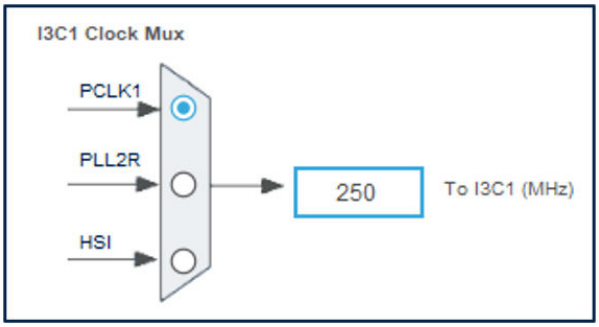



**Figure 26. I3C1 clock multiplexer**

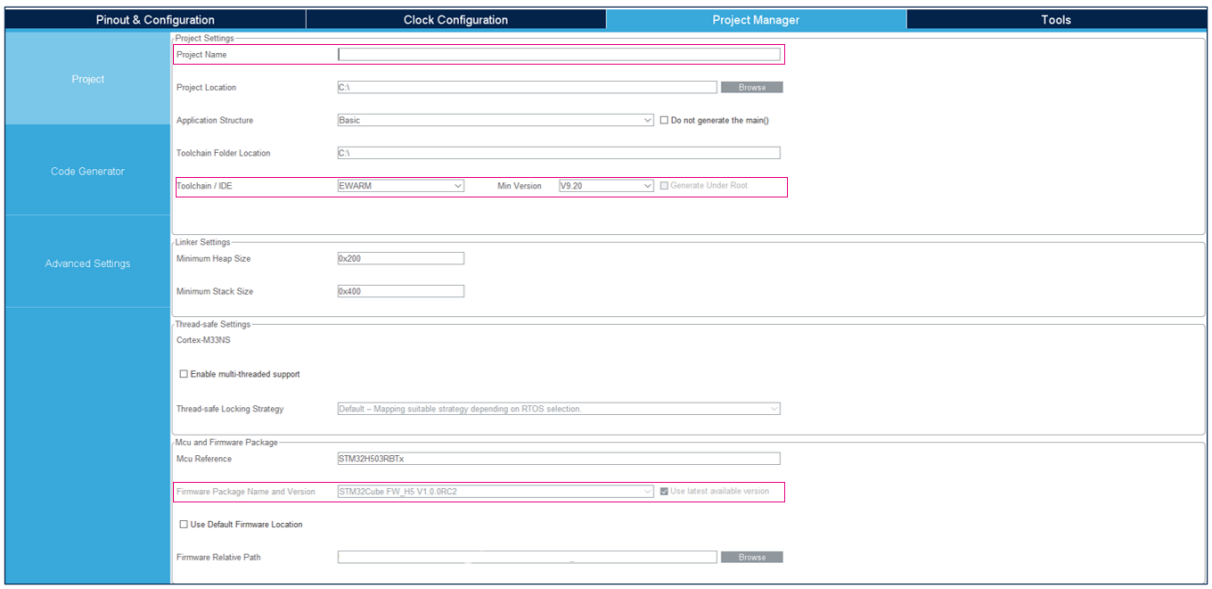


### 9.5.7 Project generation

This part demonstrates the steps to follow to generate the user code initialization:

- Select project manager.
- Name the project.
- Select the tool-chain.
- Select the firmware package.
- Generate the project.

**Figure 27. Project manager**




## 9.6 Dynamic addressing

This sample application shows how to use the I3C protocol to assign a dynamic address to a target on the I3C bus using interrupt mode.

### 9.6.1 Software settings

This section enumerates the main software routines added by the user to run dynamic addressing for the controller side.

Once the project files are created, the user should add functions outlined below.

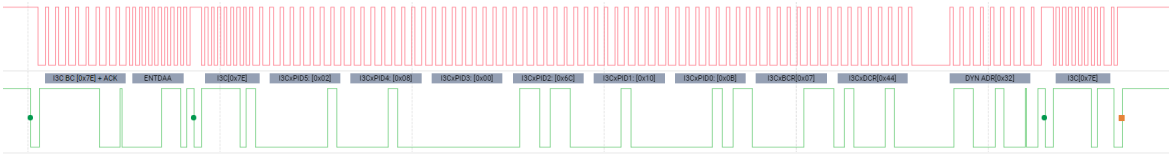
**Table 11. Software settings for dynamic addressing**

| | |
|---|---|
| `#define TARGET1_DYN_ADDR 0x32`<br><br>`/*********************/`<br>`/* Target Descriptor */`<br>`/*********************/`<br>`TargetDesc_TypeDef TargetDesc1 =`<br>`{`<br>`"TARGET_ID1",`<br>`TARGET_ID1,`<br>`0x0000000000000000,`<br>`0x00,`<br>`TARGET1_DYN_ADDR,`<br>`};`<br><br>`#endif /* __STM32_I3C_DESC_TARGET1_H */` | Target descriptor + dynamic address to assign |
| `/* USER CODE BEGIN PV */`<br><br>`/* Array contain targets descriptor */`<br>`TargetDesc_TypeDef *aTargetDesc[1] = \`<br>`{`<br>`&TargetDesc1, /* TARGET_ID1 */`<br>`};`<br><br>`/* USER CODE END PV */` | The array contains the target descriptor |
| `/* Assign dynamic address processus */`<br>`if (HAL_I3C_Ctrl_DynAddrAssign_IT(&hi3c1, I3C_ONLY_ENTDAA) != HAL_OK)`<br>`{`<br>`/* Error_Handler() function is called when error occurs. */`<br>`Error_Handler();`<br>`}` | Assigning a dynamic address in the interrupt mode |
| `/* USER CODE BEGIN 4 */`<br>`void HAL_I3C_TgtReqDynamicAddrCallback(I3C_HandleTypeDef *hi3c, uint64_t targetPayload)`<br>`{`<br><br>`TargetDesc1.TARGET_BCR_DCR_PID = targetPayload;`<br><br>`HAL_I3C_Ctrl_SetDynAddr(hi3c, TargetDesc1.DYNAMIC_ADDR);`<br><br>`}`<br><br>`void HAL_I3C_CtrlDAACpltCallback(I3C_HandleTypeDef *hi3c)`<br>`{`<br>`HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);`<br>`}`<br><br>`/* USER CODE END 4 */` | • GET target characteristics<br>• Controller set the DA |

### 9.6.2 Waveform results

The user can also check this example by verifying the waveforms of SDA and SCL with the oscilloscope as shown below:

**Figure 28. Dynamic addressing using ENTDAA CCC**




**ENTDAA with dynamic address 0x32**

S + (7h'7E + w = 0 + ACK = 0) **OD** +

ENTDAA (0x07 + T = 0) **PP** +

Sr + (7h'7E + R = 1 + ACK = 0) **PP** + target characteristics (48 provisioned ID + BCR + DCR) OD + dynamic address (0x32 + PAR = 0 + ACK = 0) **OD** +

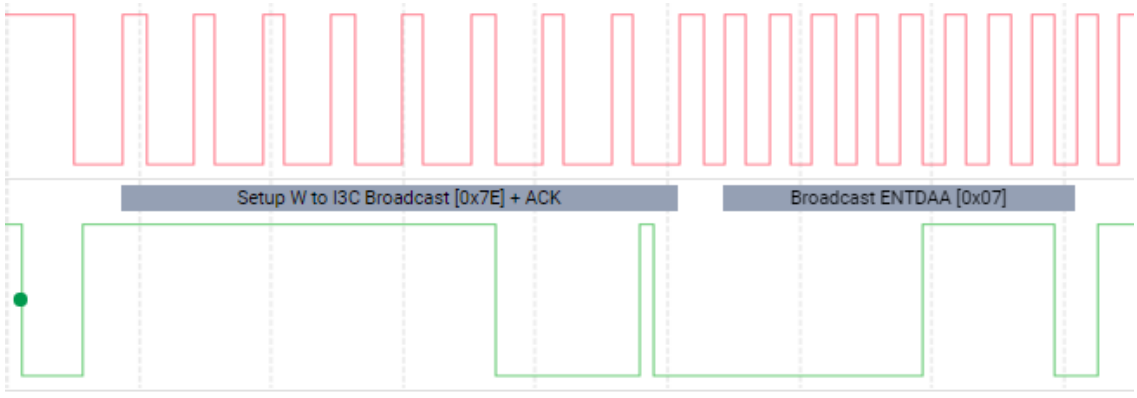
(7h'7E + R = 1 + ACK = 1) + STOP **PP**

**Table 12. Communication frequency**

| Mode | Open Drain | Push-Pull |
|---|---|---|
| Frequency | 1.904 MHz | 3.040 MHz |

1. Address header: **START + (7'h7E + W = 0 + ACK = 0) (OD phase)**
2. I3C broadcast CCC (ENTDAA = 0x07): ENTDAA (0x07 + T = 0)

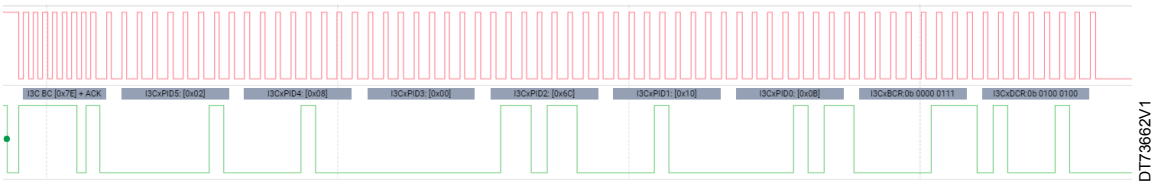
**Figure 29. ENTDAA CCC**




*Note: 0x07 is the ENTDAA value from the list of supported I3C CCCs.*
*T: Transition bit (parity bit for CCC) from the controller (drives SDA in push-pull).*

3. Target characteristics: PID, BCR, and DCR **(OD phase)**

**Figure 30. Target characteristics**



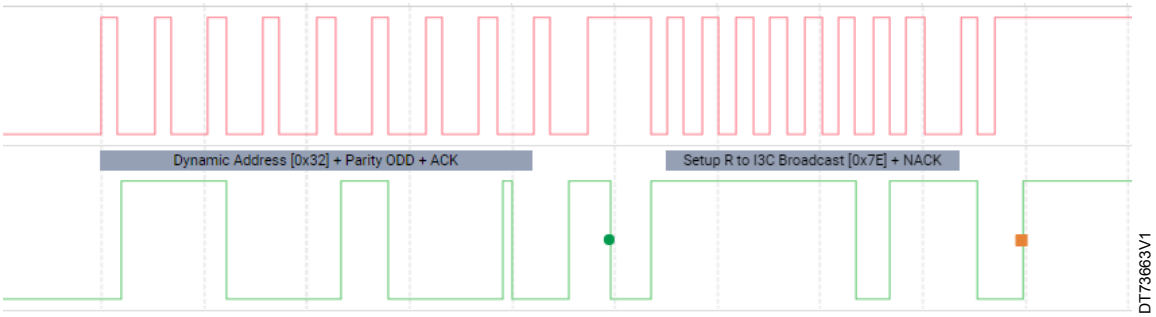

– **BCR = 0x07 = 0x0000 0111**
  - Bits [7:6] = 00: I3C target.
  - Bit 5 = 0: no support of optional advanced capabilities.
  - Bit 4 = 0: not a virtual target.
  - Bit 3 = 0: the device always responds to the I3C bus commands.
  - Bit 2 = 1: IBI payload enable.
  - Bit 1 = 1: IBI request capable (depend on the hardware).
  - Bit 0 =1: limitation max data speed.
– **DCR: 0x44 (default) MIPI I3C device characteristic register = combo (accelerometer and gyroscope)**
– 48-bit provisioned ID:

**Table 13. 48-bit provisioned ID**

| Value | Signification |
|---|---|
| Bits [47:33] = 000000100000100 | MIPI manufacturer ID |
| Bits: 32 = 0 | Provisioned ID type selector: 0b0 = vendor fixed value |
| Bits [31:16] = 0000000001101100 | Part ID |
| Bits [15:12] = 0001 | Instance ID |
| Bits [11:0] = 0001000000001011 | Reserved, must be kept at reset value |

4. Dynamic address: **Dynamic address (0x32 + PAR = 0 + ACK = 0): (OD phase)**
5. I3C broadcast address + R = 1 + STOP: (0x7E + R = 1 + ACK = 1) + STOP **(OD phase)**

**Figure 31. Dynamic addressing**



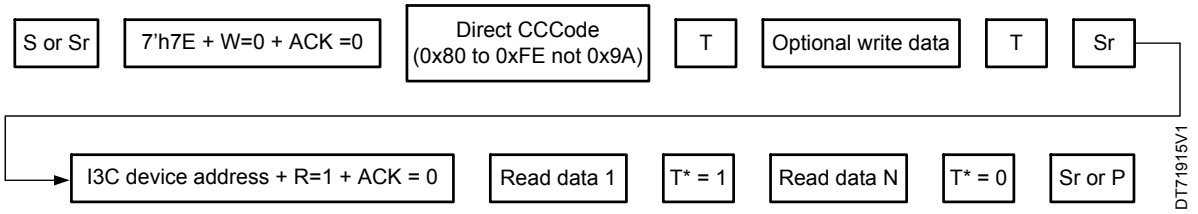

## 9.7 Direct read

This section contains the steps of usage of a typical direct read based on STM32CubeMX. This example shows how to get a data from an I3C device using common command codes in SDR mode.

The intent of this use case is to illustrate some command common codes that supports LSM6DSO like GETBCR, GETDCR, GETMRL with/without defining byte.

In this example, the ranging data are displayed on the oscilloscope. Figure 32 shows the controller issuing a direct read to one target on the bus. All commands have the same flow for the direct CCC.

**Figure 32. I3C GET CCC message**




Where:

- T: Transition bit (parity bit for write data) from controller
- T*: Transition bit (end of data for read data) from controller and/or from target

### 9.7.1 Direct read without defining byte

#### 9.7.1.1 *Software settings*

After the dynamic addressing process, add this code to use direct read:

**Table 14. Software settings without defining byte**

| | |
|---|---|
| `#define Direct_GETDCR 0x8F`<br><br>`/* Descriptor for direct CCC */`<br>`I3C_CCCTypeDef aCCCList[] =`<br>`{`<br>`{TARGET1_DYN_ADDR,Direct_GETDCR,{NULL,1},LL_I3C_DIRECTION_READ },`<br>`};`<br>DT72305V1 | • I3C CCCTypeDef structure definition<br>• GETDCR CCCode: 0x8F |
| `/* Prepare context buffers process */`<br>`buffer[I3C_IDX_FRAME_1].CtrlBuf.pBuffer = aControlBuffer;`<br>`buffer[I3C_IDX_FRAME_1].CtrlBuf.Size = 1;`<br>`buffer[I3C_IDX_FRAME_1].RxBuf.pBuffer = RxBuffer;`<br>`buffer[I3C_IDX_FRAME_1].RxBuf.Size = 1;`<br>DT72308V1 | • Prepare context buffers process<br>* uint32_t ControlBuffer[0xFF] : Buffer used by HAL to compute control data for the direct communication |
| `/* Add context buffer transfer CCC frame in Frame context */`<br>`if (HAL_I3C_AddDescToFrame(&hi3c1,`<br>`aCCCList,`<br>`NULL,`<br>`&buffer[I3C_IDX_FRAME_1],`<br>`1,`<br>`I3C_DIRECT_WITHOUT_DEFBYTE_RESTART) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br><br>`/* Start the transfer process */`<br>`if (HAL_I3C_Ctrl_ReceiveCCC_IT(&hi3c1, &buffer[I3C_IDX_FRAME_1]) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br>DT72382V1 | • Add CCC descriptor to the user data transfer descriptor.<br>• For some direct CCCs, a defining byte is optional; for others, it is always required.<br>• The controller transmits the direct read CCC command in interrupt mode. |

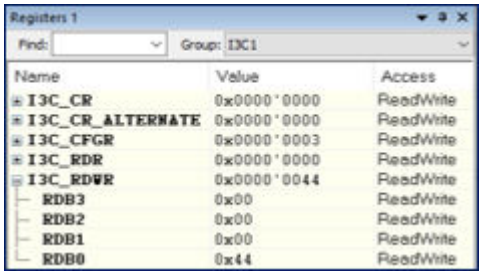
**IAR debug**

- Select I3C1 register
- Select I3C_RDWR register

In this example, the GETDCR (0x8F) code is used as a command code.

- GetDCR Code (0x8F) ⇨ 0x44

Figure 33. **I3C_RDWR register**




### 9.7.1.2 *Waveform results*

**First setup (SDA/SCL): (GETDCR)**

The following scope screenshots illustrate this first setup.

Figure 34. **SDR direct CCC GETDCR**

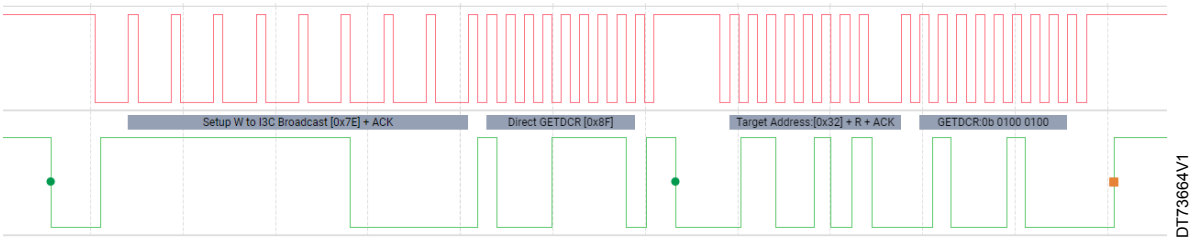
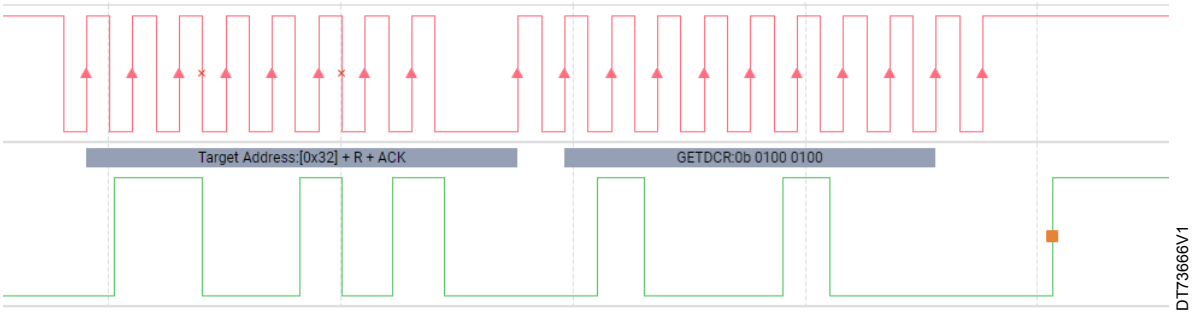



Figure 35. **Broadcast address + direct CCC for GETDCR**


Broadcast address (7'h7E) + W = 0 + ACK = 0 + direct CCC for GETDCR (0x8F) (T = 0) + Sr
Where:

- T: SDR controller written data as parity (as an odd parity); 0x8F: T-bit value is 0.

Figure 36. **Device dynamic address + MIPI I3C device characteristic register**




Device dynamic address 0x32 + R = 1 + ACK = 0 + MIPI I3C device characteristic register: 0x44 + T* = 0 + P

Where:

- T*: SDR target returned (read) data: as end of data; T* = 0 ⇨ end of data.

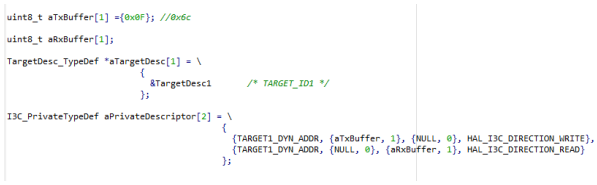
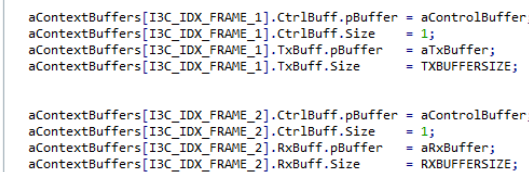
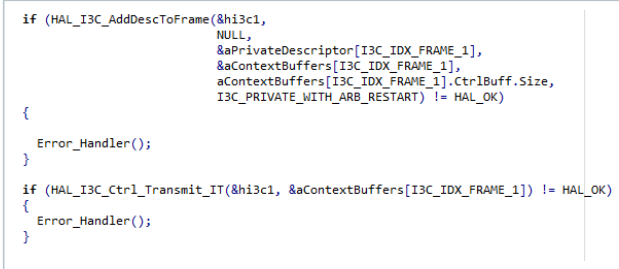
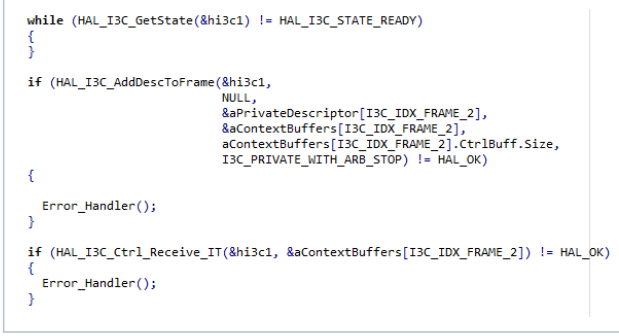
## 9.8 Private read

This example describes how the controller communicate and get data from the LSM6DSO registers (WHO_AM_I: address 0x0F) using I3C protocol in SDR mode based on STM32CubeMX.

It keeps the same configuration as mentioned but with different software parameters.

### 9.8.1 Software settings

**Table 15. Software settings with defining byte**

| | |
|---|---|
|  | • aTxBuffer: buffer used for transmission (value = reg address = 0x0F).<br>aRxBuffer: buffer used for reception.<br>aPrivateDescriptor: two directions (read and write). |
|  | • Prepare context buffers process |
|  | • Add context buffer transmit in Frame context<br>• Transmit private data process |
|  | • Check the current state of the peripheral<br>• Add context buffer receive in Frame context<br>• Receive private data process |

### 9.8.2 Waveform results

The main controller sends the register address using private write and after the target sends the data in push-pull mode. The following scope screenshots illustrate the first setup.

After a start the controller sends the target address with a write followed by the register address + T = 1.

Where:

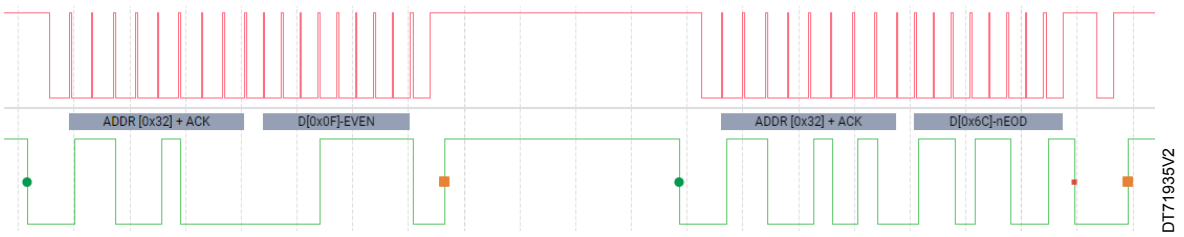
- T: SDR controller written data as parity: as an odd parity; 0x0F: T-bit value is 1.

After a repeated start, target address + R = 1, followed by the data (0X6C) + T* = 1 + STOP.

Where:

- T*: SDR target returned (read) data: as end of data; T* = 0 ⇨ end of data.

Figure 37. **I3C Private Read**




## 9.9 In-band interrupt use case

This section presents how to generate an interrupt between two devices without using an external pin in the I3C protocol.

### 9.9.1 In-band interrupt STM32CubeMX configuration

#### Controller settings

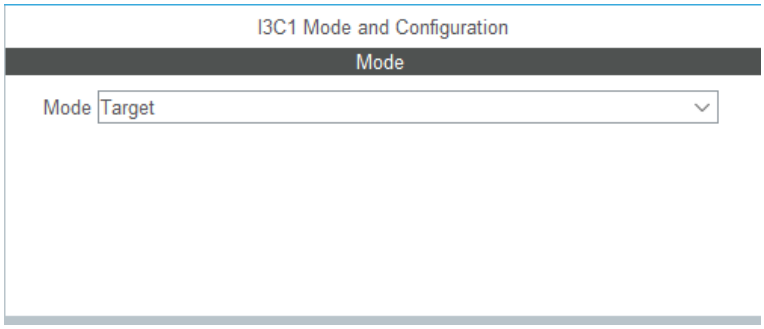
For the IBI configuration, keep the same previous configuration for the controller.

#### Target settings

Here is the configuration tab for the I3C protocol module in STM32CubeMX:

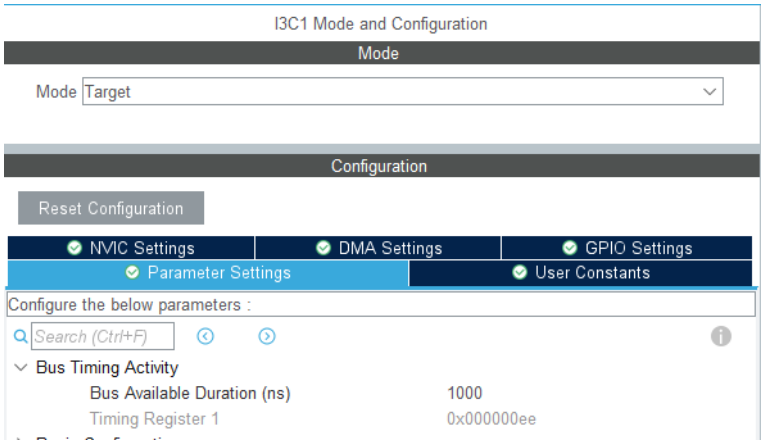
- Open STM32CubeMX and select "Access to MCU selector".
- Click on "Connectivity" and select "I3C1".
- Select "target" mode.

Figure 38. **Target mode for IBI**




#### Bus timing activity

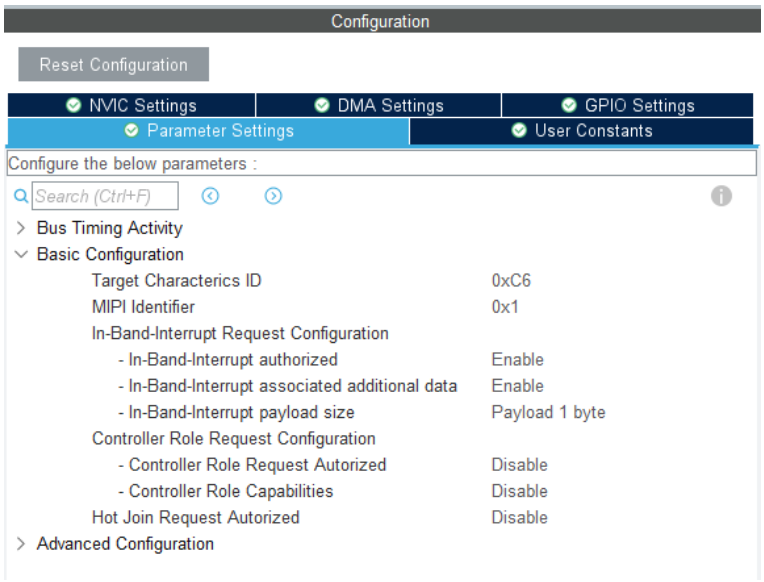
Figure 39. **Bus timing activity**



- Bus Register 1: 0x000000EE
  - **Timing_Register_1 = 0x000000EE** (when I3C acting as target):
    - Bits 7:0: AVAL [7:0] = 0xEE
    - Bus available condition time t_AVAL = 1 µs = 1000 µs

### Basic configuration

**Figure 40. Basic configuration**

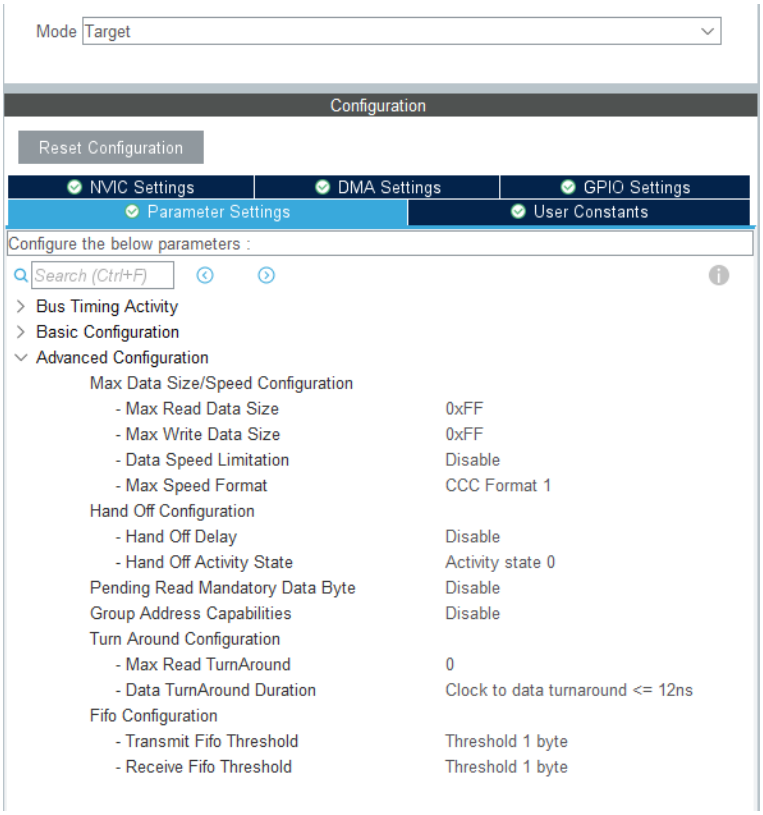



Where:

- Target characteristics ID: 0xC6
  - This value means that the device is a microcontroller. For more information, refer to the MIPI I3C device characteristics register.
- MIPI identifier = 0x1: identify every individual device
  - Bits [15:12] = MIPIID [3:0]: 4-bit MIPI instance ID
- In-band interrupt request configuration:
  - Enable the IBI authorized request.
  - Enable the data payload after an accepted IBI.
  - Payload data size: 1 byte (max 4 bytes).

#### Advanced configuration

**Figure 41. Advanced configuration**




Where:

- Max read data size: 0xFF
- Max write data size: 0xFF
  - This value must be between 0x0 and 0xFFFF
- Max speed format: CCC format 1
- FIFO configuration:
  - Transmit/Receive FIFO threshold: threshold 1 byte

#### Clock configuration

Keep the same configuration clock as the controller.

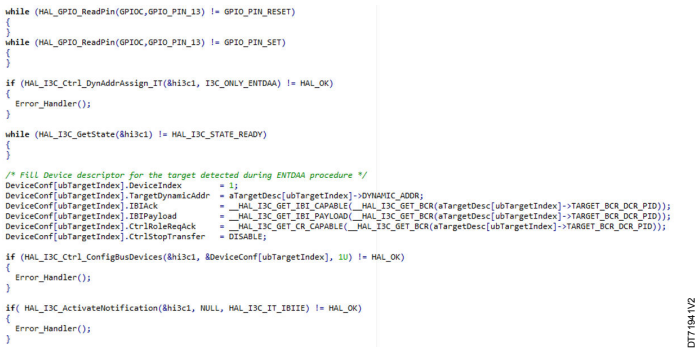
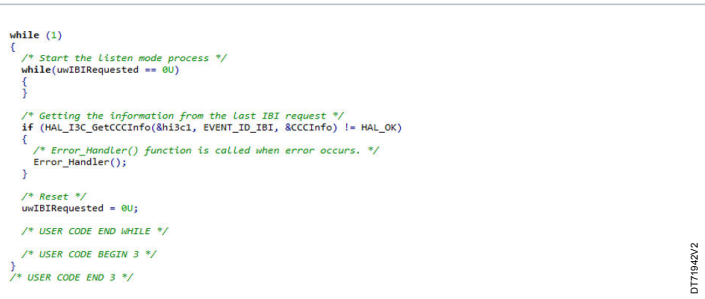
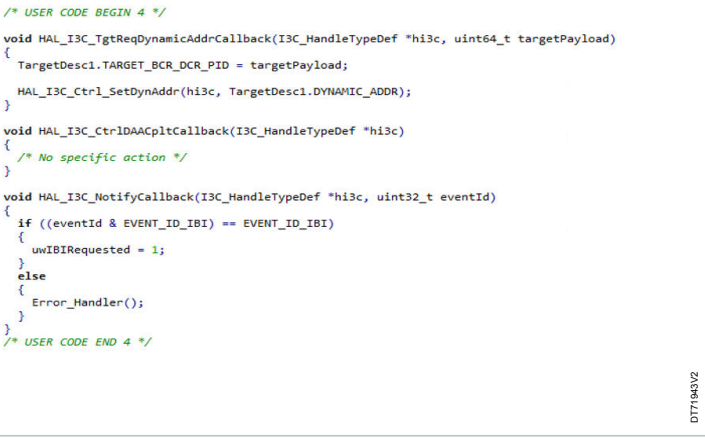
### 9.9.2 Software settings

#### *9.9.2.1 Controller*

In a first step, the controller initiates the sending of the ENTDAA CCC command. Then when ENTDAA is terminated, the controller stores the target capabilities through `HAL_I3C_Ctrl_ConfigBusDevices ()`.

Then, at reception of an in-band interrupt request from the target, the I3C controller retrieves the target dynamic address with associated data, if any, through `HAL_I3C_GetCCCInfo ()`.

**Table 16. Controller software settings for in-band interrupt**

| | |
|---|---|
|  | • Assign a dynamic address.<br>• Activate notifications for specially for this example.<br>• In-band interrupt requested by a target. |
|  | Start the listen mode process. |
|  | • I3C target requests a dynamic address callback.<br>• I3C notifies callback after receiving a notification. |

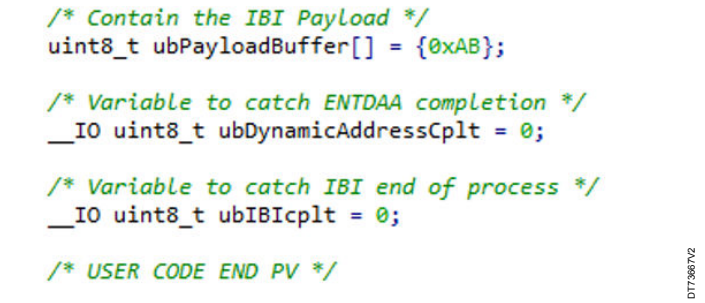
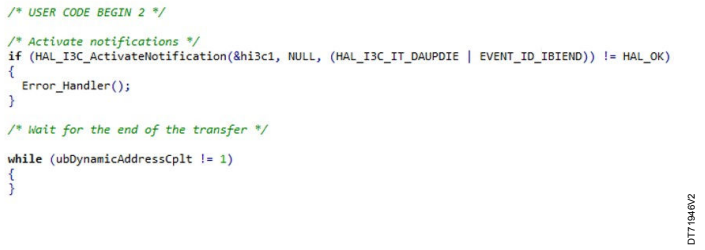
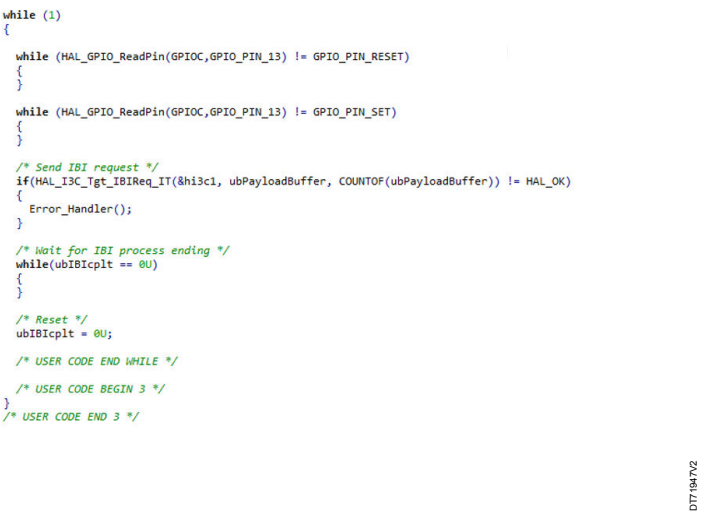
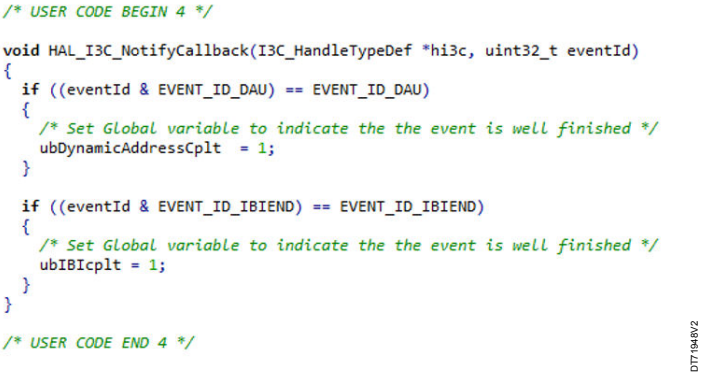
#### 9.9.2.2 *Target*

On the target side, upon receipt of the dynamic address assignment procedure from the controller, the target starts sending its payload.

Then, the target starts the communication by sending the in-band interrupt request through `HAL_I3C_Tgt_IBIReq_IT()` to the controller.

In fact, after this starting in-band-interrupt procedure, the I3C controller catches the event and requests a private communication with the target, which has sent and got acknowledgement of the in-band interrupt event.

**Table 17. Target software settings for in-band interrupt**

| | |
|---|---|
|  | Variable declaration:<br>PayloadBuffer[] = {0xAB} = 0b10101011 (in this case the user can set a free payload value) |
|  | Activate notifications specially for this example:<br>• Dynamic address association<br>• In-band interrupt ended process |
|  | Send IBI request |
|  | I3C notify callback after receiving a notification |

#### 9.9.2.3 *Software verification*

- Debug and open the I3C1
- Select I3C_IBIDR register

- Bits 7:0 **IBIDB0[7:0]** =0xab: payload data (earliest byte on I3C bus, mandatory data byte)
- Select I3C_DEVR1:
  - I3C_DEVR1.DIS=1, DA [6:0] write disabled
  - I3C_DEVR1.IBIDEN=1, IBI data enable
  - I3C_DEVR1.IBIACK=1, The controller acknowledges on the I3C bus the IBI request from the target

Also, the user can verify the payload value by looking at the content of CCCInfo:

- DA: 0x32
- Number of payload data: 1
- Payload byte: 0xab

**Figure 42. CCCInfo verification**

| Expression | Value |
|---|---|
| CCCInfo | <struct> |
| DynamicAddrValid | 0 |
| DynamicAddr | 0 |
| MaxWriteLength | 0 |
| MaxReadLength | 0 |
| ResetAction | 0 |
| ActivityState | 0 |
| HotJoinAllowed | 0 |
| InBandAllowed | 0 |
| CtrlRoleAllowed | 0 |
| IBICRTgtAddr | 0x32 |
| IBITgtNbPayload | 1 |
| IBITgtPayload | 0xab |

### 9.9.3 Waveform results

This waveform shows an example of IBI with a 0xAB mandatory byte.

**Figure 43. IBI waveform**


After the bus available condition, the controller emits a START and the target provides its dynamic address with a read-in-open-drain mode, the controller ACK-ed.

The mandatory data byte in push-pull mode + T = 0.

## 9.10 Mixed communication

This use case contains two parts:

- **Active I3C controller and I²C target on the bus**: read the content of a register (WHO_AM_I: 0x0F) from the I²C device (accelerometer LIS2DW12) on the I3C bus using interrupt mode.
- **Active I3C controller, I3C device, and I²C target on the bus**: assign a dynamic address to an I3C device and read from the I²C target.

### 9.10.1 Common communication

This configuration is used for the use case of a mixed bus (I3C and I²C on the bus).

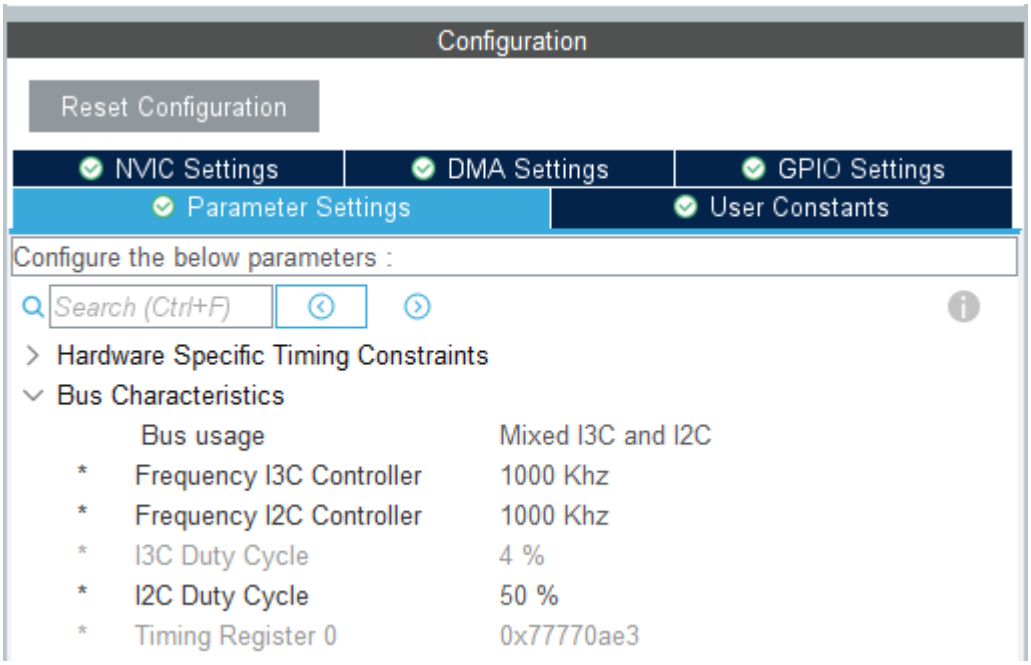
Here is the configuration based on STM32CubeMX:

- Open STM32CubeMX and select “Access to MCU selector".
- Click on “Connectivity” and select “I3C1”.
- Select “Controller” mode.

#### Bus characteristics

In this use case, the bus contains I3C and I²C devices.

**Figure 44. Bus characteristics**




- Bus usage: I3C and I²C devices connected on the bus: mixed I3C and I²C.
- Frequency: the communication runs at 1 MHz for the I3C device and for the I²C device (Fm+).
- Timing register 0: 0x77770AE3:
  - Bits 31:24 = 0x77: SCL high duration for legacy I²C messages: used for communication with device I2C. (I²C device (Fm+): tDIG_Hmin = 260 ns.)
  - Bits 23:16 = 0x77: SCL low duration in open drain phases.
  - Bits 15:8 = 0x0A: SCL high duration for I3C messages.
  - Bits 7:0 = 0xE3: SCL low duration in I3C push-pull phases.

#### Bus timing activity/basic configuration/advanced configuration

Keep the same configuration for these parameters.

#### Enabling interrupt

Enable the interrupt: I3C1 event interrupt

#### Clock configuration

Keep the same configuration for the clock configuration.

### 9.10.2 Active I3C controller and legacy I²C device on the I3C bus

#### *9.10.2.1 Software settings*

In this part, the active controller communicates and get data from register (0x0F) from the I2C target accelerometer LIS2DW12.

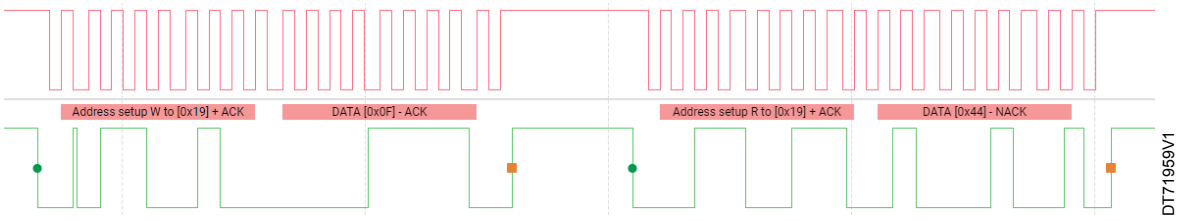
**Table 18. Software settings I3C/I2C**

| | |
|---|---|
| `I3C_XferTypeDef aContextBuffers[2];`<br>`uint32_t aControlBuffer[0xF];`<br>`uint8_t TxBuffer[1] ={0x0F};`<br>`uint8_t RxBuffer[1];`<br>`I3C_PrivateTypeDef aPrivateDescriptor[2] = \`<br>`{`<br>`{0x19, {TxBuffer, 1}, {NULL, 0}, HAL_I3C_DIRECTION_WRITE},`<br>`{0x19, {NULL, 0}, {RxBuffer, 1}, HAL_I3C_DIRECTION_READ}`<br>`};` | • aTxBuffer: buffer used for transmission (register WHO_AM_I: 0x0F)<br>• aRxBuffer: buffer used for reception.<br>• aPrivateDescriptor: 2 directions Read and Write. (0x19 static address)<br>• Prepare context buffers process |
| `aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.pBuffer = aControlBuffer;`<br>`aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size = 1;`<br>`aContextBuffers[I3C_IDX_FRAME_1].TxBuff.pBuffer = TxBuffer;`<br>`aContextBuffers[I3C_IDX_FRAME_1].TxBuff.Size = 1;`<br>`aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.pBuffer = aControlBuffer;`<br>`aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.Size = 1;`<br>`aContextBuffers[I3C_IDX_FRAME_2].RxBuff.pBuffer = RxBuffer;`<br>`aContextBuffers[I3C_IDX_FRAME_2].RxBuff.Size = 1;` | • Prepare context buffers process<br>• Add context buffer transmit in frame context |
| `if (HAL_I3C_AddDescToFrame(&hi3c1,`<br>`NULL,`<br>`&aPrivateDescriptor[I3C_IDX_FRAME_1],`<br>`&aContextBuffers[I3C_IDX_FRAME_1],`<br>`aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size,`<br>`I2C_PRIVATE_WITHOUT_ARB_RESTART)`<br>`!= HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br>`if (HAL_I3C_Ctrl_Transmit_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_1]) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br>`while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)`<br>`{`<br>`}`<br>`if (HAL_I3C_AddDescToFrame(&hi3c1,`<br>`NULL,`<br>`&aPrivateDescriptor[I3C_IDX_FRAME_2],`<br>`&aContextBuffers[I3C_IDX_FRAME_2],`<br>`aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.Size,`<br>`I2C_PRIVATE_WITHOUT_ARB_STOP ) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br>`if (HAL_I3C_Ctrl_Receive_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_2]) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}` | Frame context with:<br>• I2C_PRIVATE_WITHOUT_ARB_RESTART as direction<br>• Transmit data process<br>• Check the current state of the peripheral<br>• Add context buffer receive in Frame context with I2C_PRIVATE_WITHOUT_ARB_STOP as direction.<br>• Receive data process |

#### *9.10.2.2 Waveforms results*

The following scope screenshots illustrate the read of the data from the I²C target (LIS2DW12).
This communication is done at 1 MHz (Fast Mode Plus) based on the I3C source clock, which runs at 250 MHz.

Figure 45. **Legacy I²C read transfer on I3C bus**




- 0x0F: register address "WHO_AM_I".
- ACK: I3C target is ready on the I3C bus from the target.
- 0x44: register value.
- NACK: end of data from the controller.

### 9.10.3 Active I3C controller with legacy I²C and I3C device on the I3C bus

#### *9.10.3.1 Software settings*

In this example, the active controller assigns the dynamic address to the I3C device connected on the I3C bus and after that the controller communicates and get data from register (0x0F) from the legacy I2C (accelerometer LIS2DW12).

Table 19. **Software settings mixed bus: I3C - I3C and I²C**

| | |
|---|---|
| `/* USER CODE BEGIN PV */`<br>`I3C_XferTypeDef aContextBuffers[2];`<br>`uint32_t aControlBuffer[0xF];`<br>`uint8_t TxBuffer[1] ={0x0F};`<br>`uint8_t RxBuffer[1];`<br><br>`#define DA_Target 0x32`<br>`TargetDesc_TypeDef TargetDesc1 =`<br>`{`<br>`"TARGET_ID1",`<br>`TARGET_ID1,`<br>`0x0000000000000000,`<br>`0x00,`<br>`DA_Target,`<br>`};`<br><br>`TargetDesc_TypeDef *aTargetDesc[2] = \`<br>`{`<br>`&TargetDesc1, /* TARGET_ID1 */`<br>`};`<br>`I3C_PrivateTypeDef aPrivateDescriptor[2] = \`<br>`{`<br>`{0x19, {TxBuffer, 1}, {NULL, 0}, HAL_I3C_DIRECTION_WRITE},`<br>`{0x19, {NULL, 0}, {RxBuffer, 1}, HAL_I3C_DIRECTION_READ}`<br>`};`<br>`/* USER CODE END PV */` | • TxBuffer: buffer used for transmission (value: reg address = 0x0F) (WHO_AM_I)<br>• RxBuffer: buffer used for reception.<br>• aTargetDesc: target descriptor.<br>– (0x32 as DA)<br>• aPrivateDescriptor: two directions, read and write. (0x19 static address) |
| `/* USER CODE BEGIN 2 */`<br>`while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_RESET)`<br>`{`<br>`}`<br>`while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_SET)`<br>`{`<br>`}`<br><br>`if (HAL_I3C_Ctrl_DynAddrAssign_IT(&hi3c1, I3C_RSTDAA_THEN_ENTDAA) != HAL_OK)`<br>`{`<br>`Error_Handler();`<br>`}`<br><br>`while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)`<br>`{`<br>`}` | • Assigning DA in interrupt mode (initiate a RSTDAA follow by a ENTDAA)<br>• check the current state of the peripheral.<br>• Prepare context buffers process |

```
aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.pBuffer = aControlBuffer;
aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size    = 1;
aContextBuffers[I3C_IDX_FRAME_1].TxBuff.pBuffer   = TxBuffer;
aContextBuffers[I3C_IDX_FRAME_1].TxBuff.Size      = 1;

aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.pBuffer = aControlBuffer;
aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.Size    = 1;
aContextBuffers[I3C_IDX_FRAME_2].RxBuff.pBuffer   = RxBuffer;
aContextBuffers[I3C_IDX_FRAME_2].RxBuff.Size      = 1;

if (HAL_I3C_AddDescToFrame(&hi3c1,
                           NULL,
                           &aPrivateDescriptor[I3C_IDX_FRAME_1],
                           &aContextBuffers[I3C_IDX_FRAME_1],
                           aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size,
                           I2C_PRIVATE_WITHOUT_ARB_RESTART)
                           != HAL_OK)
{

  Error_Handler();
}


if (HAL_I3C_Ctrl_Transmit_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_1]) != HAL_OK)
{
  Error_Handler();
}
 while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)
{
}
if (HAL_I3C_AddDescToFrame(&hi3c1,
                           NULL,
                           &aPrivateDescriptor[I3C_IDX_FRAME_2],
                           &aContextBuffers[I3C_IDX_FRAME_2],
                           aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.Size,
                           I2C_PRIVATE_WITHOUT_ARB_STOP ) != HAL_OK)
{

  Error_Handler();
}
if (HAL_I3C_Ctrl_Receive_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_2]) != HAL_OK)
{
  Error_Handler();
}
/* USER CODE END 2 */
```

- Add context buffer transmit in Frame context with I2C_PRIVATE_WITHOUT_ARB_RESTART as direction
- Transmit data process
- check the current state of the peripheral.
- Add context buffer receive in Frame context with I2C_PRIVATE_WITHOUT_ARB_STOP as direction.
- Receive data process

### 9.10.3.2 *Waveforms results*

**Figure 46. DAA for I3C device and legacy I²C read transfer on I3C bus**

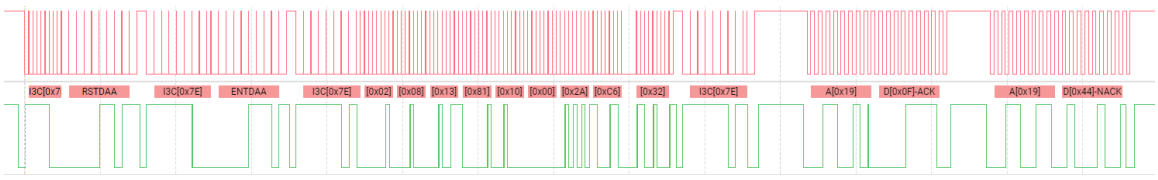



**Figure 47. Dynamic address assignment for I3C device**

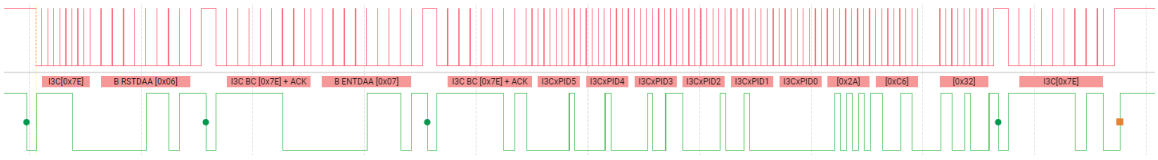
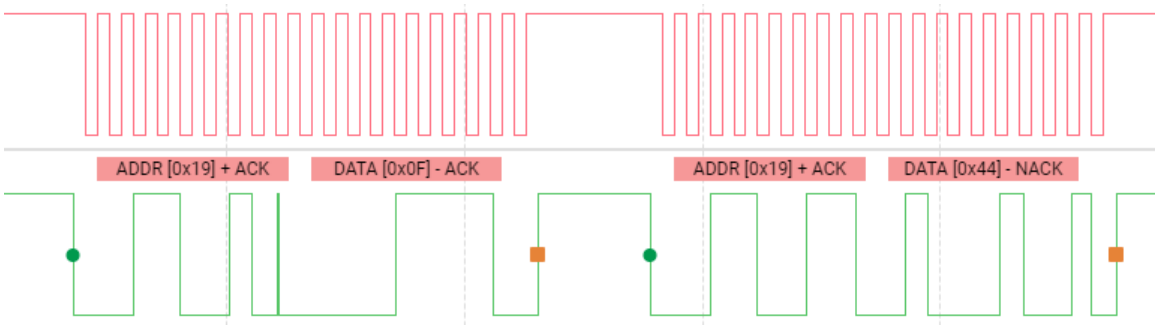



**Figure 48. Legacy I²C read transfer**



# Revision history

Table 20. **Document revision history**

| Date | Revision | Changes |
|---|---|---|
| 13-Mar-2023 | 1 | Initial release |
| 18-Mar-2024 | 2 | Updated:<br>• Document title<br>• Section Introduction<br>• Section 1: General information<br>• Section 4: STM32H5 implementation<br>• Table 3. I3C peripheral versus MIPI v1.1<br>• Section 4.2: I3C peripheral integration (Integration with RCC)<br>• Section 5: I3C hardware requirements<br>• Section 6: Bus timing<br>• Section 8.10: In-band interrupt (IBI)<br>• Section 9: Examples of I3C communications with an STM32H503RB Nucleo board<br>• Table 9. Firmware available examples<br>• Section 9.3: Hardware and software settings<br>• Section 9.5.3: Communication at 12.5 MHz push pull<br>• Section 9.6.1: Software settings<br>• Section 9.6.2: Waveform results<br>• Section 9.7.1: Direct read without defining byte<br>• Section 9.7.1.1: Software settings<br>• Section 9.7.1.2: Waveform results<br>• Section 9.8.2: Waveform results<br>• Section 9.9.2: Software settings<br>• Section 9.9.3: Waveform results<br>• Section 9.10.2.2: Waveforms results<br>• Section 9.10.3.2: Waveforms results<br>General document cleanup |
| 04-Jun-2024 | 3 | Updated:<br>• Figure 1. I3C communication interface<br>• Table 3. I3C peripheral versus MIPI v1.1<br>• Figure 5. Example of I3C mode GPDMA architecture<br>• Figure 6. STM32H503xx communication interfaces operating with VDDIO2 at 1.2 V (USART/LPUART, SPI, and $I^2C$/I3C)<br>• Figure 14. Legacy $I^2C$ transfer on I3C bus |
| 12-Feb-2025 | 4 | Added STM32U3 series<br>Added STM32N6 series<br>Updated:<br>• Table 1. Applicable products<br>• Block diagram<br>• Section 6: Bus timing |

# Contents

# List of tables

# List of figures