AN5879

Application note

Introduction to I3C for STM32H5 series MCU

Introduction

The number of connected sensors to a main device acting as controller has increased since many years. Managing multiple sensors is becoming more complex with heterogeneous serial interfaces like the still widespread low cost I2C, a more bandwidth capable SPI or a traditional UART, and extra signals (GPIOs) for interrupt and/or power modes.

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 today defined by the MIPI I3C specification, version 1.1.1. The I3C supports the new features and maintains some backward compatibility with the I2C.

The new series of STM32 MCUs integrate the I3C peripheral, supporting the set of the 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 the STM32CubeMX, to covers most of the I3C communication modes, available on the STM32 microcontrollers, and to provide recommendations for the correct use of the I3C peripheral.

It starts with an I3C bus overview, then a description of the I3C features, and finally 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, LIS2DW12) in private or direct mode, and the management of in-band interrupt from targets.

1 General information

This document applies to the STM32H5 series microcontrollers that are Arm® Cortex® core-based devices.

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

1.1 Reference documents

- STM32H503 datasheet (DS14093)
- STM32H562/563 datasheet (DS14258)
- STM32H573 datasheet (DS14121)
- STM32H503 erratasheet (ES0561)
- STM32H503 series Arm®-based 32-bit MCUs reference manual (RM0492)
- STM32H563/H573 and STM32H562 Arm®-based 32-bit MCUs reference manual (RM0481)

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 the I3C in SDR mode.

2.1 Operations

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: assigns a dynamic address unlike I2C, which has a static address
- Private read/write transfers
- Legacy I2C messages: the I3C controller can communicate with I2C devices on the bus
- In Band interrupt IBI: the target device, which is connected on 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 the initialization
- Controller role request

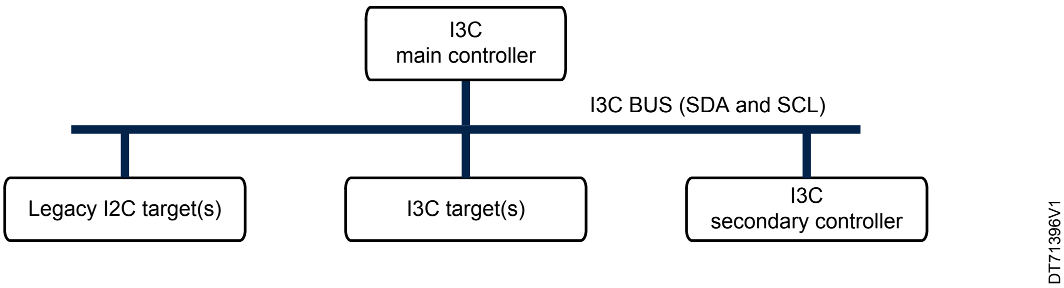
2.2 I3C bus

Bus configuration

I3C is a two-wire interface similar as the I2C 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 there is connected an I2C target device without clock stretching). There are four main types of devices on the bus:

- I3C main controller
- I3C secondary controller
- I3C targets
- Legacy I2C targets

Figure 1. I3C communication interface



Bus communication

SCL can run at up to 12.5 MHz in push-pull mode and up to 4MHz 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 term of performances.

- Mixed fast bus:(I2C and I3C devices on the bus)

Case I2C 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 on the bus.

2.3 Bus format

All transfers on the I3C bus begin with a START (S) followed with 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 to generate 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	

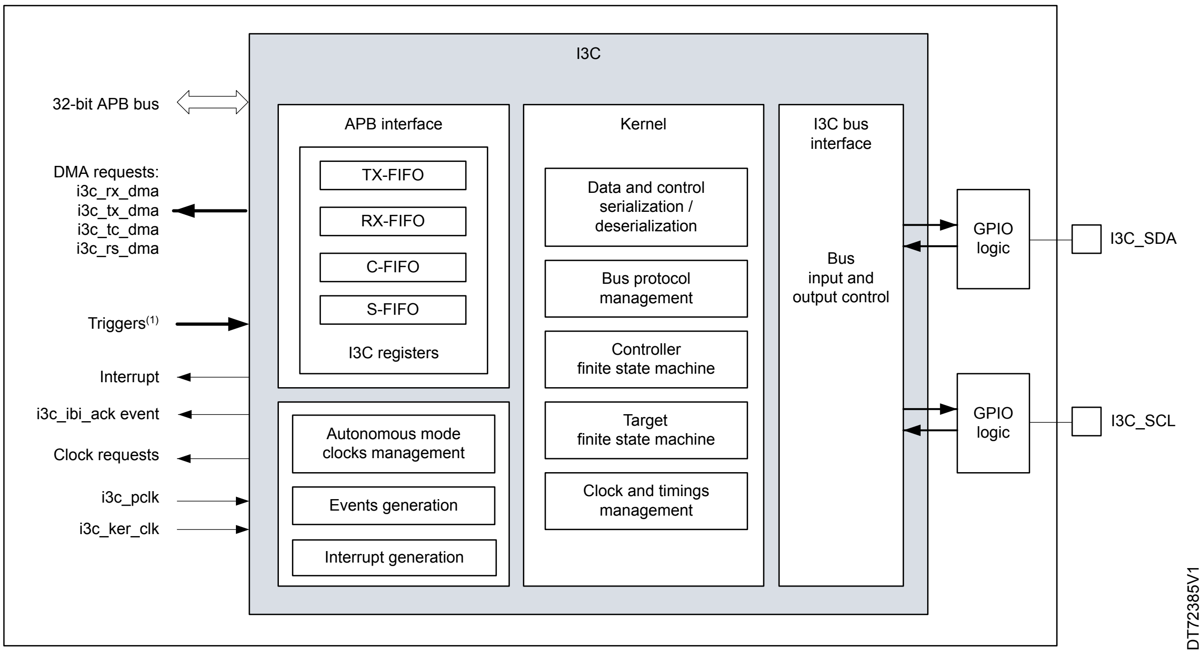
T-bit:

- 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



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 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 also because SDA can be in open-drain mode with an integrated controller pull-up.

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

Table 1. I3C versus I2C capabilities

Parameters	I3C	I2C
Protocol	Serial, half-duplex	Serial, half-duplex
Frequency	Up to 12.5MHz (SDR)	Up to 1Mhz
Signaling	Open-drain 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 10-bit static address
Dynamic addressing	Supported	Not supported
In-band interrupt	Supported	Not supported
Hot-join	Supported	Not supported
Multicontrollers	Supported	Supported

4 STM32 implementation

The I3C SDR-only peripheral supports all the features of the MIPI I3C specification v1.1 as I3C primary controller, secondary controller, and target. It can control all I3C bus-specific sequencing, protocol, arbitration, and timing, and can be acting as controller or as target.

4.1 I3C MIPI support

The I3C peripheral supports the MIPI specification v1.1 as show in:

Table 2. I3C peripheral versus MIPI v1.1

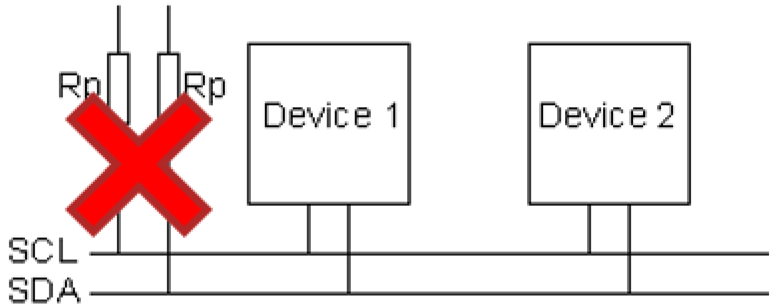
Feature	MIPI I3C v1.1	I3C peripheral		Comments
		Controller	Target	
I3C SDR message	X	X	X	-
Legacy I²C message	X	X	-	Mandatory as controller when there is an I²C target. Optional in MIPI v1.1 when target.
HDR message	X	-	-	Optional in MIPI v1.1
Dynamic address	X	X	X	-
Static address	X	X	-	I3C peripheral cannot be a target/slave on an I²C 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	-	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
Multilane	X	-	-	Optional in MIPI v1.1
Monitoring device early termination	X	-	-	Optional in MIPI v1.1

4.2 I3C peripheral integration

Integration with bus

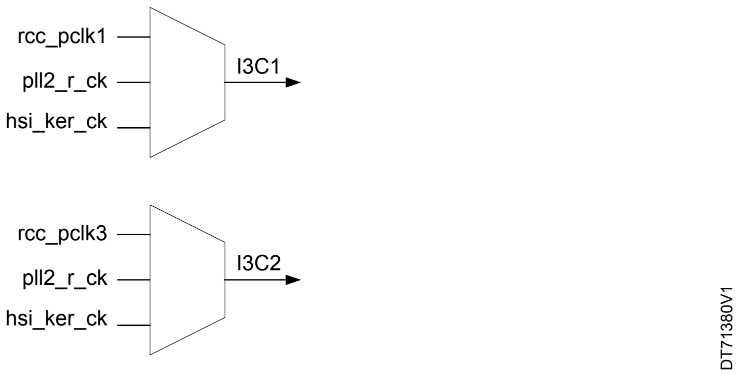
Pull-up is on during open-drain phase, and is off during push-pull phase. It is handled automatically by the hardware.

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



Integration with RCC

Figure 4. I3Cx multiplexer



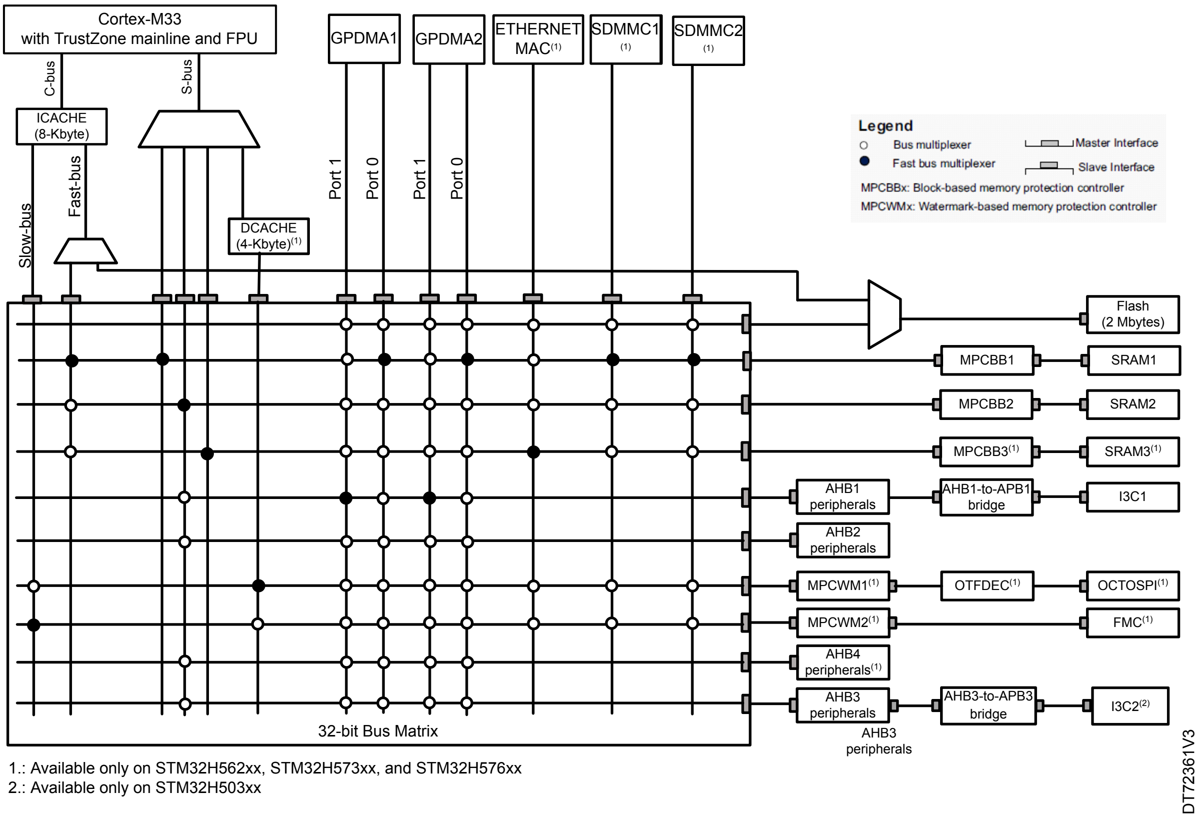
From the I3Cx mux, we can select the clock source:

- rcc_pclk1/3
- pll2_r_ck
- hsi_ker_ck

GPDMA

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

Figure 5. I3C mode GPDMA architecture



1.: Available only on STM32H562xx, STM32H573xx, and STM32H576xx
2.: Available only on STM32H503xx

5 I3C hardware requirements

The I3C interface provides major efficiencies in bus power while providing greater significant speed improvements over I2C.

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

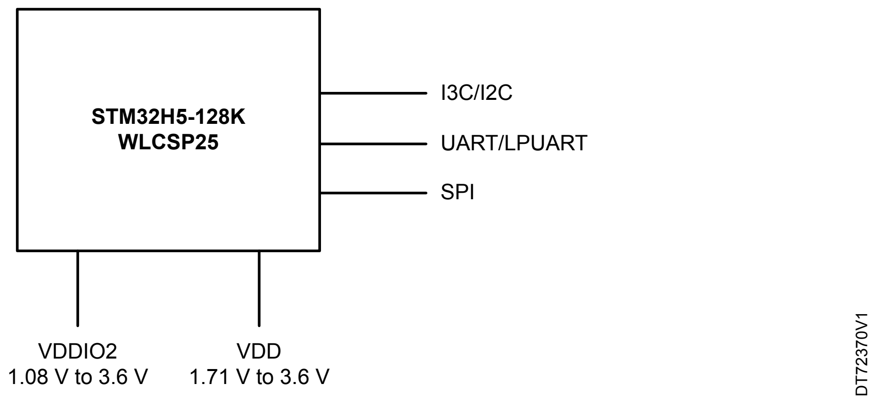
Table 3. **Electrical characteristics**

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 I2C, USART/LPUART, and SPI are mapped on GPIOs that have a dedicated supply pin VDDIO2 down to 1.08V. 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 I2C/I3C)**



6 Bus timing

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

- The device is acting as a controller: the user can adjust timings by using I3C timing register 0, 1, and 2.
- The device is acting 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.

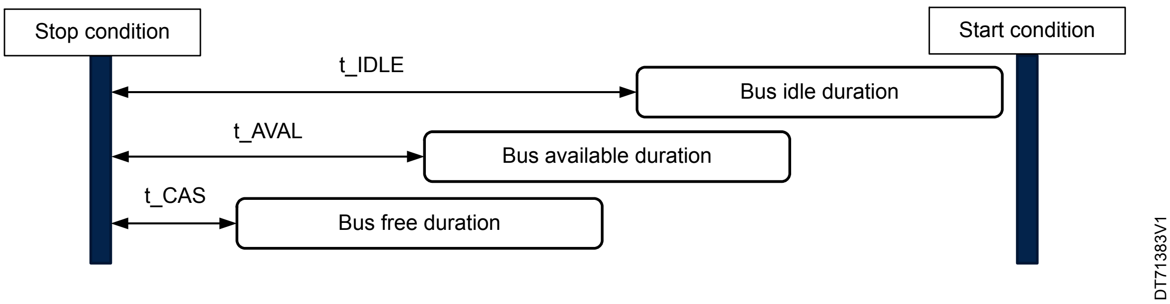
Frequency of the I3CCLK kernel clock > 2x frequency of the SCL clock

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 below:

1. Bus free duration:
- For pure bus: only I3C devices are connected on the bus
- For mixed bus: at least one legacy I2C device is present on the I3C bus
1. Bus available duration: A target may only issue a START request after a t_AVAL
2. Bus idle duration: the SDA and SCL lines sustain High level for a duration of at least t_IDLE

Figure 7. Bus timing



7 tSCO timing

tSCO (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 tSCO 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 tSCO > 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 tsco > 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 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 I2C 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.

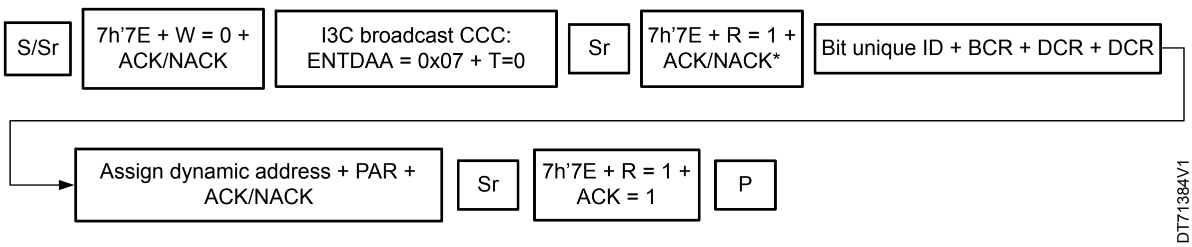
The command code for the ENTDAA broadcast CCC is 0x07. Support for this CCC is required, unless this I3C device has an I2C static address.

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 4. **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 5. **Target address available for use**

Target dynamic address		Restriction	Description
Binary	Hex		
0000000	7’h00	Shall not use	I3C reserved
0001000 - 0111101	7’h08 – 7’h3D	Available for use	54 addresses
0111111 -1011101	7’h3F – 7’h5D	Available for use	31 addresses
101 1111-110 1101	7’h5F – 7’h6D	Available for use	15 addresses
110 1111-111 0101	7’h6F – 7’h75	Available for use	7 addresses
111 0111	7’h77	Available for use	1 address
111 1111	7’h7F	Shall not use	I3C reserved: Broadcast address single bit error detect

Table 6. **Conditional target address**

Target dynamic address		Restriction	Description
Binary	Hex		
0000011-0000011	7’h03-7’h04	Conditional	Available for use only if no legacy I2C devices supporting I2C “High-speed mode” are present on the bus
1111000-1111001	7’h78-7’h79	Conditional	Available for use only if no legacy I2C devices are present on the bus that both 1. support I2C “extended address mode” 2. either have an extended address, or would be impacted by the extended address mechanism
111 1011	7’h7B	Conditional	Available for use only if no legacy I2C devices are present on the bus that both: 1. support I2C “extended address mode” 2. either have an extended address, or would be impacted by the extended address mechanism
111 1101	7’h7D	Conditional	Available for use only if no legacy I2C devices supporting I2C “device ID mode” are on the bus.

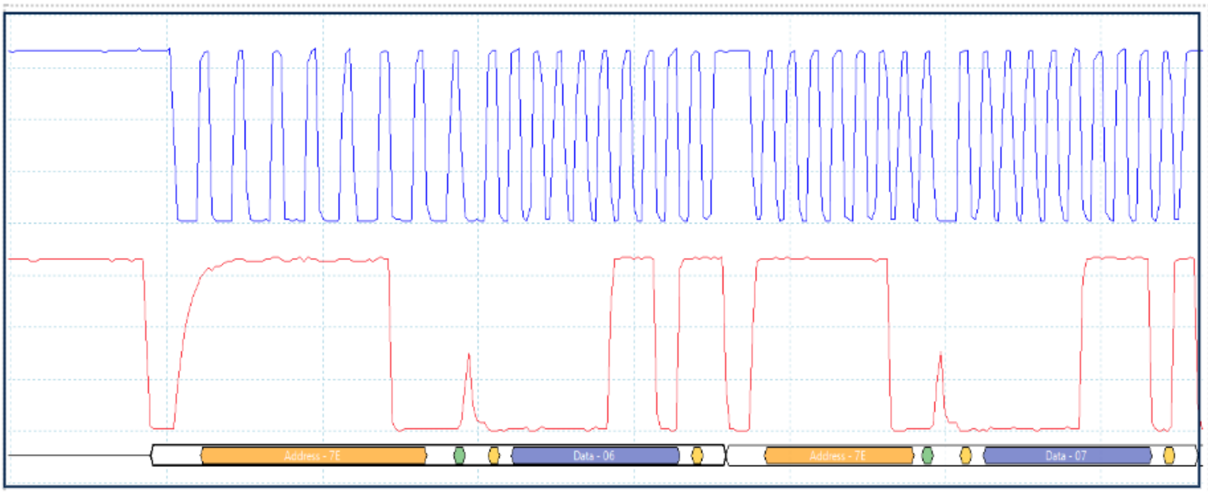
8.5 Changing dynamic address

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 is acting as target so check thus flags:

1. *I3C_EVR.DAUPDF: dynamic address update flag*
2. *I3C_DEVR0.DAVAL: is cleared on the acknowledge 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. (example RSTDAA, ENTDAA, DISEC, and so on)
- Direct read/write: is seen by a specific target by selecting its dynamic address. (example of direct read GETPID, GETBCR)

8.7 Frame format

All command codes share the same frame format for the broadcast/direct CCCs as show in:

- 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

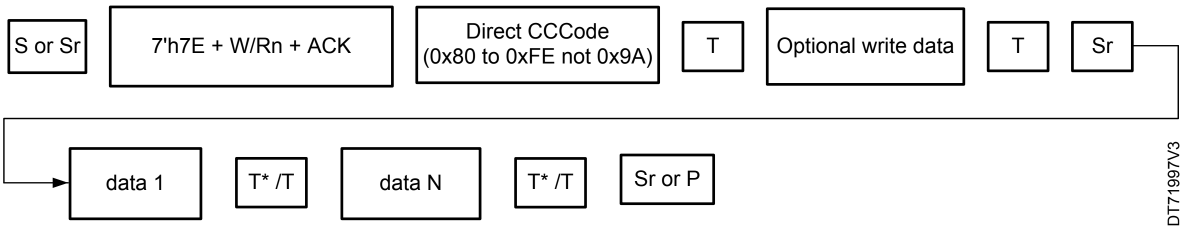
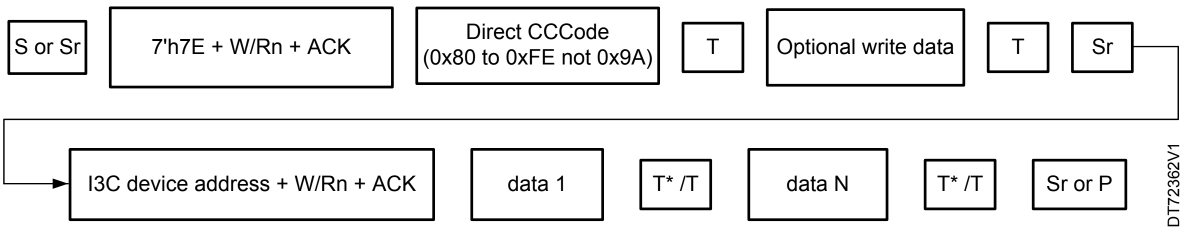


Figure 11. I3C direct CCC transfer



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 7. List of Supported CCCs supported by LSM6DSO

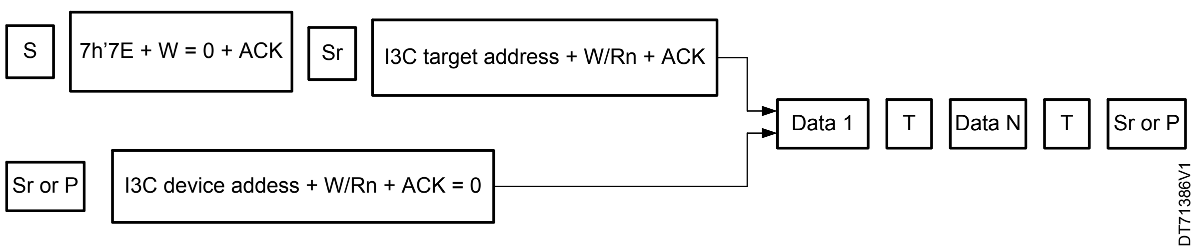
Command Name	Command Code	CCC Type
ENTDAA	0x07	Broadcast
SETDASA	0x87	Direct
ENEC	0x00	Broadcast
	0x80	Direct
DISEC	0x01	Broadcast
	0x81	Direct
ENTAS 0...3	0x82 to 0x85	Direct
	0x02 to0x05	Broadcast
SETXTIME	0x28	Broadcast
	0x98	Direct
GETXTIME	0x99	Direct
RSTDAA	0x06	Broadcast
	0x86	Direct
SETMWL	0x08	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.

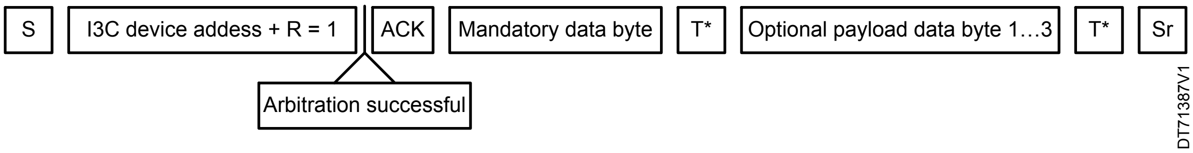
If two devices are sending their own address (it may be the reserved address 7'h7E + RnW=0 from the controller), the first device, which is not actively pulling SDA low and instead is issuing passively a '1' bit (high-Z) starting from the most significant bit until the low significant bit of the address, 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:(ref 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 I2C on the I3C bus

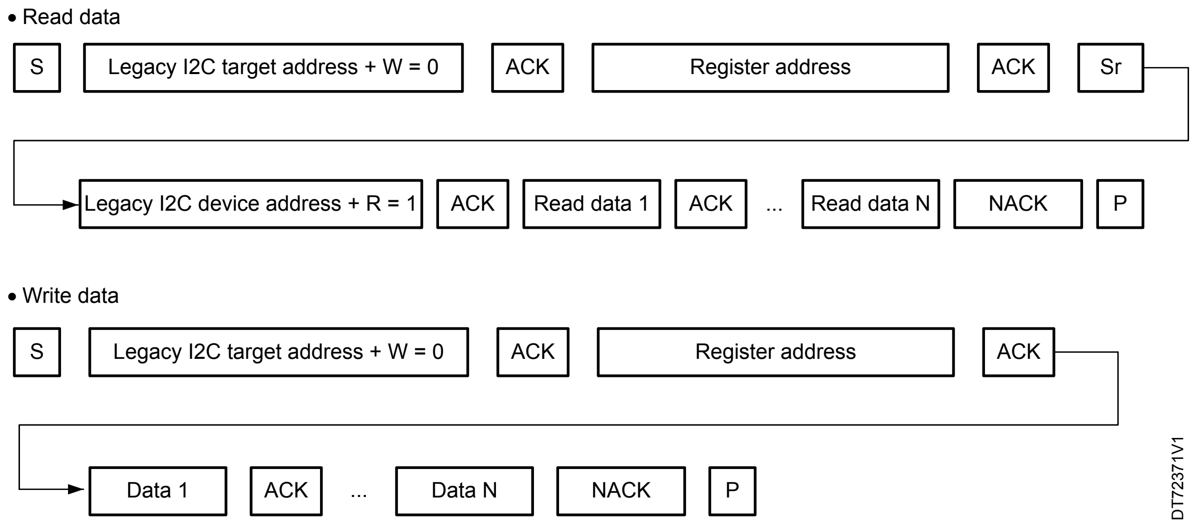
I3C protocol supports existing I2C target devices and messages on the I3C bus with some limitations:

- If a spike filter is implemented on the I2C 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 I²C target device, the I3C bus allows the slowest frequency of the legacy I2C, and can eliminate certain I3C bus features.
- The extended address with 10 bits is not used on the I3C bus.
- The legacy I2C 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

This section aims at demonstrating how to use STM32CubeMX to create an I3C serial communication application for a NUCLEO-H503R8 and X-NUCLEO-IKS01A3 boards, and provides an easy guide for examples 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 8. Firmware available examples

Firmware package	Example name	Board
STM32CubeH5	I3C_controller_ENTDA_IT	NUCLEO-H503RB
	I3C_Target_ENTDAA_IT	
	I3C_Controller_Private_Command_IT	
	I3C_Target_Private_Command_IT	
	I3C_Controller_InBandInterrupt_IT	
	I3C_Target_InBandInterrupt_IT	
	I3C_Target_WakeUpFromStop	
	I3C_Controller_I2C_Com	
	I3C_Target_I2C_Com	

9.2 I3C examples based on STM32CubeMX

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

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

9.3 Hardware and software settings

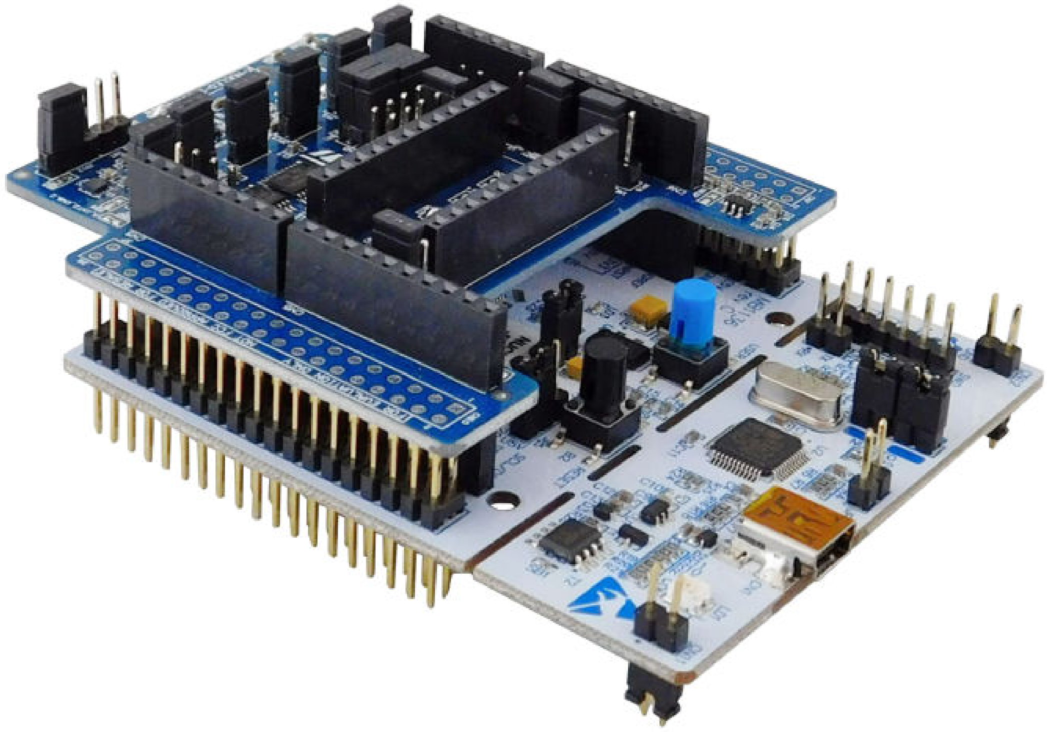
The table below lists the development environment details.

Table 9. Software and hardware environment

Use Case	Hardware	Software
Dynamic addressing	STM32H503 board (controller I3C) Accelerometer and gyroscope LSM6DSO (target I3C) • Keep the JP2	STM32CubeMX STM32Cube_FW_H5_V1.0.0 IAR Systems
Direct transfer		
Private transfer		
IBI	STM32H503 board (controller I3C) STM32H503 board (target I3C)	
Mixed Bus	STM32H503 board (controller I3C) STM32H503 board (target I3C) Accelerometer LIS2DW12 (Legacy I2C) • Keep the JP1	

The X-NUCLEO-IKS01A3 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**



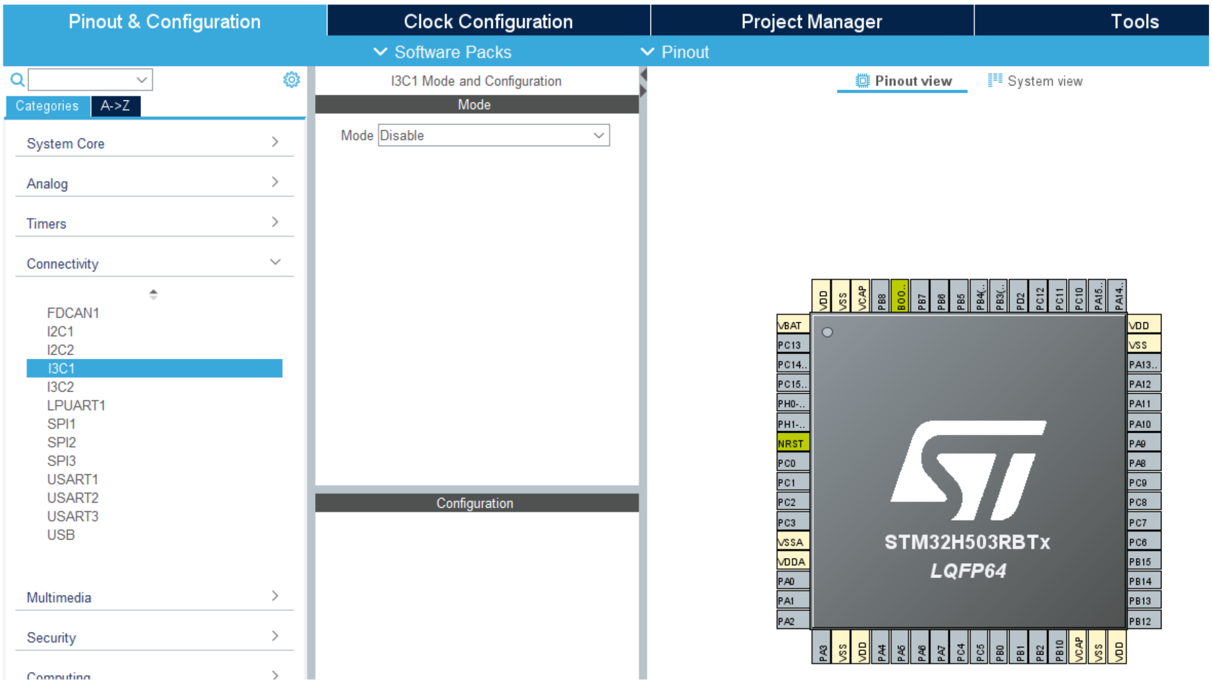
9.4 Common configuration

9.4.1 STM32CubeMX - I3C GPIOs configuration

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

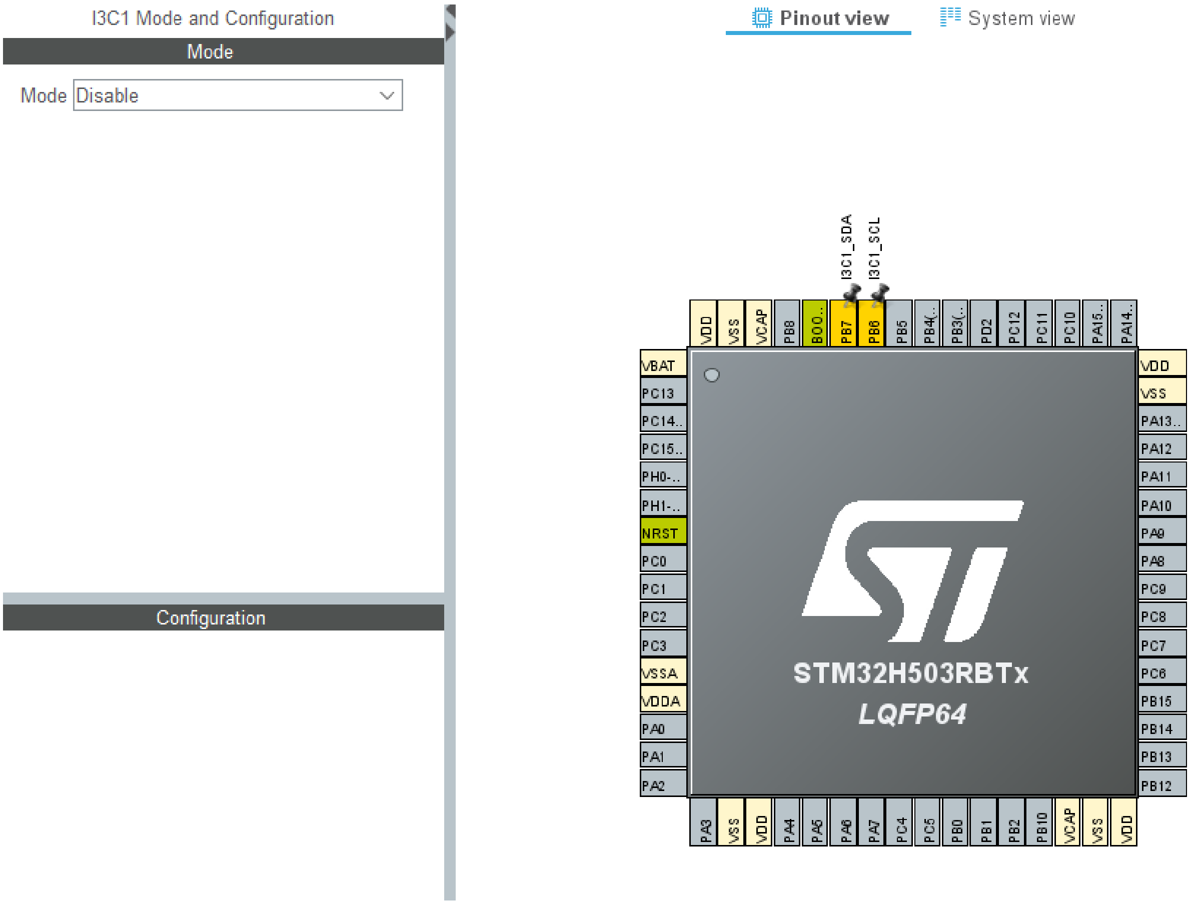
- Creating a new STM32CubeMX project and selecting the corresponding MCU
- Click on “Connectivity > I3C1”

Figure 16. I3C1 selection



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

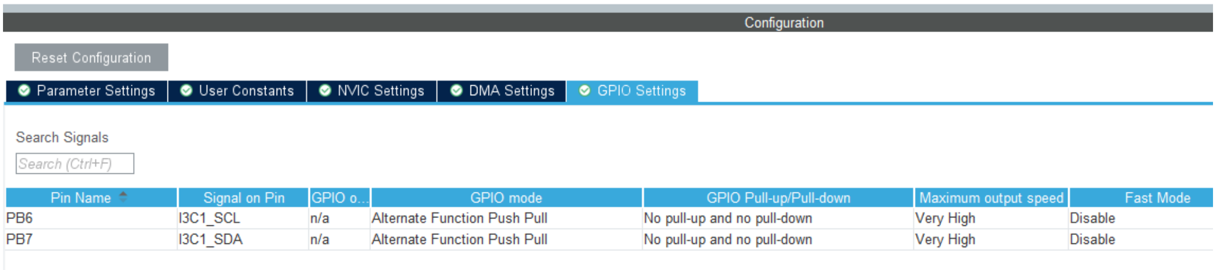
Figure 17. I3C1 pins



- GPIOs settings:

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

Figure 18. I3C1 GPIOs configuration

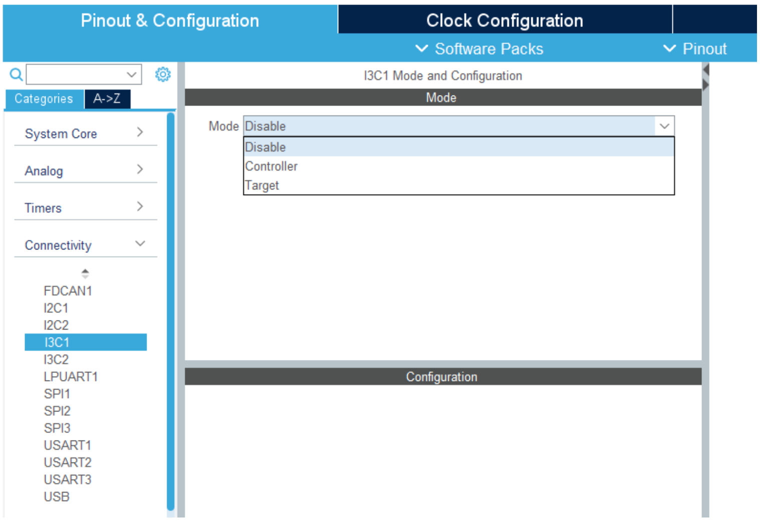


Pin Name	Signal on Pin	GPIO o...	GPIO mode	GPIO Pull-up/Pull-down	Maximum output speed	Fast Mode
PB6	I3C1_SCL	n/a	Alternate Function Push Pull	No pull-up and no pull-down	Very High	Disable
PB7	I3C1_SDA	n/a	Alternate Function Push Pull	No pull-up and no pull-down	Very High	Disable

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

Select controller or target mode

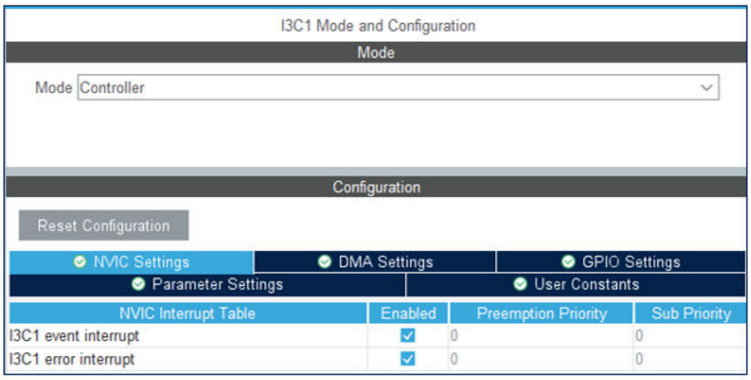
Figure 19. **I3C mode selection**



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 choice frequency, duty cycle, and bus usage.

9.5.1 Bus characteristics

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

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

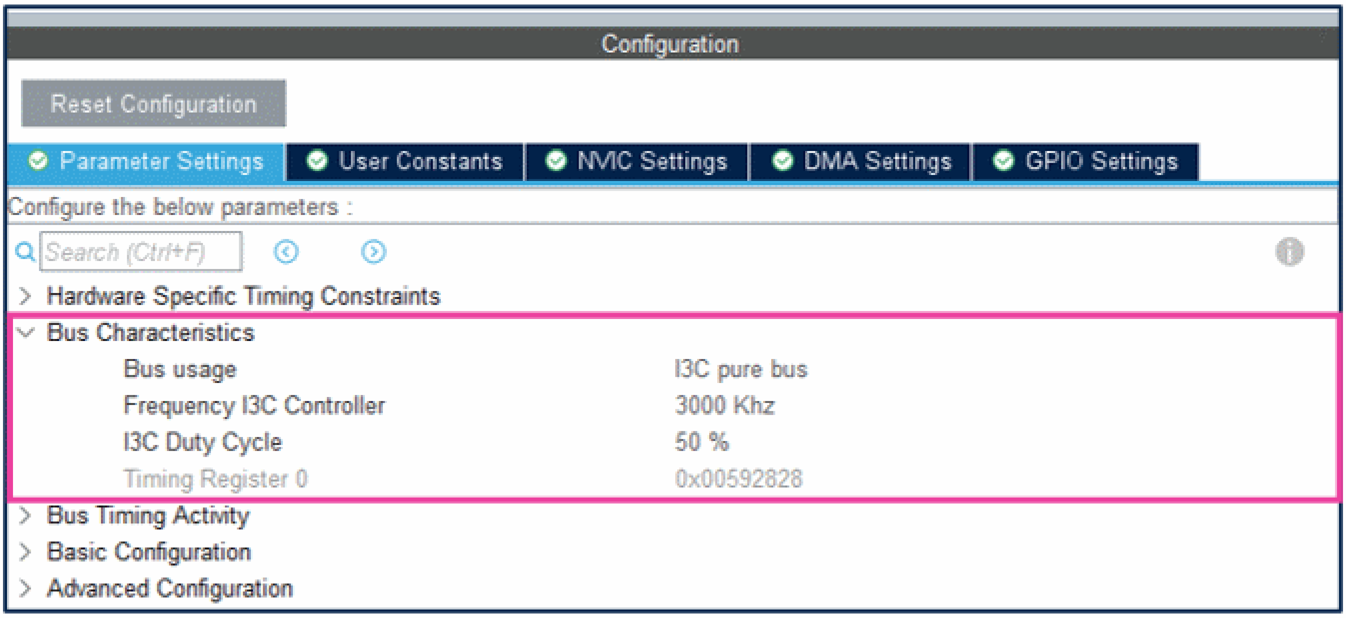
9.5.2 Communication at 3 MHz push pull

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

These values are calculated automatically by CubeMX:

- Bits 31:24 = 0x00: SCL high duration for legacy I2C messages: NO I2C 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**



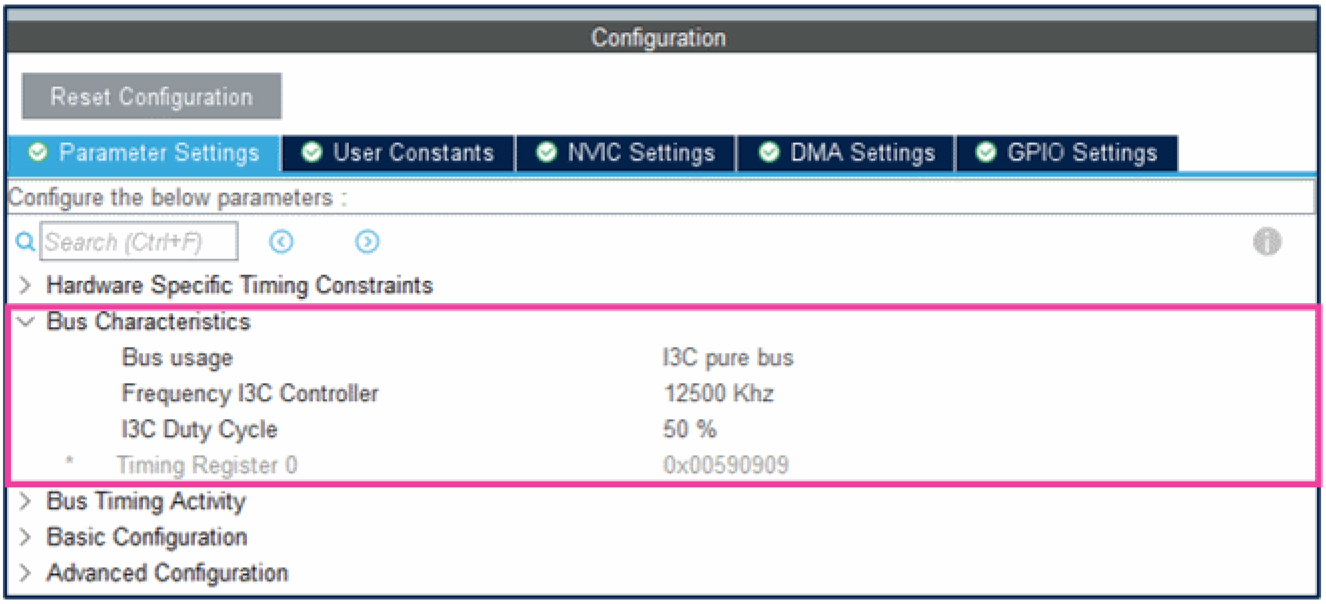
Note: The definition of duty cycle depends on user hardware constraints, and it impact open drain phases and ensure a good of quality of SDA signal. More we increase the frequency, the more we reduce the duty cycle.

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 12.5MHz
- Timing register 0: 0x00550909
  – Bits 31:24 = 0x00: SCL high duration for legacy I2C messages: NO i2c 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 = 0x09: SCL high duration for I3C messages.
    t_{SCLH_I3C} = (SCLH_I3C+1)xt_{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)xt_{I3CCLK} = (09 + 1)*(1/250) = 40 ns

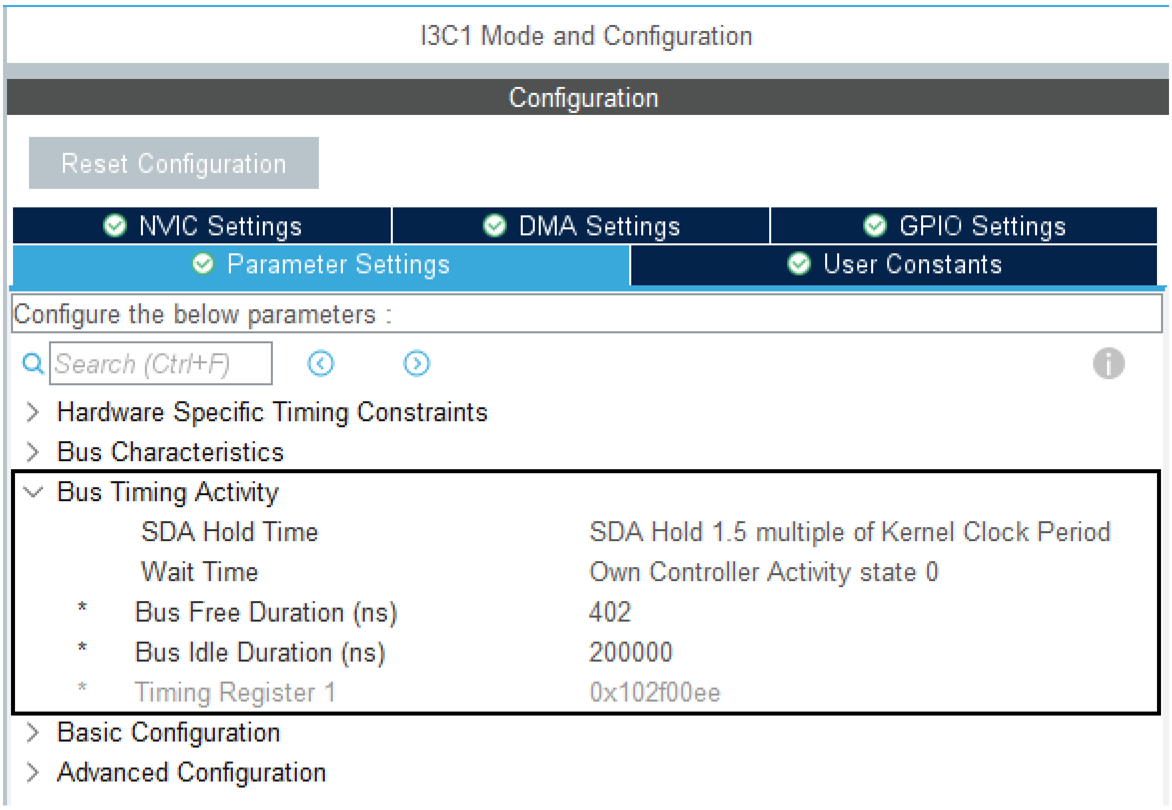
Figure 22. Bus characteristics for 12.5 MHz frequency



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.

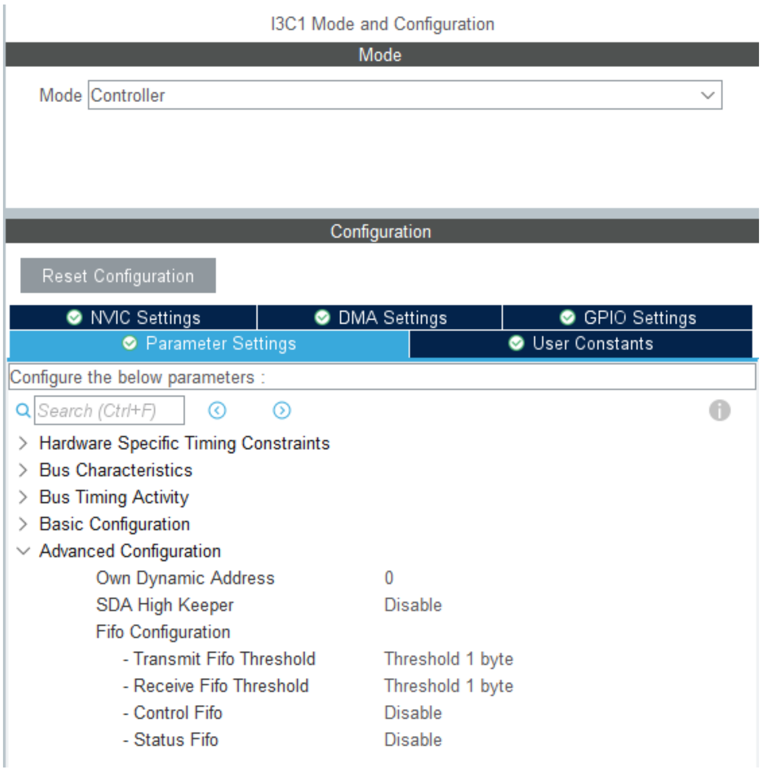
- Bits 27:23: reset value “0”
- Bits 22:16= FREE [6:0] = 0x02f
- t_{CAS} = [(FREE [6:0] +1) x2 - (0,5 + SDA_HD)] x tI3CCLK
- Bus Free Duration =
  t_{CAS} = [([6:0] +1) x 2 - (0,5 + SDA_HD)] x t_i3cclk
  t_{CAS} = [(47+1)x2 - 1.5] x (1/250 Mhz)
  t_{CAS} = 378 ns
- Bits 15:0: reset value “0”
- Bits 9:8: “00” no new controller
- Bits 7 :0=0xee: t_AVAL= (AVAL [7 :0] +2) xtI3CCLK and t_IDLE= t_AVAL*200:
  – Bus available duration = 1us
  – Bus idle duration= 200us
  – The value of bus free duration set on this example is 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

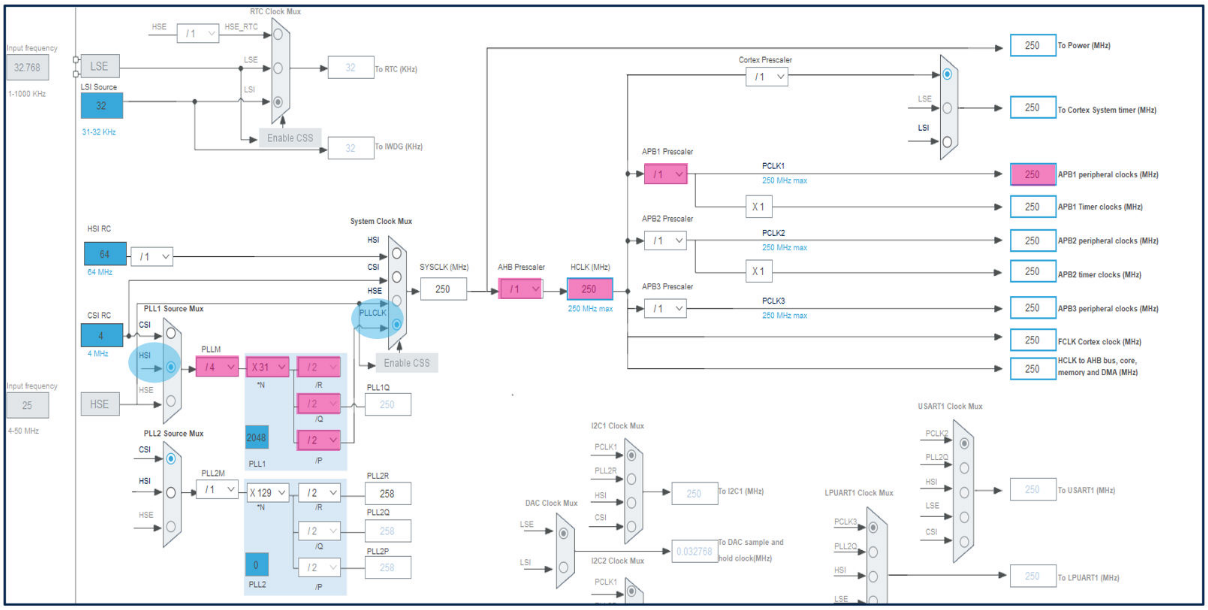
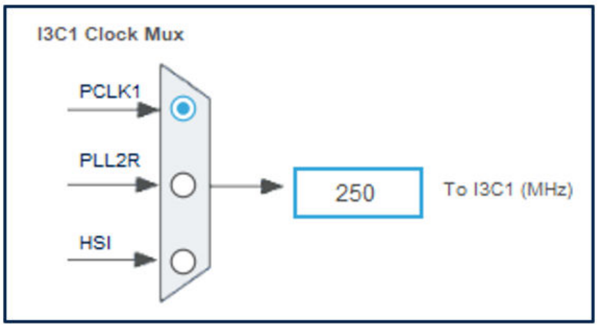


Figure 26. I3C1 clock multiplexer

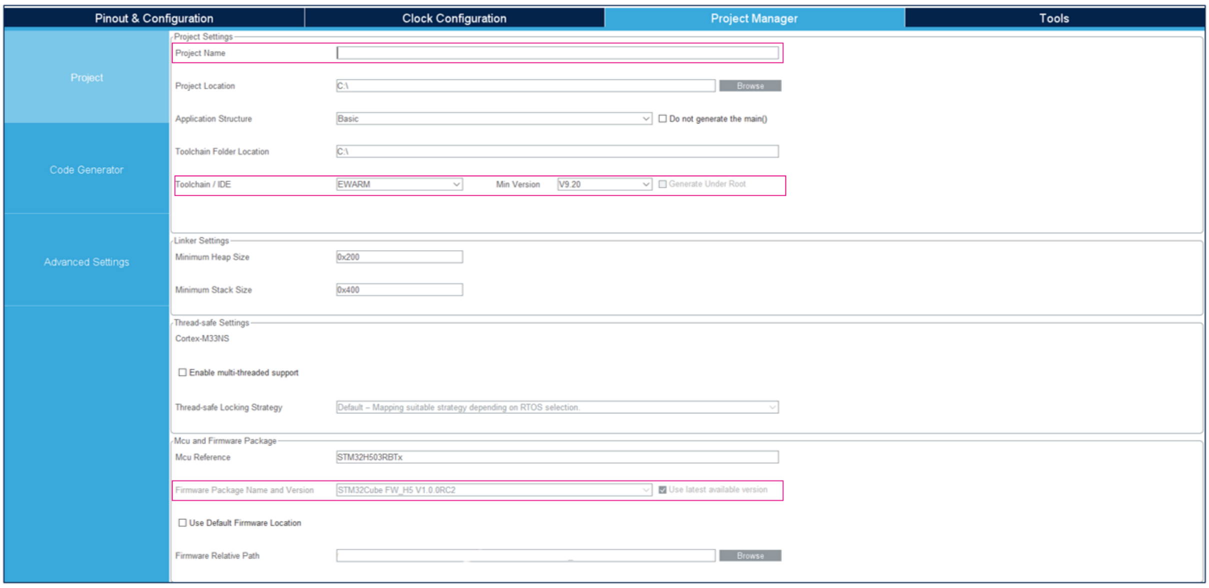


9.5.7 Project generation

This part follows the previous steps to generate at the end 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 the following functions.

Table 10. Software settings for dynamic addressing

#define TARGET1_DYN_ADDR 0x32
 	
/********************/
/* Target Descriptor */
/********************/
TargetDesc_TypeDef TargetDesc1 =
{
  "TARGET_ID1",
  TARGET_ID1,
  0x0000000000000000,
  0x00,
  TARGET1_DYN_ADDR,
};
 	
#endif /* __STM32_I3C_DESC_TARGET1_H */
	• target descriptor + dynamic address to assign

/* USER CODE BEGIN PV */
 	
/* Array contain targets descriptor */
TargetDesc_TypeDef *aTargetDesc[1] = \
  {
    &TargetDesc1,   /* TARGET_ID1 */
  };
 	
/* USER CODE END PV */
	• The array contains the target descriptor

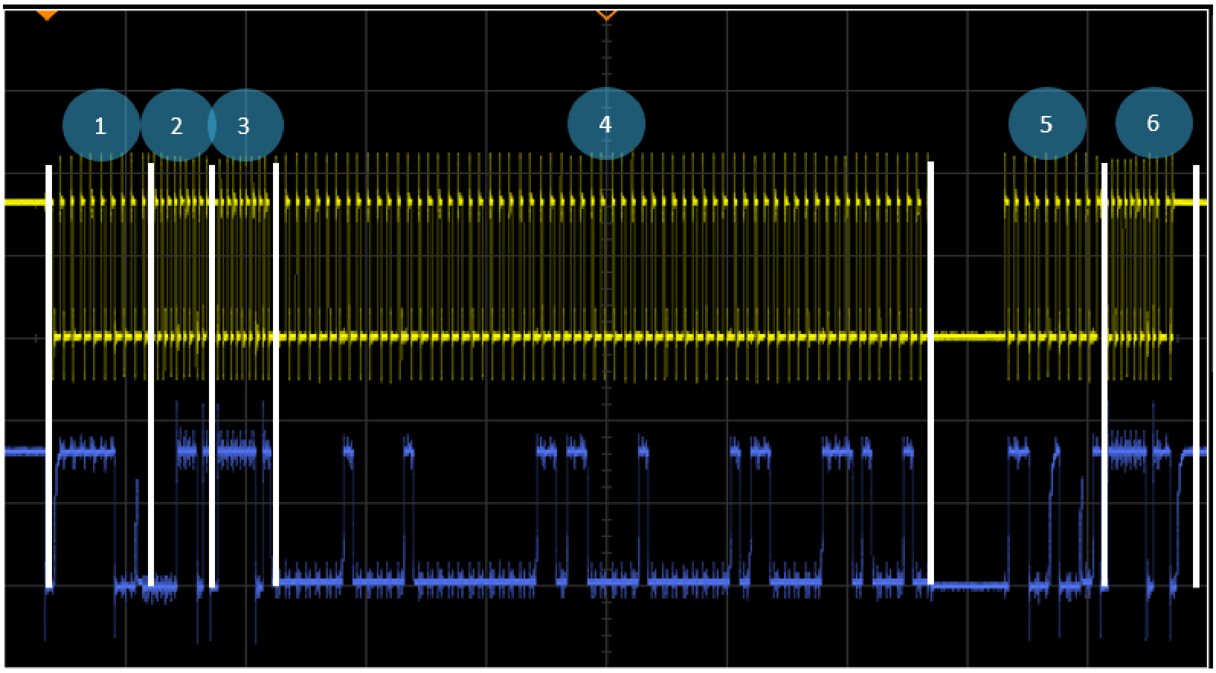
/* Assign dynamic address processus */
if (HAL_I3C_Ctrl_DynAddrAssign_IT(&hi3c1, I3C_ONLY_ENTDAA) != HAL_OK)
{
  /* Error_Handler() function is called when error occurs. */
  Error_Handler();
}
	Assigning a dynamic address in the interrupt mode (initiate an ENTDAA without RSTDAA)

/* USER CODE BEGIN 4 */
void HAL_I3C_TgtReqDynamicAddrCallback(I3C_HandleTypeDef *hi3c, uint64_t targetPayload)
{
 	
  TargetDesc1.TARGET_BCR_DCR_PID = targetPayload;
 	
  HAL_I3C_Ctrl_SetDynAddr(hi3c, TargetDesc1.DYNAMIC_ADDR);
 	
}
 	
void HAL_I3C_CtrlDAACpltCallback(I3C_HandleTypeDef *hi3c)
{
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
}
 	
/* USER CODE END 4 */
	• GET target characteristics
	• 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. DAA waveform



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** +

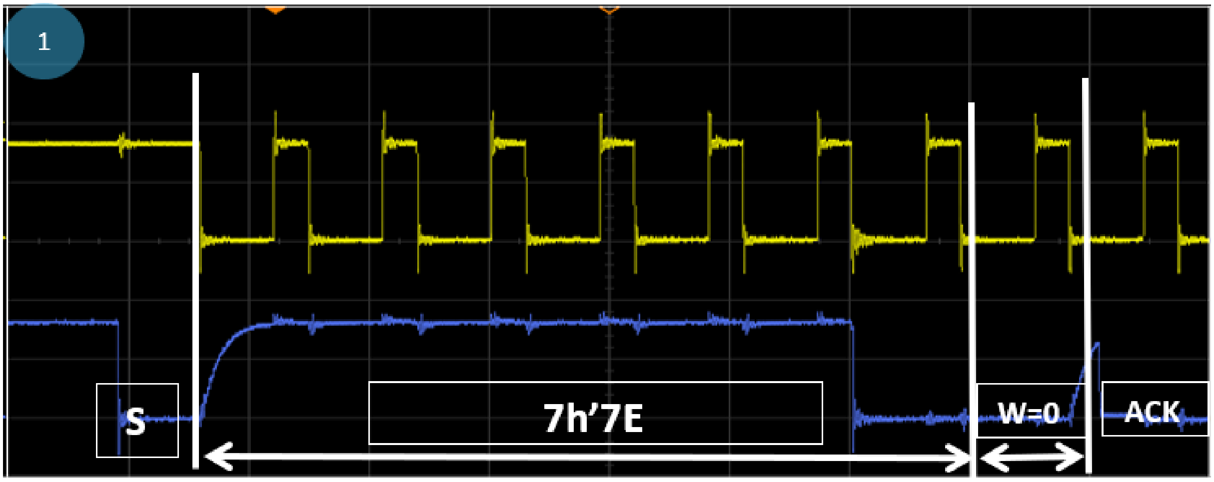
(7h'7E + R=1 +ACK=1) + stop **PP**

Table 11. Communication frequency

Mode	Open Drain	Push-Pull
Frequency	1.904 MHz	3.040 MHz

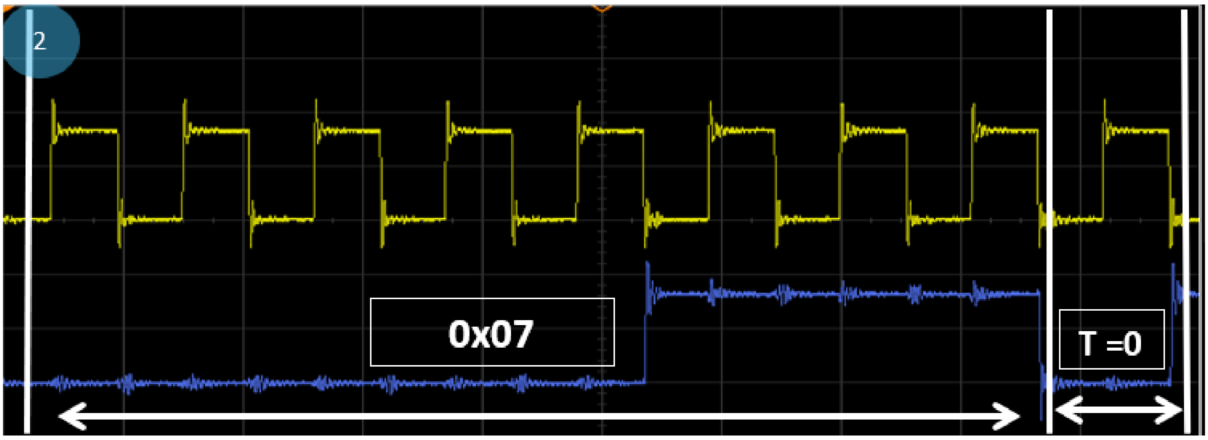
1. Address header: **START + (7’h7E +W=0+ACK=0) (OD phase)**

Figure 29. Address header



2. I3C broadcast CCC (ENTDAA=0x07): **ENTDAA ((0x07) + T=0)**

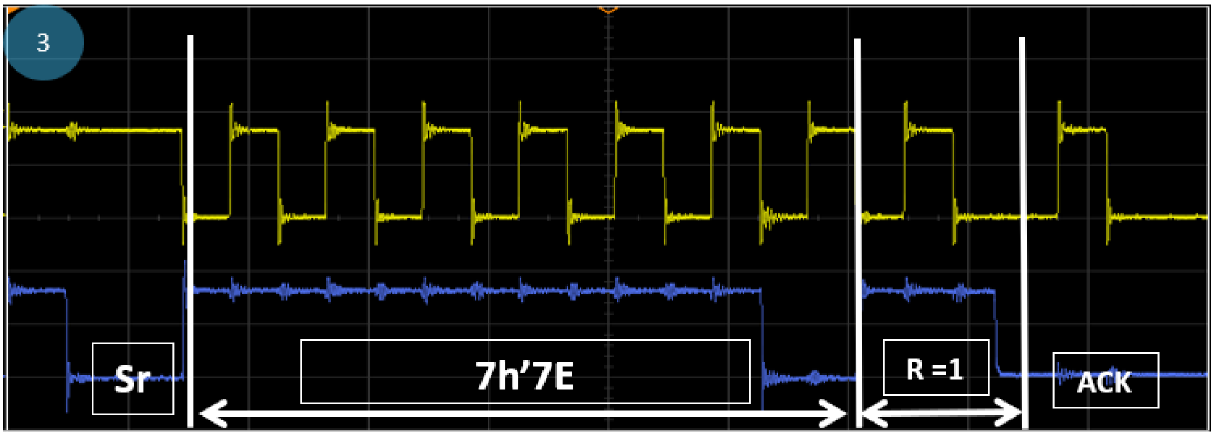
Figure 30. ENTDAA CODE



Note: The 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).

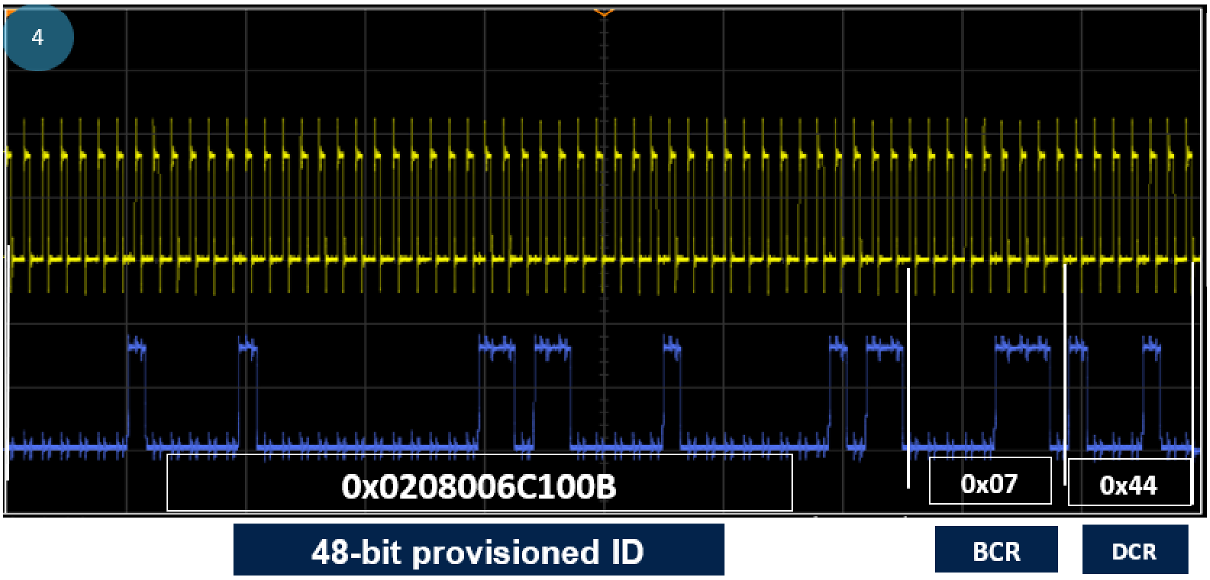
3. Repeated START + (7’h7E + R=1 +ACK): **(PP phase)**

Figure 31. **7’h7E with a read**



4. Target characteristics: **(OD phase)**

Figure 32. **Target characteristics**



- **BCR = 0x07 = 0000 0111**
  - ◦ • Bits [7:6] = 00: I3C target.
    - Bit 5 = 0: no support of optional advanced capabilities.
    - Bit 4 = 0: Is 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 12. **48-bit provisioned ID**

Value	Signification
Bits [47 :33] = 000000100000100	MIPI manufacturer ID
Bits: 32 =0	Provisioned ID type selector: 1’b0: 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

5. Dynamic address: **Dynamic address (0x32 +PAR=0 +ACK=0): (OD phase)**

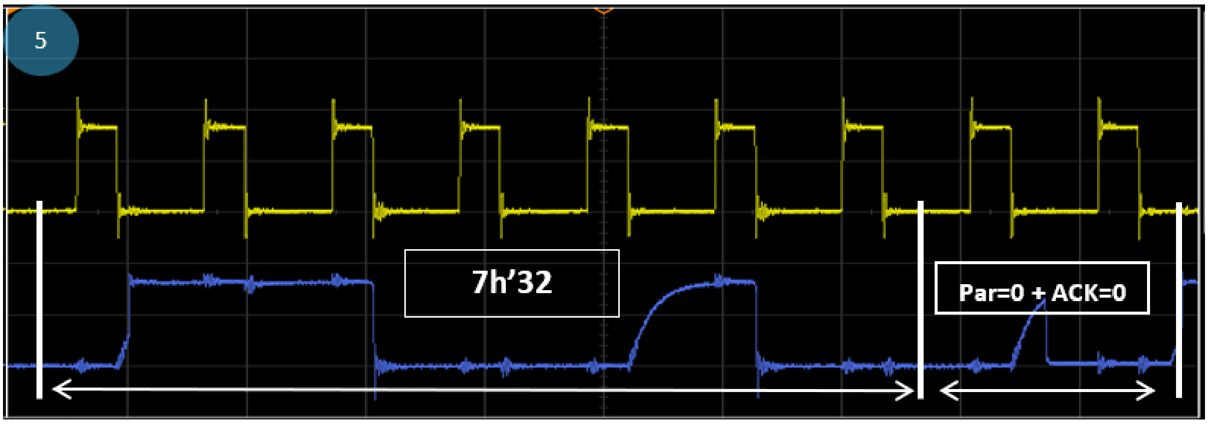


Figure 33. **Dynamic address**

6. I3C broadcast address + R=1 + stop:**(0x7E + R=1 +ACK=1) +stop: (OD phase)**

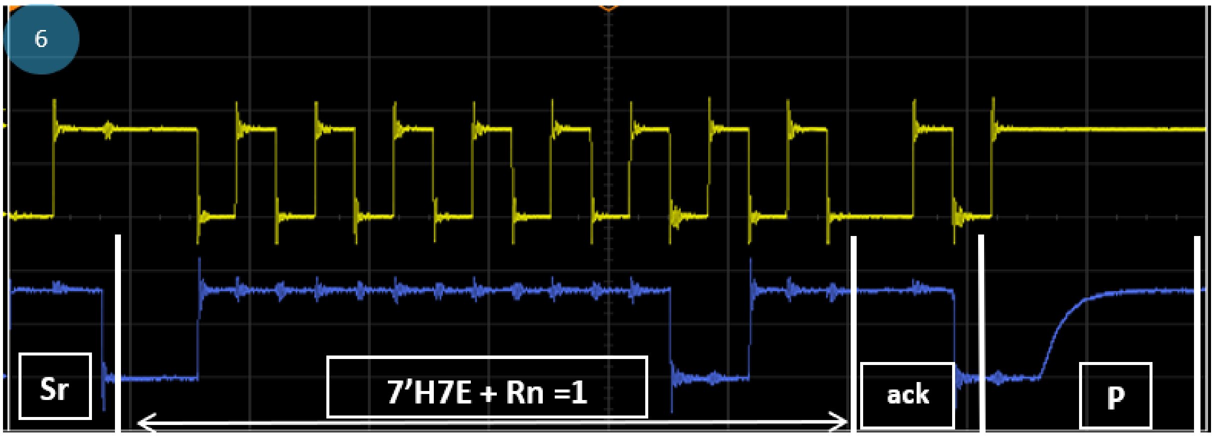


Figure 34. **End DAA procedure waveform**

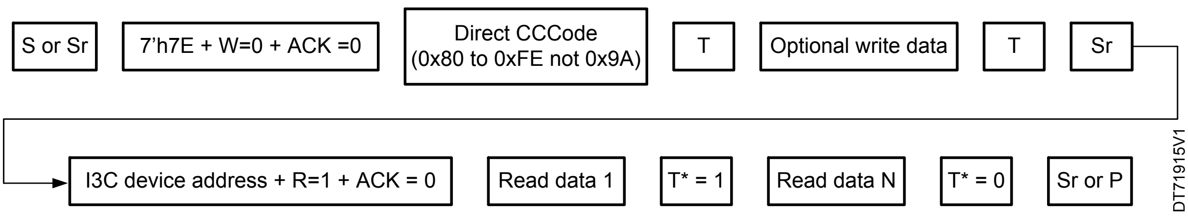
9.7 Direct read

This section contains the steps of usage of a typical direct read based on the 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. The Figure 35 shows the controller issuing a direct read to one target on the bus. All commands have the same flow for the direct CCC.

Figure 35. **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 with defining byte

9.7.1.1 Software settings

Table 13. Software settings with defining byte

```
/* Descriptor for direct read CCC */
I3C_CCCTypeDef cccbuff1[] =
{
/* Target Addr ,CCC Value ,CCC data + defbyte pointer  CCC size + defbyte ,Direction */
    {TARGET1_DYN_ADDR, 0x8F ,{NULL,6},HAL_I3C_DIRECTION_READ},
};
```

- I3C CCCTypeDef structure definition
- GETDCR: 0x8F

```
  aContextBuffers[I3C_IDX_FRAME_1].CtrlBuf.pBuffer = ControlBuffer;
  aContextBuffers[I3C_IDX_FRAME_1].CtrlBuf.Size    = 1;
  aContextBuffers[I3C_IDX_FRAME_1].TxBuf.pBuffer   = TxBuffer;
  aContextBuffers[I3C_IDX_FRAME_1].TxBuf.Size      = 1;

  aContextBuffers[I3C_IDX_FRAME_2].CtrlBuf.pBuffer = ControlBuffer;
  aContextBuffers[I3C_IDX_FRAME_2].CtrlBuf.Size    = 1;
  aContextBuffers[I3C_IDX_FRAME_2].RxBuf.pBuffer   = RxBuffer;
  aContextBuffers[I3C_IDX_FRAME_2].RxBuf.Size      = 1;
```

- Prepare context buffers process
  * uint32_t ControlBuffer[0xFF] : Buffer used by HAL to compute control data for the direct communication

```
if (HAL_I3C_AddDescToFrame(&hi3c1,
                              &cccbuff1[I3C_IDX_FRAME_1],
                              NULL,
                              &buff[I3C_IDX_FRAME_1],
                              1,
                              I3C_DIRECT_WITH_DEFBYTE_RESTART) != HAL_OK)
 {
   Error_Handler();
 }

while ( HAL_I3C_Ctrl_TransmitCCC_IT( &hi3c1 ,&buff[I3C_IDX_FRAME_1]) != HAL_OK)
 {
     Error_Handler();
 }
```

- Add CCC descriptor in the user data transfer descriptor for transmit with defining byte. For some Direct CCCs, a Defining Byte is optional; for others, it is always required.
- The controller transmit direct CCC command in interrupt mode

```
  while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)
 {

 }

  if (HAL_I3C_AddDescToFrame(&hi3c1,
                             &cccbuff1[I3C_IDX_FRAME_2],
                             NULL,
                             &buff[I3C_IDX_FRAME_2],
                             1,
                             I3C_DIRECT_WITH_DEFBYTE_STOP) != HAL_OK)
 {
  Error_Handler();
 }

  while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_RESET)
 {
 }
  while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_SET)
 {
 }

 while ( HAL_I3C_Ctrl_Receive_IT( &hi3c1 ,&buff[I3C_IDX_FRAME_2]) != HAL_OK)
 {
    Error_Handler();
 }
```

- Add CCC descriptor in the user data transfer descriptor for reception
- Controller read the data in interrupt mode.

```c
/* USER CODE BEGIN 4 */
void HAL_I3C_TgtReqDynamicAddrCallback(I3C_HandleTypeDef *hi3c, uint64_t targetPayload)
{
  TargetDesc1.TARGET_BCR_DCR_PID = targetPayload;

  HAL_I3C_Ctrl_SetDynAddr(hi3c, TargetDesc1.DYNAMIC_ADDR);
}

void HAL_I3C_CtrlDAACpltCallback(I3C_HandleTypeDef *hi3c)
{
  BSP_LED_On(LED2);
}

void HAL_I3C_CtrlTxCpltCallback(I3C_HandleTypeDef *hi3c)
{
  BSP_LED_Toggle(LED2);
}

void HAL_I3C_CtrlRxCpltCallback(I3C_HandleTypeDef *hi3c)
{
  BSP_LED_Toggle(LED2);
}

/* USER CODE END 4 */
```

- I3C target request a dynamic address callback.
- Controller dynamic address assignment Complete callback.
- Controller transmit/receive complete callback.

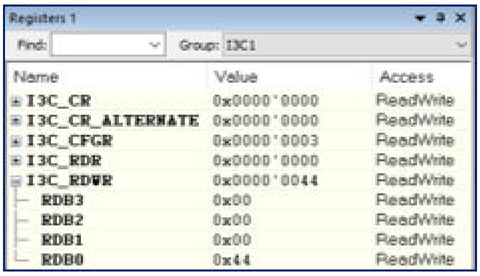
IAR debug

- Select: I3C1 register
- Select I3C_RDWR register

In this example, we are using GETDCR (0x8F) code as a command code.

- GetDCR Code (0x8F) ⇨ 0x44

Figure 36. **I3C_RDWR register**



Name	Value	Access
I3C_CR	0x0000'0000	ReadWrite
I3C_CR_ALTERNATE	0x0000'0000	ReadWrite
I3C_CFGR	0x0000'0003	ReadWrite
I3C_RDR	0x0000'0000	ReadWrite
I3C_RDWR	0x0000'0044	ReadWrite
RDB3	0x00	ReadWrite
RDB2	0x00	ReadWrite
RDB1	0x00	ReadWrite
RDB0	0x44	ReadWrite

9.7.1.2 *Waveform results*

First setup (SDA/SCL): (GETDCR)

The following scope screenshots illustrate this first setup.

Figure 37. **SDR DIRECT CCC GETDCR**

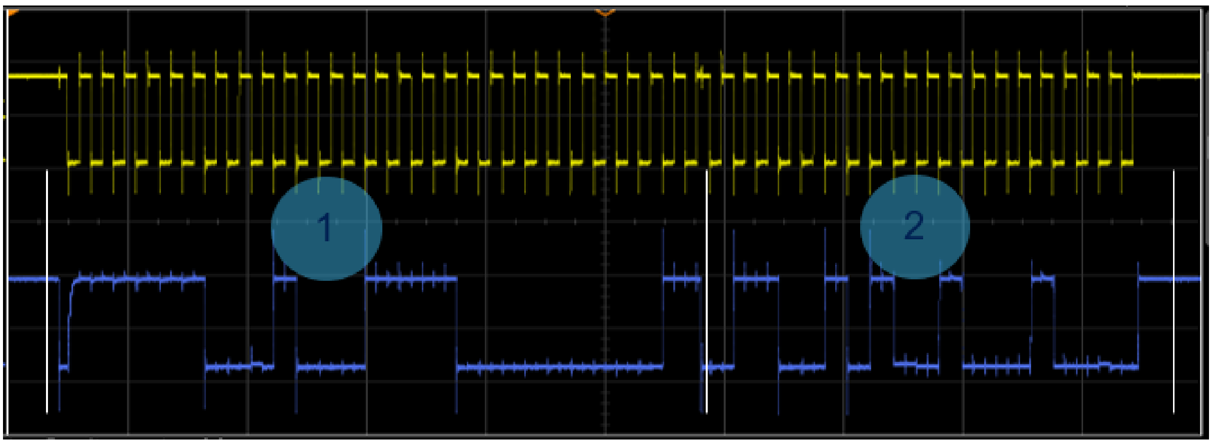
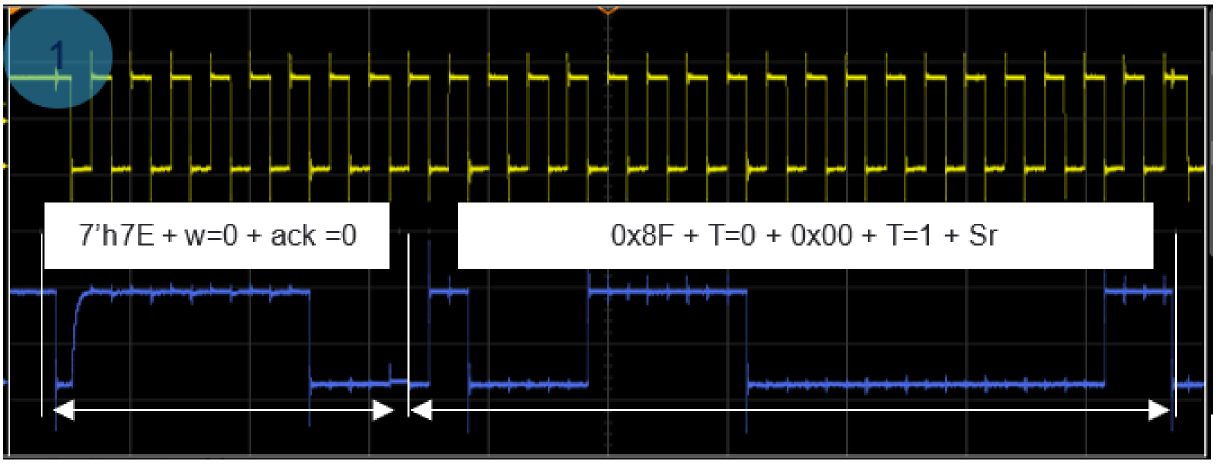


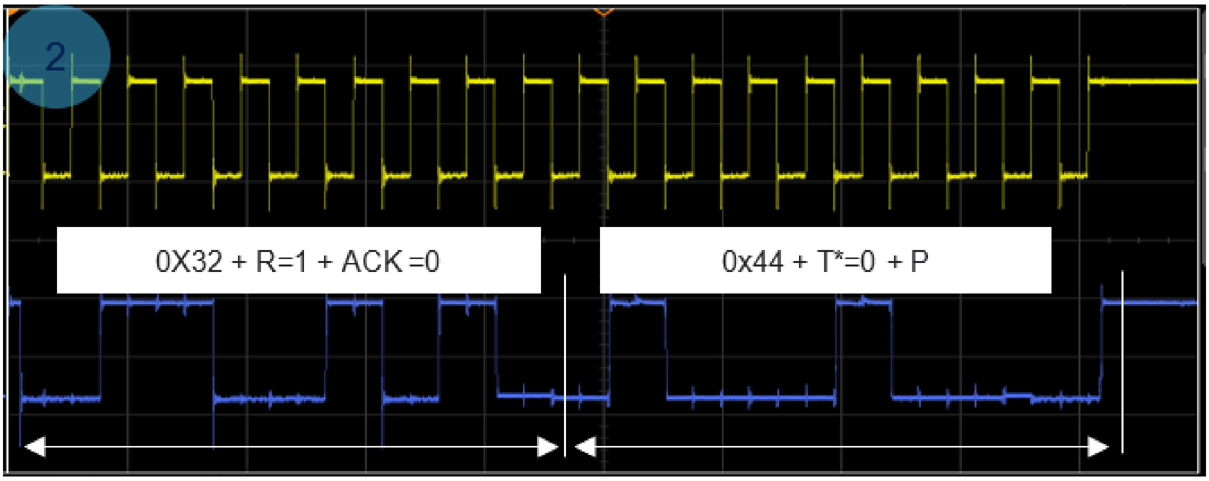
Figure 38. **broadcast address (7’h7E) + W =0 + ACK =0 + direct CCC for GETDCR (0x8F) (T=0) +optional write data = 0x00 (T=1) + Sr**



Where:

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

Figure 39. **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.

Second setup (SDA/SCL): (GETMXDS)

The following scope screenshots illustrate the second setup.

Figure 40. **SDR DIRECT CCC GETMXDS**

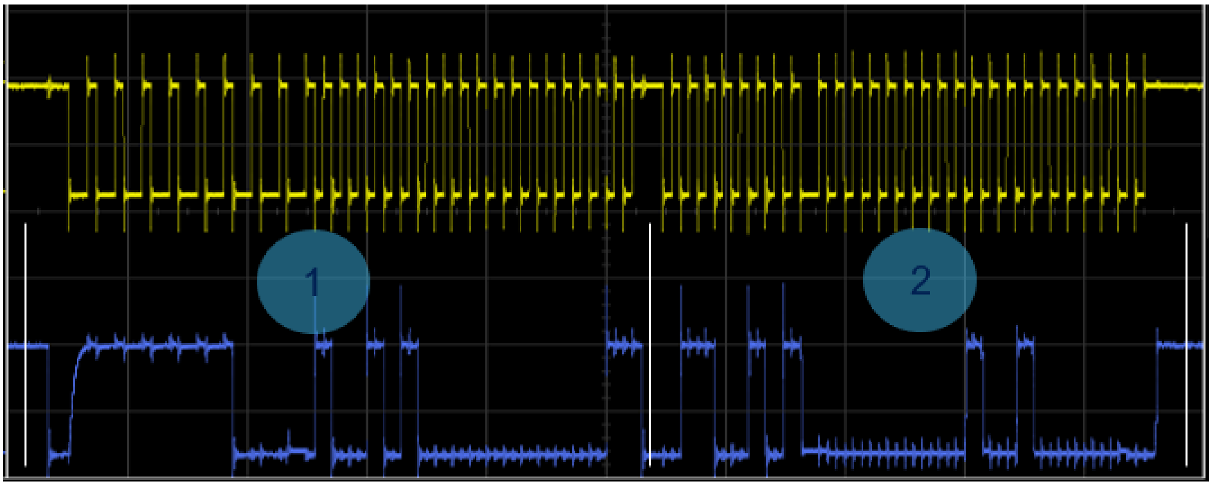
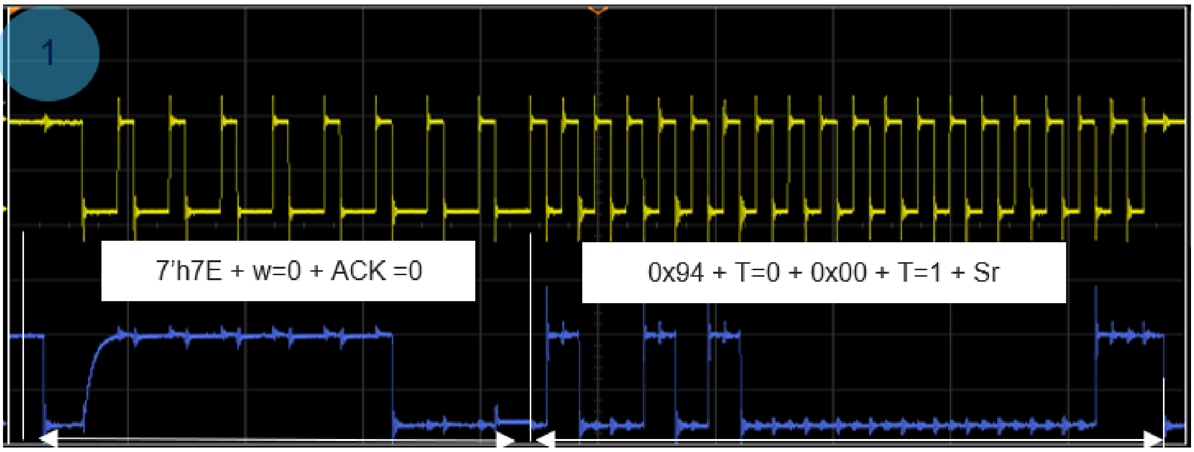


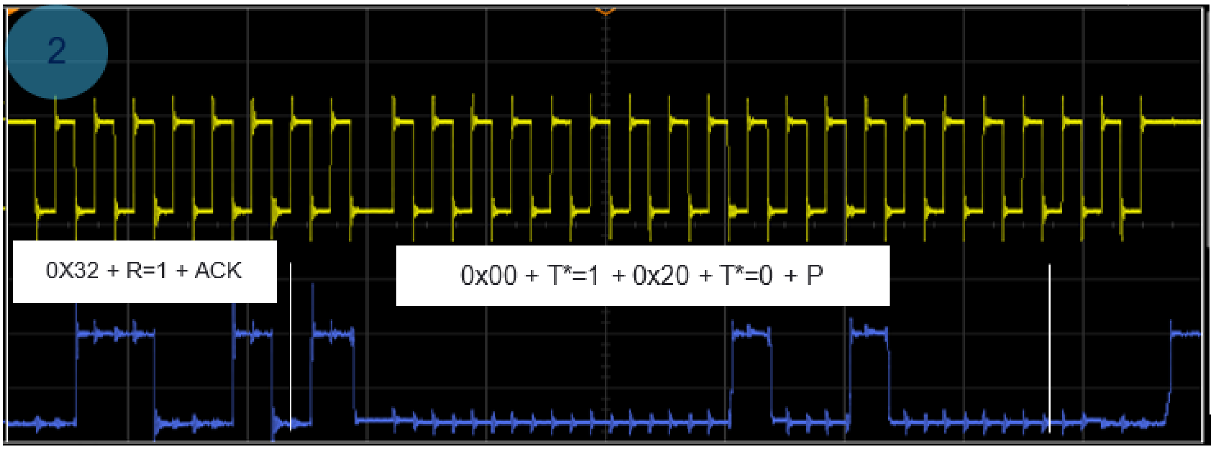
Figure 41. **Broadcast address (7’h7E) + W =0 + ACK = 0 + direct CCC for GETDCR (0x94) (T=0) + optional write data = 0x00 (T=1) + Sr**



Where:

- T: SDR controller written data as parity (as an odd parity): 0x94: T-bit value is be 0.
- T: SDR controller written data as parity (as an odd parity): 0x00: T-bit value is 1.

Figure 42. **Device dynamic address 0x32 + R = 1 + ACK = 0 + 0x00 + T* = 1 + 0x20 + T* = 0 + P**



Where:

- T*: SDR target returned (read) data: As end-of-data: T*=1⇨ Continue the message.
- T*: SDR target returned (read) data: As end-of-data: T*=0 ⇨ End of data.

9.7.2 Direct read without defining byte

9.7.2.1 *Software settings (“Direction: without defining byte GETBCR)*

Figure 43. **Software settings without defining byte**

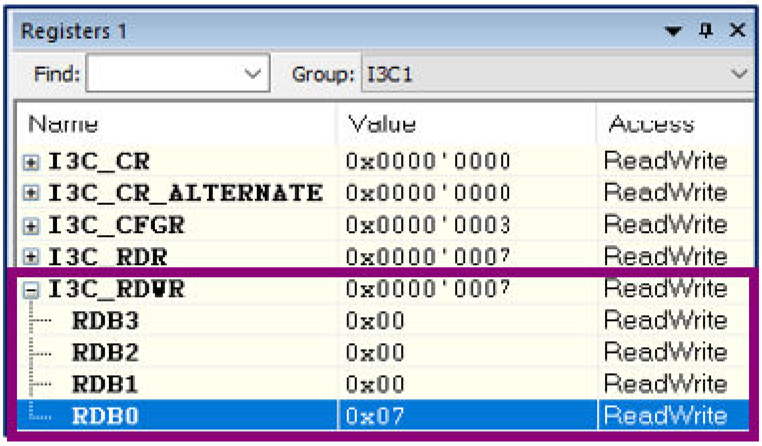
```
if (HAL_I3C_AddDescToFrame(&hi3c1,
                           &cccbuff1[I3C_IDX_FRAME_1],
                           NULL,
                           &buff[I3C_IDX_FRAME_1],
                           1,
                           I3C_DIRECT_WITH_DEFBYTE_RESTART) != HAL_OK)
{
  Error_Handler();
}


if (HAL_I3C_Ctrl_TransmitCCC_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_1]) != HAL_OK)
{
  Error_Handler();
}
```

- Add CCC descriptor in the user data transfer descriptor for transmit **without defining byte**.

IAR debug

- Select: I3C1 register
- Select I3C_RDWR register
- GetBCR c'\ode (0x8E) ⇨ 0x07

Figure 44. **IAR debug**


Registers 1

Find: Group: I3C1

Name	Value	Access
I3C_CR	0x0000'0000	ReadWrite
I3C_CR_ALTERNATE	0x0000'0000	ReadWrite
I3C_CFGR	0x0000'0003	ReadWrite
I3C_RDR	0x0000'0007	ReadWrite
I3C_RDWR	0x0000'0007	ReadWrite
RDB3	0x00	ReadWrite
RDB2	0x00	ReadWrite
RDB1	0x00	ReadWrite
RDB0	0x07	ReadWrite

9.7.2.2 *Waveform results*

Figure 45. **Waveform CCC direct read without DEFBYTE GETBCR**

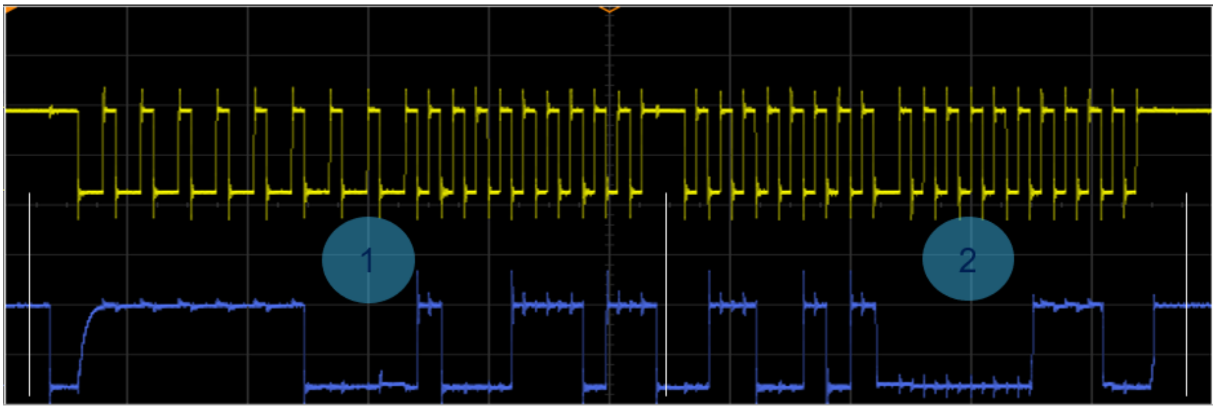
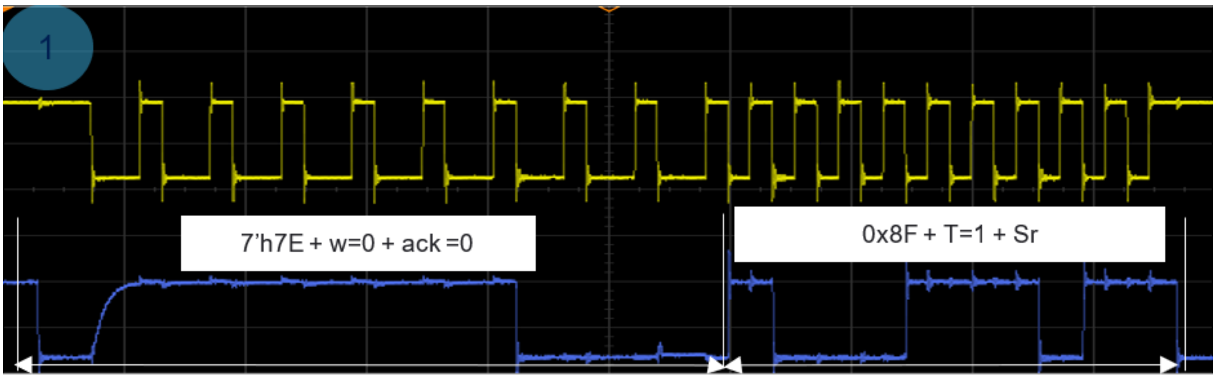


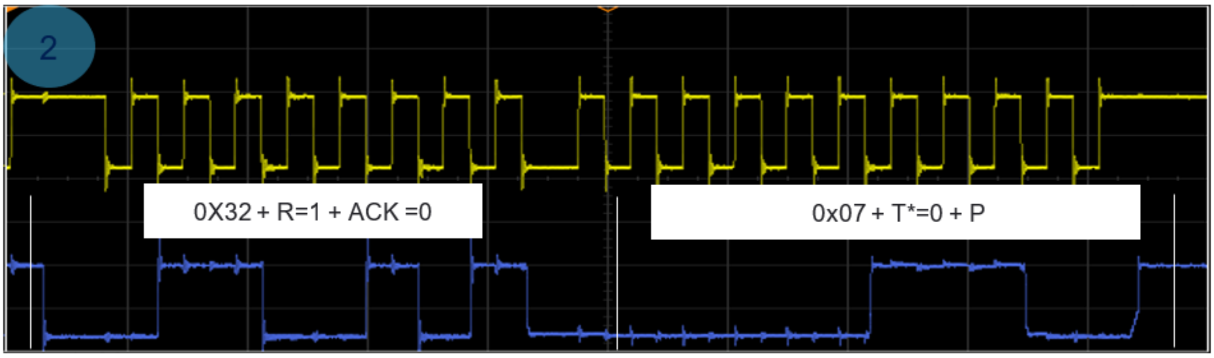
Figure 46. **B""\roadcast address (7'h7E) + W = 0 + ACK = 0 + direct CCC for GETDCR (0x8F) (T = 1)**



Where:

- T: SDR controller written data as parity: as an odd parity: 0x8F: T-bit value is 1.

Figure 47. Sr + device dynamic address 0x32 + R = 1 + ACK = 0 + 0x07 + 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 LSM6DSO registers (WHO_AM_I: address 0x0F) using I3C protocol in SDR mode base on STM32CubeMX.

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

9.8.1 Software settings

Table 14. Software settings with defining byte

<table>
<tr><td><code>uint8_t aTxBuffer[1] ={0x0F}; //0x6c</code>
<code>uint8_t aRxBuffer[1];</code>
<code>TargetDesc_TypeDef *aTargetDesc[1] = \</code>
<code>{</code>
<code>&TargetDesc1 /* TARGET_ID1 */</code>
<code>};</code>
<code>I3C_PrivateTypeDef aPrivateDescriptor[2] = \</code>
<code>{</code>
<code>{TARGET1_DYN_ADDR, {aTxBuffer, 1}, {NULL, 0}, HAL_I3C_DIRECTION_WRITE},</code>
<code>{TARGET1_DYN_ADDR, {NULL, 0}, {aRxBuffer, 1}, HAL_I3C_DIRECTION_READ}</code>
<code>};</code></td><td>• aTxBuffer: buffer used for transmission (value = reg address= 0x0F).
aRxBuffer: buffer used for reception.
aPrivateDescriptor: 2 directions read and write.</td></tr>
<tr><td><code>aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.pBuffer = aControlBuffer;</code>
<code>aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size = 1;</code>
<code>aContextBuffers[I3C_IDX_FRAME_1].TxBuff.pBuffer = aTxBuffer;</code>
<code>aContextBuffers[I3C_IDX_FRAME_1].TxBuff.Size = TXBUFFERSIZE;</code>

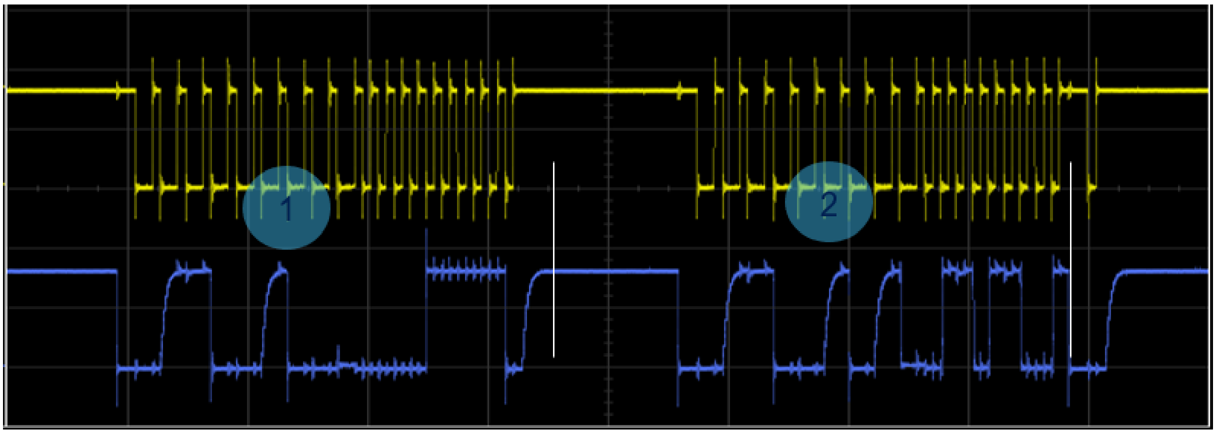
<code>aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.pBuffer = aControlBuffer;</code>
<code>aContextBuffers[I3C_IDX_FRAME_2].CtrlBuff.Size = 1;</code>
<code>aContextBuffers[I3C_IDX_FRAME_2].RxBuff.pBuffer = aRxBuffer;</code>
<code>aContextBuffers[I3C_IDX_FRAME_2].RxBuff.Size = RXBUFFERSIZE;</code></td><td>• Prepare context buffers process</td></tr>
<tr><td><code>if (HAL_I3C_AddDescToFrame(&hi3c1,</code>
<code>NULL,</code>
<code>&aPrivateDescriptor[I3C_IDX_FRAME_1],</code>
<code>&aContextBuffers[I3C_IDX_FRAME_1],</code>
<code>aContextBuffers[I3C_IDX_FRAME_1].CtrlBuff.Size,</code>
<code>I3C_PRIVATE_WITH_ARB_RESTART) != HAL_OK)</code>
<code>{</code>
<code>Error_Handler();</code>
<code>}</code>
<code>if (HAL_I3C_Ctrl_Transmit_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_1]) != HAL_OK)</code>
<code>{</code>
<code>Error_Handler();</code>
<code>}</code></td><td>• Add context buffer transmit in Frame context
• Transmit private data process</td></tr>
</table>

<pre>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,
                          I3C_PRIVATE_WITH_ARB_STOP) != HAL_OK)
{
  Error_Handler();
}
 	
if (HAL_I3C_Ctrl_Receive_IT(&hi3c1, &aContextBuffers[I3C_IDX_FRAME_2]) != HAL_OK)
{
  Error_Handler();
}</pre>
	• Check the current state of the peripheral
	• Add context buffer receive in Frame context
	• Receive private data process

9.8.2 Waveform result

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.

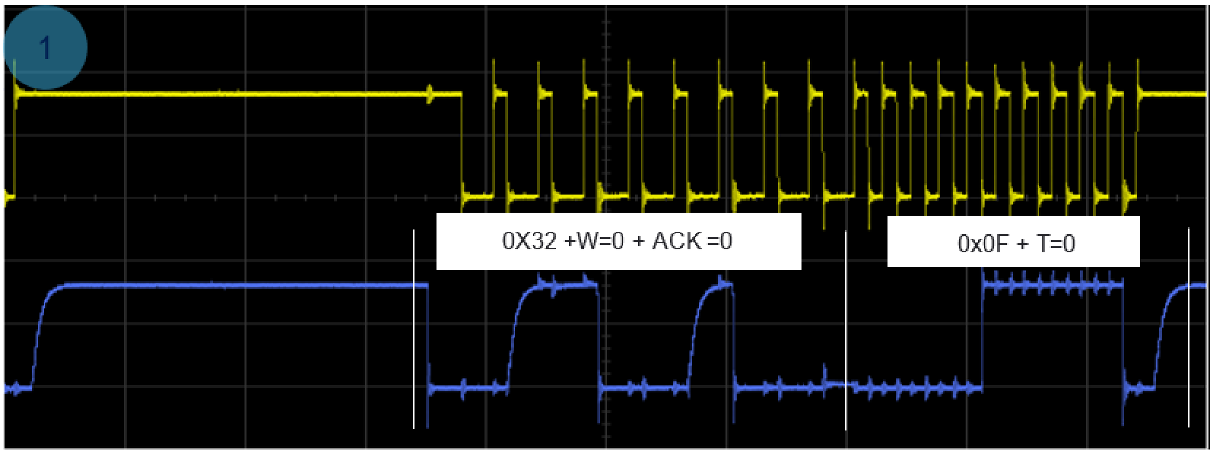
Figure 48. After a start the controller sends the target address with a write followed by the register address + T = 1 (parity bit as an odd parity)



Where:

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

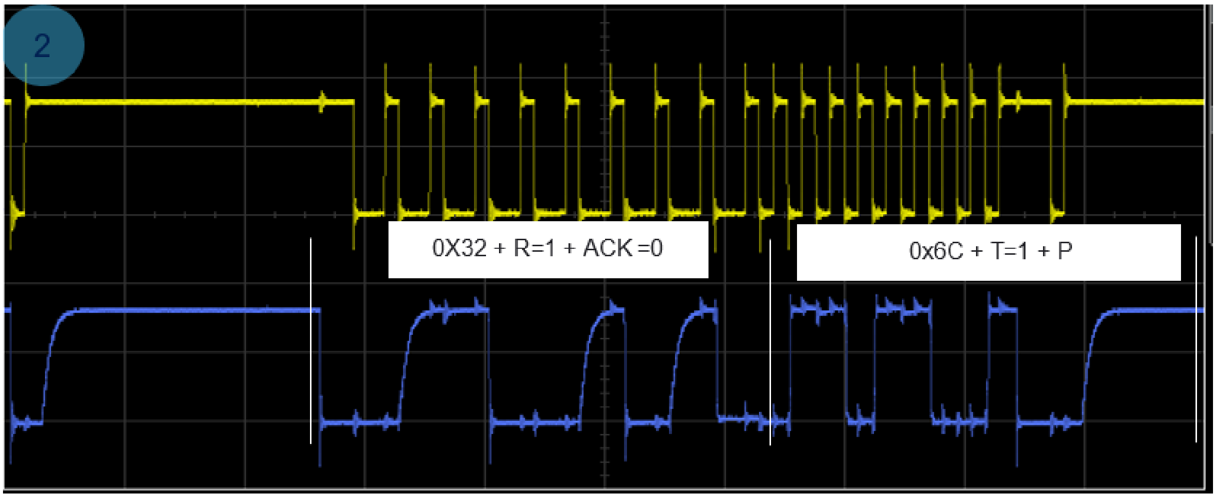
Figure 49. After a repeated start, target address + R = 1 followed by the data (0X6C) + T = 1 +stop



Where:

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

Figure 50. **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.

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

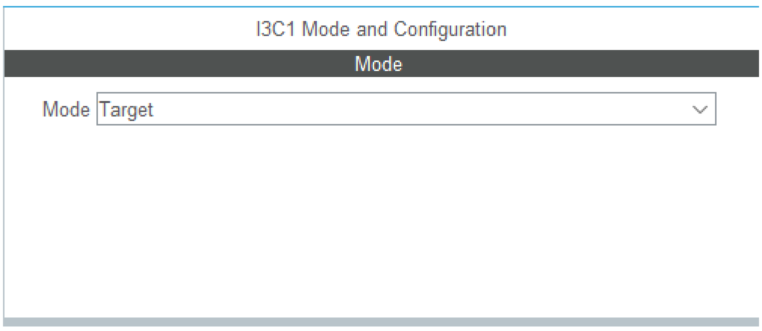
For the IBI configuration, keep the same previous configuration for the controller.

Target settings

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

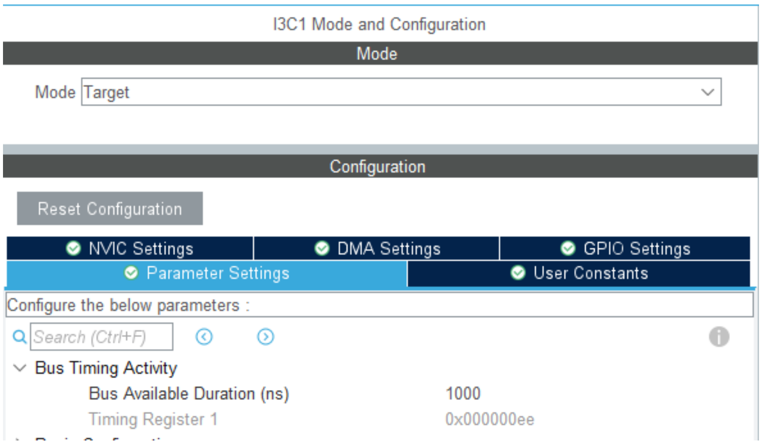
- Open STM32CubeMX and select “Access to MCU selector"
- Click on “Connectivity and select “I3C1”
- Select “target” mode.

Figure 51. **Target mode for IBI**



Bus timing activity

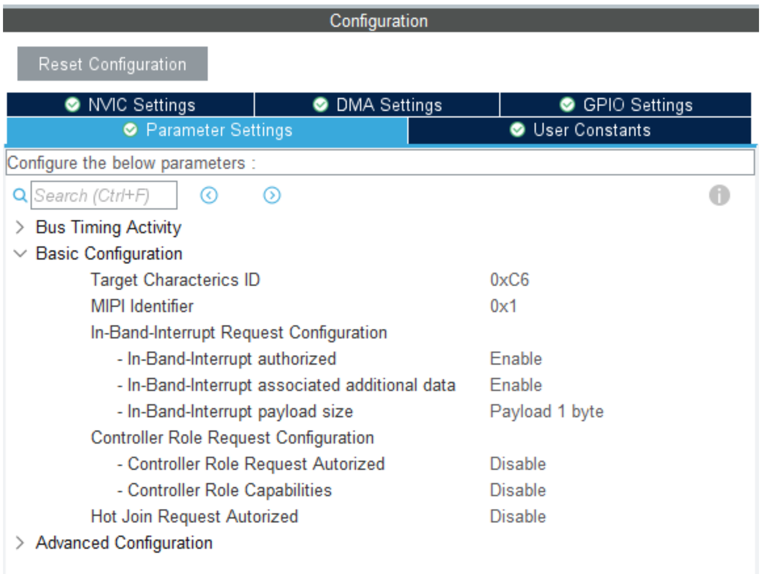
Figure 52. **Bus timing activity**



- 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 us = 1000 ns

Basic configuration

Figure 53. **Basic configuration**

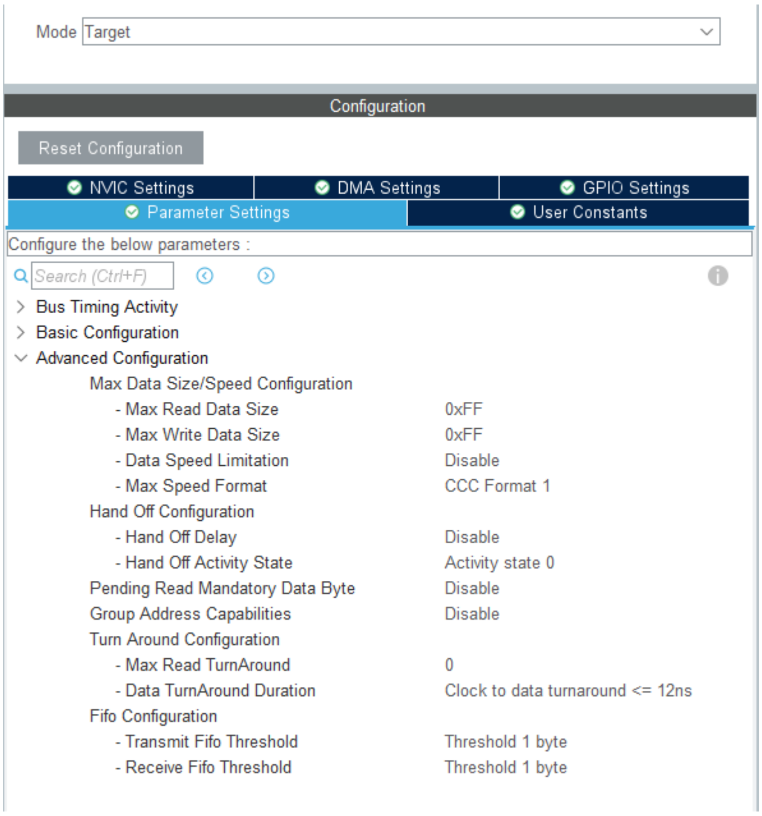


Where:

- Target characteristics ID: 0xC6
  - This value means that the device is a microcontroller. For more information, see 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 54. 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 setting

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 retrieve Target Dynamic Address with associated data if any through HAL_I3C_GetCCCInfo ().

Figure 55. Controller software settings - first part

```
 while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_RESET)
{
}
  while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_SET)
{
}

if (HAL_I3C_Ctrl_DynAddrAssign_IT(&hi3c1, I3C_RSTDAA_THEN_ENTDAA) != HAL_OK)
{
  Error_Handler();
}


while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)
{
}

/* Fill Device descriptor for target detected during ENTDAA procedure */

  DeviceConf[ubTargetIndex].DeviceIndex        = 1;
  DeviceConf[ubTargetIndex].TargetDynamicAddr  = TargetDesc1.DYNAMIC_ADDR;
  DeviceConf[ubTargetIndex].IBIAck             = __HAL_I3C_GET_IBI_CAPABLE(__HAL_I3C_GET_BCR(TargetDesc1.TARGET_BCR_DCR_PID));
  DeviceConf[ubTargetIndex].IBIPayload         = __HAL_I3C_GET_IBI_PAYLOAD(__HAL_I3C_GET_BCR(TargetDesc1.TARGET_BCR_DCR_PID));
  DeviceConf[ubTargetIndex].CtrlRoleReqAck     = __HAL_I3C_GET_CR_CAPABLE(__HAL_I3C_GET_BCR(TargetDesc1.TARGET_BCR_DCR_PID));
  DeviceConf[ubTargetIndex].CtrlStopTransfer   = DISABLE;

  if (HAL_I3C_Ctrl_ConfigBusDevices(&hi3c1, &DeviceConf[ubTargetIndex], 1U) != HAL_OK)
  {
    Error_Handler();
  }


if( HAL_I3C_ActivateNotification(&hi3c1, HAL_I3C_IT_IBIIE) != HAL_OK)
{
  Error_Handler();
}
```

Figure 56. Controller software settings - second part

```
 while (1)
 {
   /*Start the listen mode process*/
   while(uwIBIRequested == 0U)
   {
   }

   /*Getting the information from the last IBI request*/
   if (HAL_I3C_GetCCCInfo(&hi3c1, EVENT_ID_IBI, &CCCInfo) != HAL_OK)
   {
     Error_Handler();
   }
   else
   {
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
   }

   /* Reset */
   uwIBIRequested = 0U;

 }
 }
```

Figure 57. Controller software settings - third part

```
/* USER CODE BEGIN 4 */

void HAL_I3C_TgtReqDynamicAddrCallback(I3C_HandleTypeDef *hi3c, uint64_t targetPayload)
{
  TargetDesc1.TARGET_BCR_DCR_PID = targetPayload;


  HAL_I3C_Ctrl_SetDynAddr(hi3c, TargetDesc1.DYNAMIC_ADDR);
}


void HAL_I3C_CtrlDAACpltCallback(I3C_HandleTypeDef *hi3c)
{
}


void HAL_I3C_NotifyCallback(I3C_HandleTypeDef *hi3c, uint32_t eventId)
{
  if ((eventId & EVENT_ID_IBI) == EVENT_ID_IBI)
  {
    uwIBIRequested = 1;
  }
  else
  {
    Error_Handler();
  }
}

/* USER CODE END 4 */
```

The third part contains the callback functions:

- The reception of the completion callback HAL_I3C_CtrlDAACpltCallback (): ENTDAA is terminated
- Call the callback HAL_I3C_NotifyCallback (): the IBI event treatment is completed.

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 catch the event and request a private communication with the Target which have sent and have got acknowledge of the In-Band-Interrupt event.

`/* Contain the IBI Payload */`
`uint8_t ubPayloadBuffer[] = {0xAB};`
 	
`/* Variable to catch ENTDAA completion */`
`uint8_t ubDynamicAddressCplt = 0;`
 	
`/* Variable to catch IBI end of process */`
`uint8_t ubIBIcplt = 0;`
	After this starting of the In-Band-Interrupt procedure, the I3C controller catchs the event and requests a private communication with the target that sent and receive confirmation of the In-Band-Interrupt event.

Target software setting

- Variable declaration:
  – PayloadBuffer[] ={0xAB} = 0b 10101011 (in this case the user can set a free payload value)

Figure 58. Target software settings - first part

```
/* USER CODE BEGIN 2 */

if (HAL_I3C_ActivateNotification(&hi3c1, (HAL_I3C_IT_DAUPDIE | EVENT_ID_IBIEND)) != HAL_OK)
{
  Error_Handler();
}

  while (ubDynamicAddressCplt != 1)
{
}

/* USER CODE END 2 */
```

Figure 59. Target software settings - second part

```
 while (1)
 {

  while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_RESET)
 {
 }
   while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_SET)
 {
 }

   /*Send IBI request*/
   if(HAL_I3C_Tgt_IBIReq_IT(&hi3c1, ubPayloadBuffer, COUNTOF(ubPayloadBuffer)) != HAL_OK)
   {
     Error_Handler();
   }

   /* Wait for IBI process ending */

   while(ubIBIcplt == 0U)
   {
   }

   /* Reset*/
   ubIBIcplt = 0U;

}
}
```

Figure 60. Target software settings - third part

```
/* USER CODE BEGIN 4 */

void HAL_I3C_NotifyCallback(I3C_HandleTypeDef *hi3c, uint32_t eventId)
{
  if ((eventId & EVENT_ID_DAU) == EVENT_ID_DAU)
  {
    ubDynamicAddressCplt  = 1;
  }

  if ((eventId & EVENT_ID_IBIEND) == EVENT_ID_IBIEND)
  {

    ubIBIcplt = 1;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
}
}

/* USER CODE END 4 */
```

Note:
- *The end of reception of a DA is HAL_I3C_NotifyCallback () on Target side.*
- *The end of IBI communication is HAL_I3C_NotifyCallback () on Target side.*

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 61. 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

9.9.2.1 *Waveform result*

Figure 62. Waveform illustrates example of IBI with 0xAB mandatory byte

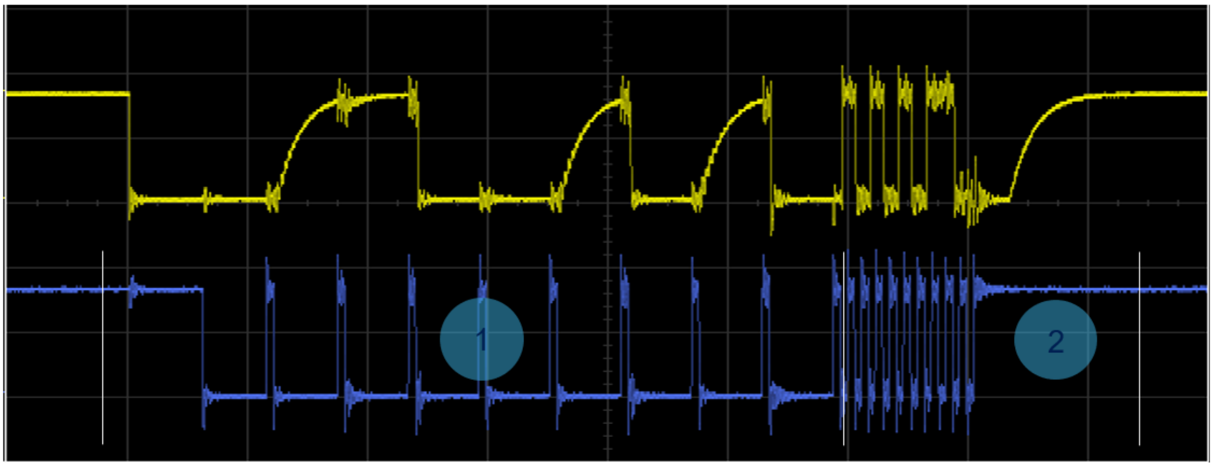


Figure 63. After the bus available condition, the controller emit a Start and the target provide its dynamic address with a read in open-drain mode, the controller ACKed

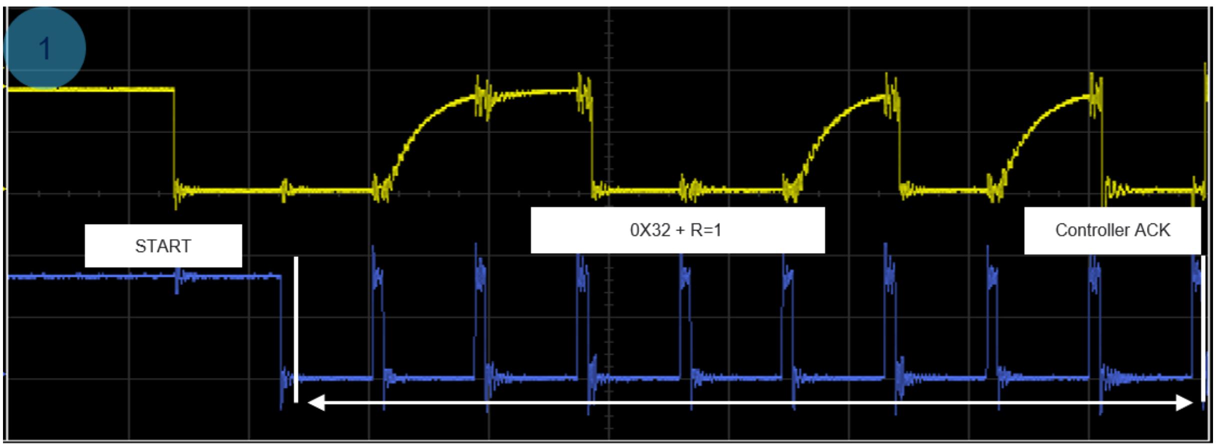
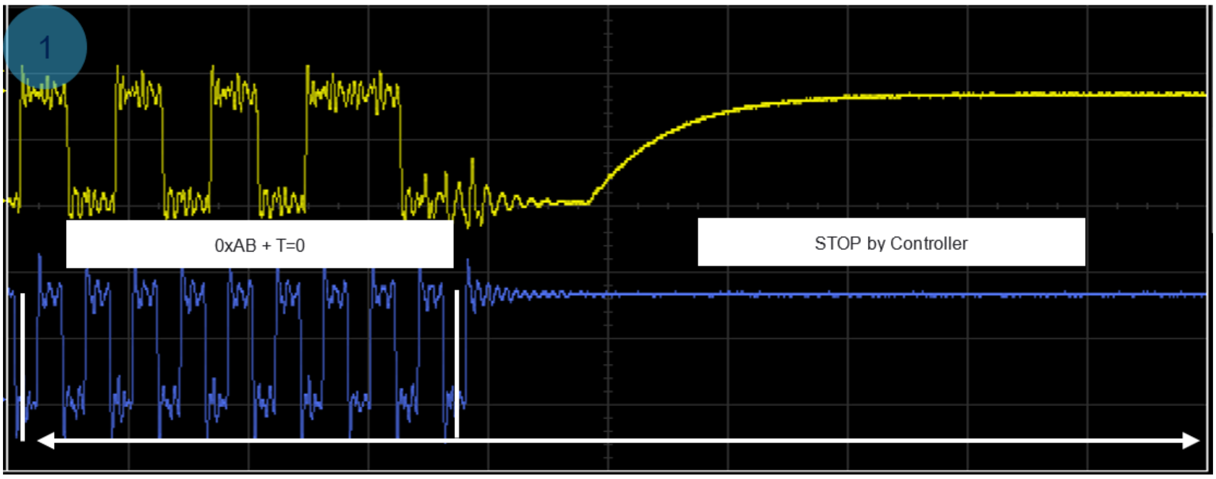


Figure 64. **The mandatory data byte in Push-Pull mode + T=0**



Where:

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

9.10 Mixed communication

This use case contains two parts:

- **Active controller I3C 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 controller I3C, 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).

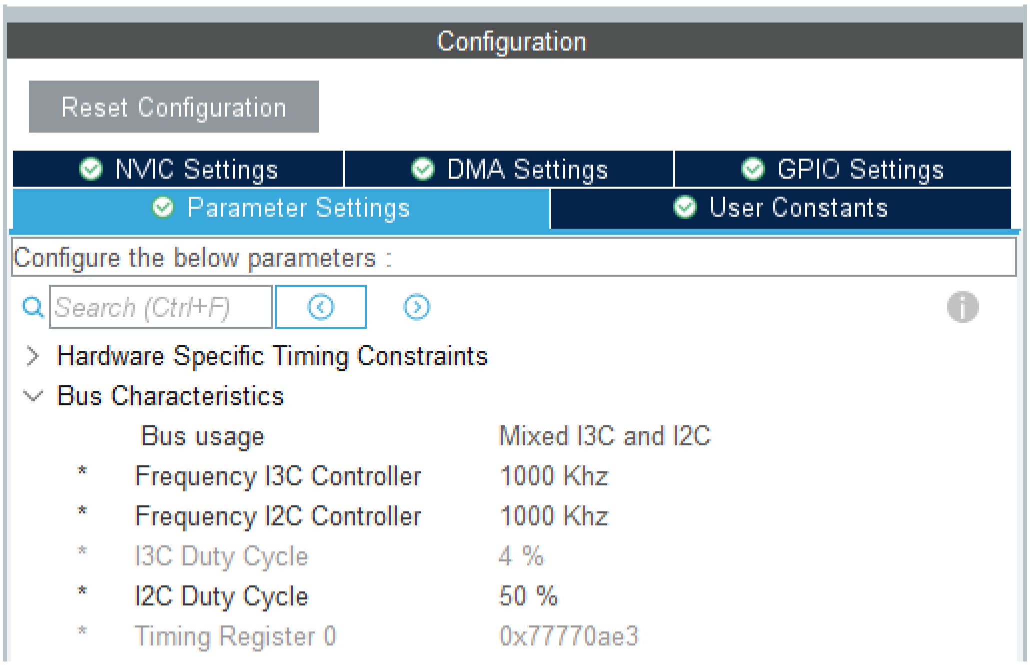
Here is the configuration based on STM32 CubeMX:

- 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 65. 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 controller I3C and legacy I2C 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 15. Software settings I3C/I2C

<table>
<tr>
<td>
<pre>
I3C_XferTypeDef aContextBuffers[2];

uint32_t aControlBuffer[0xF];

uint8_t TxBuffer[1] ={0x0F};

uint8_t RxBuffer[1];

I3C_PrivateTypeDef aPrivateDescriptor[2] = \
                                         {
                                           {0x19, {TxBuffer, 1}, {NULL, 0}, HAL_I3C_DIRECTION_WRITE},
                                           {0x19, {NULL, 0}, {RxBuffer, 1}, HAL_I3C_DIRECTION_READ}
                                         };
</pre>
</td>
<td>
• aTxBuffer: buffer used for transmission (register WHO_AM_I: 0x0F)
• aRxBuffer: buffer used for reception.
• aPrivateDescriptor: 2 directions Read and Write. (0x19 static address)
• Prepare context buffers process
</td>
</tr>
<tr>
<td>
<pre>
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;
</pre>
</td>
<td>
• Prepare context buffers process
• Add context buffer transmit in frame context
</td>
</tr>
<tr>
<td>
<pre>
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();
}
</pre>
</td>
<td>
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
</td>
</tr>
</table>

9.10.2.2 *Waveforms results*

The following scope screenshots illustrate the read of the data from the I2C target (LIS2DW12).

Figure 66. **Legacy I2C read transfer on I3C bus**

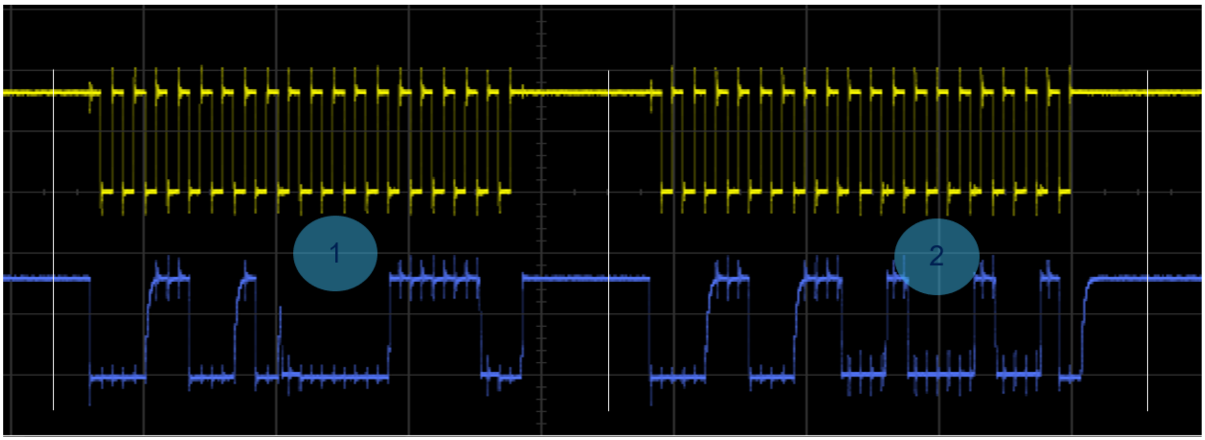
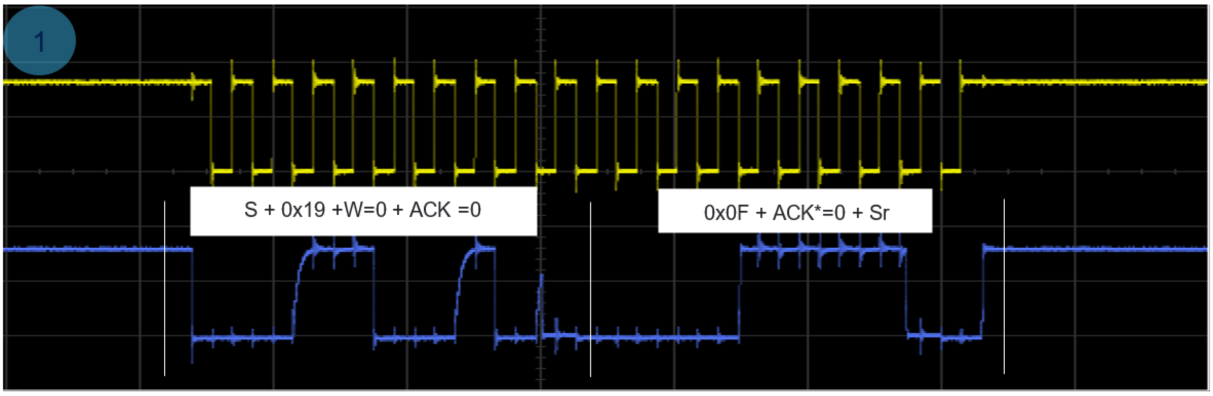


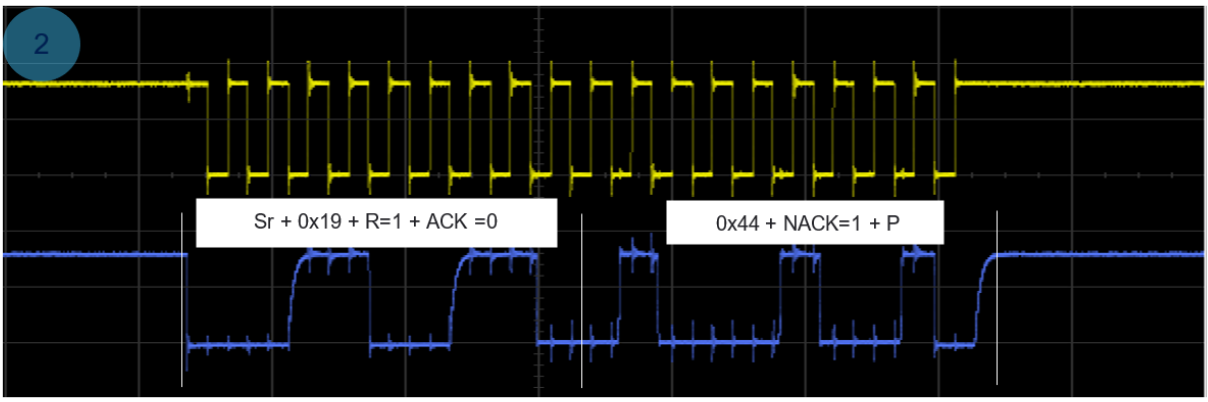
Figure 67. **First step: write the value of the register address (0x0F)**



Where:

- ACK*: From target: end write data: ACK* = 0.

Figure 68. **Second step: Read data from the register 0x0F**



Where:

- NACK: From controller: End-of-data: NACK=1 ⇨ end of data.

9.10.3 I3C active 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 16. Software settings mixed bus: I3C - I3C and I2C

<table>
<tr>
<td>

```c
/* USER CODE BEGIN PV */
I3C_XferTypeDef aContextBuffers[2];
uint32_t aControlBuffer[0xF];
uint8_t TxBuffer[1] ={0x0F};
uint8_t RxBuffer[1];


#define DA_Target        0x32
TargetDesc_TypeDef TargetDesc1 =
{
  "TARGET_ID1",
  TARGET_ID1,
  0x0000000000000000,
  0x00,
  DA_Target,
};



TargetDesc_TypeDef *aTargetDesc[2] = \
                          {
                            &TargetDesc1,       /* TARGET_ID1 */
                          };
I3C_PrivateTypeDef aPrivateDescriptor[2] = \
                                          {
                                            {0x19, {TxBuffer, 1}, {NULL, 0}, HAL_I3C_DIRECTION_WRITE},
                                            {0x19, {NULL, 0}, {RxBuffer, 1}, HAL_I3C_DIRECTION_READ}
                                          };
/* USER CODE END PV */
```

</td>
<td>

- TxBuffer: buffer used for transmission (value: reg address= 0x0F) (WHO_AM_I)
- RxBuffer: buffer used for reception.
- aTargetDesc: target descriptor.
  - (0x32 as DA)
- aPrivateDescriptor: 2 directions Read and Write. (0x19 static address)

</td>
</tr>
<tr>
<td>

```c
  /* USER CODE BEGIN 2 */
    while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_RESET)
  {
  }
    while (HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13) != GPIO_PIN_SET)
  {
  }

    if (HAL_I3C_Ctrl_DynAddrAssign_IT(&hi3c1, I3C_RSTDAA_THEN_ENTDAA) != HAL_OK)
  {
    Error_Handler();
  }


      while (HAL_I3C_GetState(&hi3c1) != HAL_I3C_STATE_READY)
  {
  }
```

</td>
<td>

- Assigning DA in interrupt mode (initiate a RSTDAA follow by a ENTDAA)
- check the current state of the peripheral.
- Prepare context buffers process

</td>
</tr>
<tr>
<td>

```c
  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 */
```

</td>
<td>

- 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

</td>
</tr>
</table>

9.10.3.2 Waveforms results

Figure 69. DAA for I3C device and legacy I2C read transfer on I3C bus

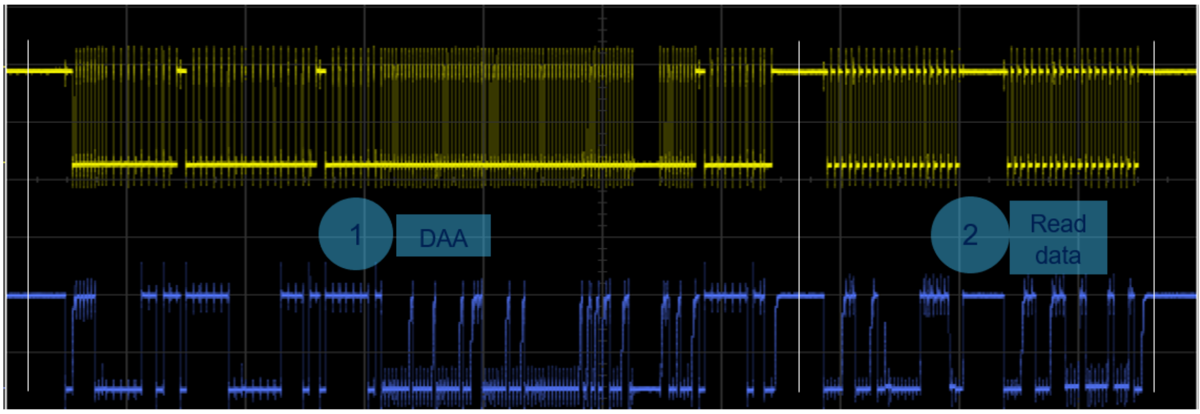


Figure 70. Dynamic address assignment for I3C device

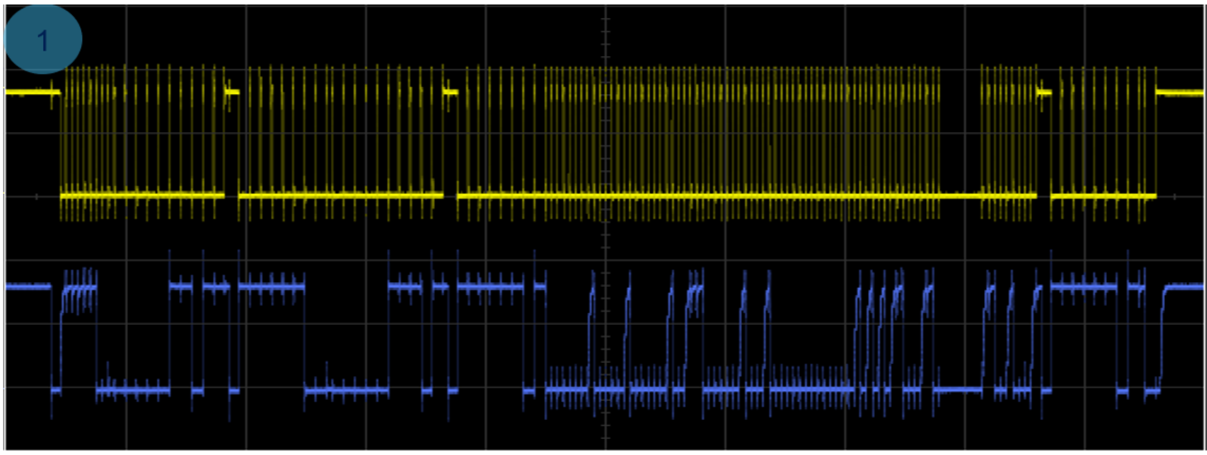
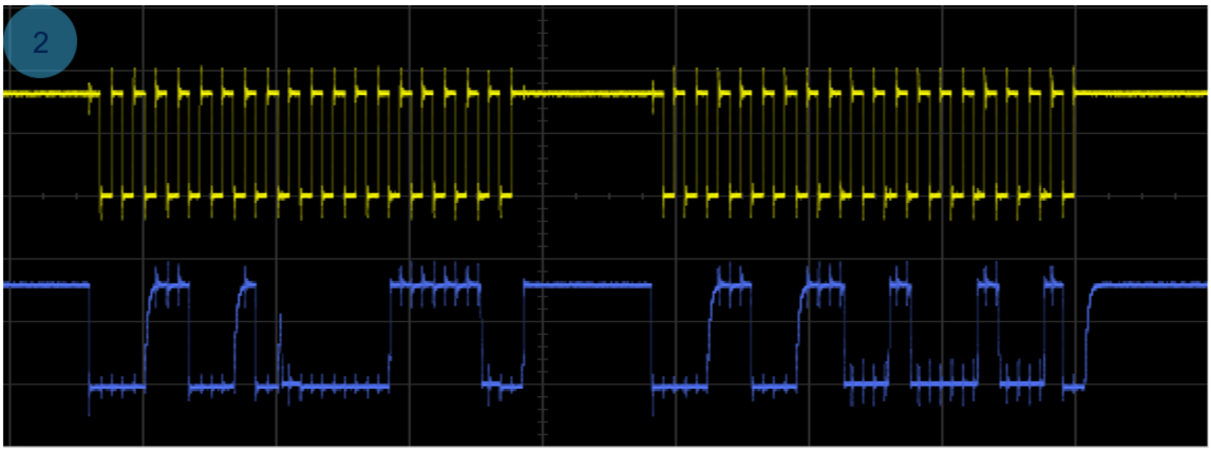


Figure 71. Legacy I2C read transfer



Revision history

Table 17. **Document revision history**

Date	Revision	Changes
13-Mar-2023	1	Initial release.

Contents

List of tables

List of figures